QuickXP 1.0

User's Guide

Any trademarks or registered trademarks used in this document belong to the companies that own them.

Copyright © 2007, Texas Memory Systems, Inc. All rights reserved. No part of this work may be reproduced or used in any form or by any means – graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage or retrieval systems – without the permission of the copyright owner.

Table of Contents

Table of Contents i
Chapter 1 - Introduction 1-1
1.1 Related Texts 1-1
1.2 Typographical Conventions 1-1
1.3 Overview 1-1
1.4 QuickXP 1-2
1.4.1 Pipes 1-2
1.4.2 Modules 1-3
1.4.3 Metadata 1-4
1.5 Using QuickXP 1-5
1.5.1 Using QuickXP in C 1-5
1.5.2 Using QuickXP from the Python Scripting Layer 1-7
1.6 LIBXP 1-7
Chapter 2 - Installation 2-8
Chapter 3 - Programming Guide 3-10
3.1 Module Programming 3-10
3.1.1 Implementing Modules 3-10
3.1.2 Metadata 3-13
3.1.3 Control 3-16
3.1.4 Declaring Modules 3-16
3.1.5 Building Modules 3-17
3.2 Super-Module and Application Programming 3-18
3.2.1 Super-Module and Application Programming in C 3-19
3.2.2 Super-Module and Application Programming in Python 3-19
Chapter 4 - Pipe API 4-22
4.1 Creating and Opening Pipes 4-22
4.1.1 xpq_create 4-22
4.1.2 xpq_create_sized 4-22
4.1.3 xpq_destroy 4-22
4.1.4 xpq_open 4-23
4.1.5 xpq_close 4-23
4.2 Writing to Pipes 4-23
4.2.1 xpq_write_from_host 4-23
4.2.2 xpq_write_from_vp 4-24
4.3 Reading from Pipes 4-24
4.3.1 xpq_read_to_host 4-24
4.3.2 xpq_read_to_vp 4-25
4.4 Writing and Reading Pipe Metadata 4-26
4.4.1 xpq_meta_write 4-26
4.4.2 xpq_meta_read 4-26
4.4.3 xpq_meta_free 4-27
4.4.4 xpq_meta_set_cast_func 4-27
4.5 Metadata Creation 4-27
4.5.1 tms_meta_init 4-27
4.5.2 tms_meta_init_from_file 4-27
4.5.3 tms_meta_datatype_from_string 4-28
4.5.4 tms_meta_double_mag10_from_string 4-28
4.5.5 tms_meta_units_from_string 4-29
4.6 Retrieving Pipe Information 4-29
4.6.1 xpq_get_datasize 4-29
4.6.2 xpq_get_stats 4-29
Chapter 5 - Module API 5-30

5.1 Module Creation 5-30
5.1.1 XPQ_MODULE 5-30
5.1.2 XPQ_TOPLEVEL 5-30
5.1.3 xpq_module_get_name 5-30
5.1.4 xpq_module_get_fullpath 5-31
5.2 Module Control 5-31
5.2.1 xpq_control_addr_init 5-31
5.2.2 xpq_control_addr_get_name 5-31
5.2.3 xpq_control_addr_get_fullpath 5-31
5.2.4 xpq_control_send 5-32
5.2.5 xpq_control_sendto 5-32
5.2.6 xpq_control_receive 5-32
5.2.7 xpq_control_release 5-33
5.2.8 xpq_active 5-33
5.2.9 xpq_stop_all_modules 5-33
5.2.10 xpq_wait_all_modules 5-34
5.3 Module Cleanup 5-34
5.3.1 xpq_module_cleanup_push 5-34
5.3.2 xpq_module_cleanup_pop 5-34
5.4 Module Flow Control 5-34
5.4.1 xpq_control_sleep 5-34
5.4.2 xpq_control_flow 5-35
5.4.3 xpq_elements_per_time 5-35
5.5 Module Error Logging 5-36
5.5.1 xpq_log_error 5-36
Chapter 6 - Scripting API 6-38
6.1 Creating and Opening Pipes 6-38
6.1.1 xpq.create 6-38
6.1.2 xpq.destroy 6-38
6.1.3 xpq.open 6-38
6.1.4 xpq.close 6-39
6.2 Writing to Pipes 6-39
6.2.1 xpq.write_from_host 6-39
6.2.2 xpq.write_from_vp 6-39
6.3 Reading from Pipes 6-40
6.3.1 xpq.read_to_host 6-40
6.3.2 xpq.read_to_vp 6-40
6.4 Writing and Reading Pipe Metadata 6-41
6.4.1 xpq.meta_write 6-41
6.4.2 xpq.meta_read 6-41
6.4.3 xpq.meta_free 6-41
6.5 Metadata Creation 6-42
6.5.1 xpq.meta_init_from_file 6-42
6.6 Retrieving Pipe Information 6-42
6.6.1 xpq.get_stats 6-42
6.7 Module Declaration 6-42
6.7.1 @xpq.FunctionModule 6-42
6.7.2 xpq.ClassModule 6-43
6.8 Module Information 6-43
6.8.1 xpq.module_get_name 6-43
6.8.2 xpq.module_get_fullpath 6-43
6.9 Module Control 6-44
6.9.1 xpq.control_send 6-44
6.9.2 xpq.control_receive 6-44
6.9.3 xpq.control_release 6-44

6.9.4 xpq.active 6-45
6.9.5 xpq.modules.stop 6-45
6.9.6 xpq.modules.wait 6-45
6.10 Module Flow Control 6-45
6.10.1 xpq.control_sleep 6-45
Chapter 7 - Pre Existing Modules 7-47
7.1 File Reader Module 7-47
7.2 File Writer Module 7-47
7.3 Pipe Statistics Module 7-48
7.4 Data Plot Module 7-48
7.5 Data Grid Module 7-48

Chapter 1 - Introduction

Texas Memory Systems, Inc. designs and builds digital signal processing solutions for very high bandwidth real-time environments.

Texas Memory Systems (TMS) digital signal processing (DSP) solutions help solve computation-intensive problems such as hunting for quasars, mapping oil deposits, or enhancing national security.

1.1 Related Texts

The following Texas Memory Systems user manuals may be useful for reference purposes:

LIBXP User Guide

1.2 Typographical Conventions

Text within this document is typed in the same typeface as this paragraph. User input and code is in the same typeface as the example below:

```
xpq_create("inputpipe", "/dev/xp/0");
```

1.3 Overview

Typical scalar processors (Intel, AMD, IBM) that run on most computers are good for general purpose processing but do not fill the needs for extreme digital signal processing. Today's DSP requirements demand additional number crunching power that is not available with scalar processors. TMS has developed cost-effective DSP accelerator boards and systems to address these challenges.

TMS developed the TM-44 ASIC that specializes in DSP vector processing and is easy to program. It executes hundreds of DSP math functions quickly and easily. With its high GFLOPS rating, high-bandwidth, and low power, it provides maximum DSP performance per square inch, per watt, and per dollar.

The TM-44 chip is the basis of our XP accelerator cards, the XP-30 and XP-35. They integrate with existing computer systems, working as back-end DSP number crunchers. Raw data is fed to TMS accelerators and processed results are pushed to host computers, minimizing their workload while maximizing their capabilities. Our solutions turn workstations and servers into real-time supercomputers.

XP accelerator cards are deployed in many digital signal processing applications, achieving increased power through a parallel processor architecture. TMS offers flexible software development tools that allow customers to harness and manage the power of parallel processing.

QuickXP, supplied by TMS, manages the synchronization needed for parallel processing. With QuickXP, powerful applications developed for one TMS platform operate efficiently on any other. QuickXP is surprisingly simple and easy to learn. This manual introduces our QuickXP toolset and describes how it is used to:

- ✓ Script a multi-process application
- ✓ Optimize computer resources for DSP
- ✓ Process high-bandwidth streams of data
- ✓ Code complicated algorithms to run on DSP hardware easily and seamlessly.

QuickXP makes easy solutions out of complicated problems.

1.4 QuickXP

QuickXP helps programmers easily harness and manage TMS DSP power in a Linux environment. QuickXP removes DMA bookkeeping while minimizing latency and maximizing processing bandwidth. It runs massively paralleled applications, but codes like a simple program. And it facilitates easy prototyping with an intuitive scripting layer, mirroring the programming layer so well that scripted applications can be converted to compiled applications in seconds. Finally, QuickXP is not difficult to learn; a new user can do meaningful work quickly, because all the complication has been hidden and abstracted away.

QuickXP provides a simple C and script interface for the programmer and assists in transferring data into and out of data control abstractions called pipes. Pipes manage data bookkeeping, simplifying the DSP accelerator architecture.

Possibly the biggest advantage of QuickXP is its ability to abstract synchronization issues inherent to the interaction of multiple DSP accelerator nodes (also called VPs, for “Vector Processors”). Although each VP executes operations in the order issued, synchronization concerns occur because VP nodes operate asynchronously from the local processor and other VP nodes. This asynchronous design increases performance by hiding the overhead of calling VP routines on the local processor and by allowing multiple VP nodes to be assigned work and process data simultaneously. QuickXP performs synchronization by managing data flow via pipes. Each asynchronous task is separated into program units called modules that are linked together by pipes. Data flows from module to pipe to module.

A QuickXP application is made up of 3 basic components:

- Pipes
- Modules
- Metadata

A QuickXP programmer needs to have a basic understanding of these concepts. The following sections provide an overview of each.

1.4.1 Pipes

Pipes in QuickXP are modeled after the UNIX pipe construct that is a special form of inter-process communication where one process writes into a pipe and another process reads from it. The reading process does not communicate with the writing process; it just waits for data to enter the pipe. Similarly, the writing process does not communicate with the reading process; it just waits for data to empty from the pipe before writing more data. Like UNIX pipes, QuickXP pipes only allow one writer. Unlike UNIX pipes, these pipes can have multiple readers.

QuickXP tracks when multiple modules issue read and write DMAs to a pipe. By abstracting this concept, programmers can easily build modules that interact with each other since QuickXP manages the bookkeeping overhead behind the scenes. Additionally, since most data is destined for VPs, the pipes move data close to the VP nodes and keep it there for as long as possible, maximizing locality and minimizing buffer copies.

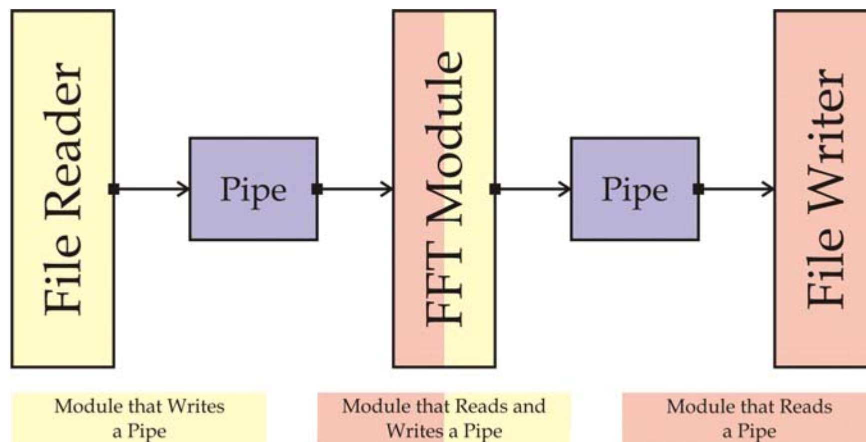

Figure 1 - Simple Application Using Pipes

Figure 1 above describes an application that reads a file, performs an FFT on the file's data, and then writes a new file with the processed data. In this application, pipes connect the processes, allowing each module to operate independently of the other. This is made possible by the pipe abstraction.

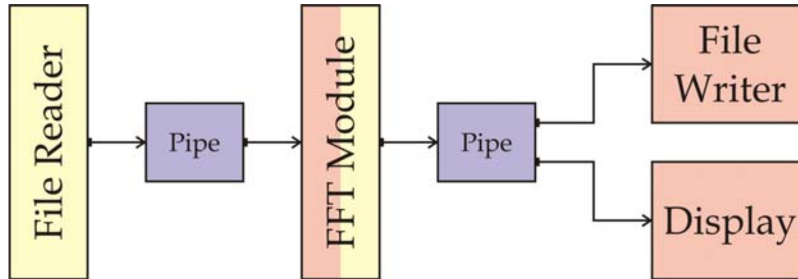

Figure 2 - Application that Uses Multiple Readers

In Figure 2 above, the application utilizes the multiple reader feature of QuickXP pipes to both write a file with the FFT results and display the data that is being written to the file.

1.4.2 Modules

Modules are a basic functional component of a QuickXP application. Modules perform the data processing and other work of an application, and pipes provide the glue that connects the modules with each other. Figure 1 and Figure 2 in the previous section show some basic examples of modules connected via pipes. Each block represents a separate module as part of a simple application.

A QuickXP module, by definition, must interface with one or more pipes. A module can have any number of input and output pipes. Modules that display or record pipe statistics may also only peek at pipes without reading or writing. While it is possible for a module to read and write the same pipe, this sort of behavior is discouraged. Modules are not restricted to a single operation. In fact, with regard to Vector Processors, it is usually more efficient to do as much work on the VP before writing data back into a pipe. The module writer needs to balance efficiency with functional abstraction.

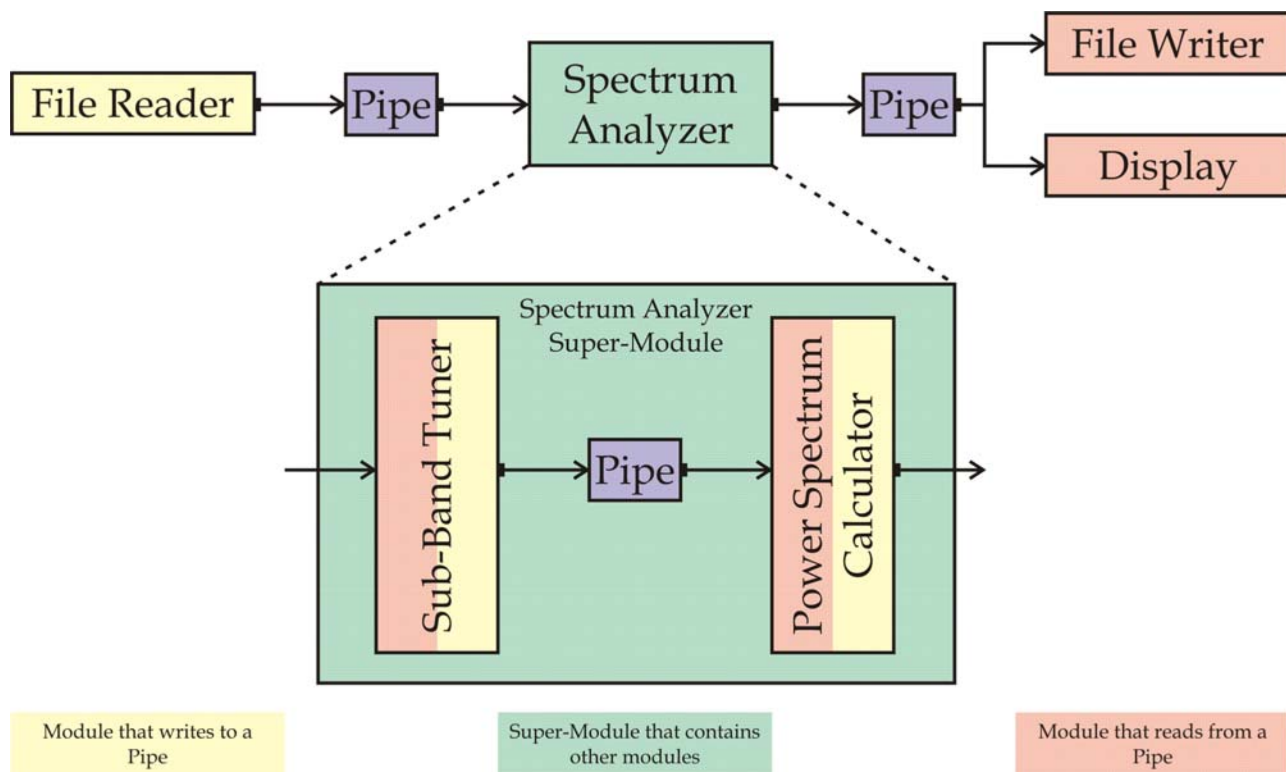

Figure 3 – QuickXP Super-Module Example

Modules are not restricted to a single task, either. Modules can contain nested modules and pipes. These modules are known as Super-Modules. An application is simply a Super-Module that can run as a standalone program. Figure 3 above illustrates a real-world application that could take advantage of a Super-Module. The *Sub-Band Tuner* module takes a signal, applies an oscillator, and decimates the data. The resulting data is passed to the *Power Spectrum Calculator* module, which converts the signal from time to frequency domain and performs averaging. Putting both of these modules into a Super-Module creates a Spectrum Analyzer module. Add input and output devices and you have a real-world application

1.4.3 Metadata

Put simply, metadata is information that describes data in a pipe. Where modules do the work and pipes act as the glue, metadata is the messenger that passes information from one module to another. Metadata is always associated with a grouping of data. When metadata is written into a pipe, all data written into the pipe from that point on is associated with that metadata until the metadata is written again. When that data is read from the other end of the pipe, the metadata is made available to the module.

Metadata transitions in pipes are considered important events in QuickXP, and have an affect on pipe reads. Reads from pipes cannot cross a metadata transition. If QuickXP detects a metadata transition within data being read, data is read up until the metadata transition. The rest of the read is truncated, a metadata change is reported, and the pipe read pointer now points at the first data element of the new metadata. Programmers must pay attention to the return values of pipe reads and be able to handle truncated reads, including reads that read no data. The later can happen when a metadata change is detected at the beginning of a read. QuickXP attempts to minimize this condition, but the asynchronous nature of pipes keeps QuickXP from detecting it in all cases. The following figure illustrates a pipe with metadata transitions and what happens when reads reach these data boundaries.

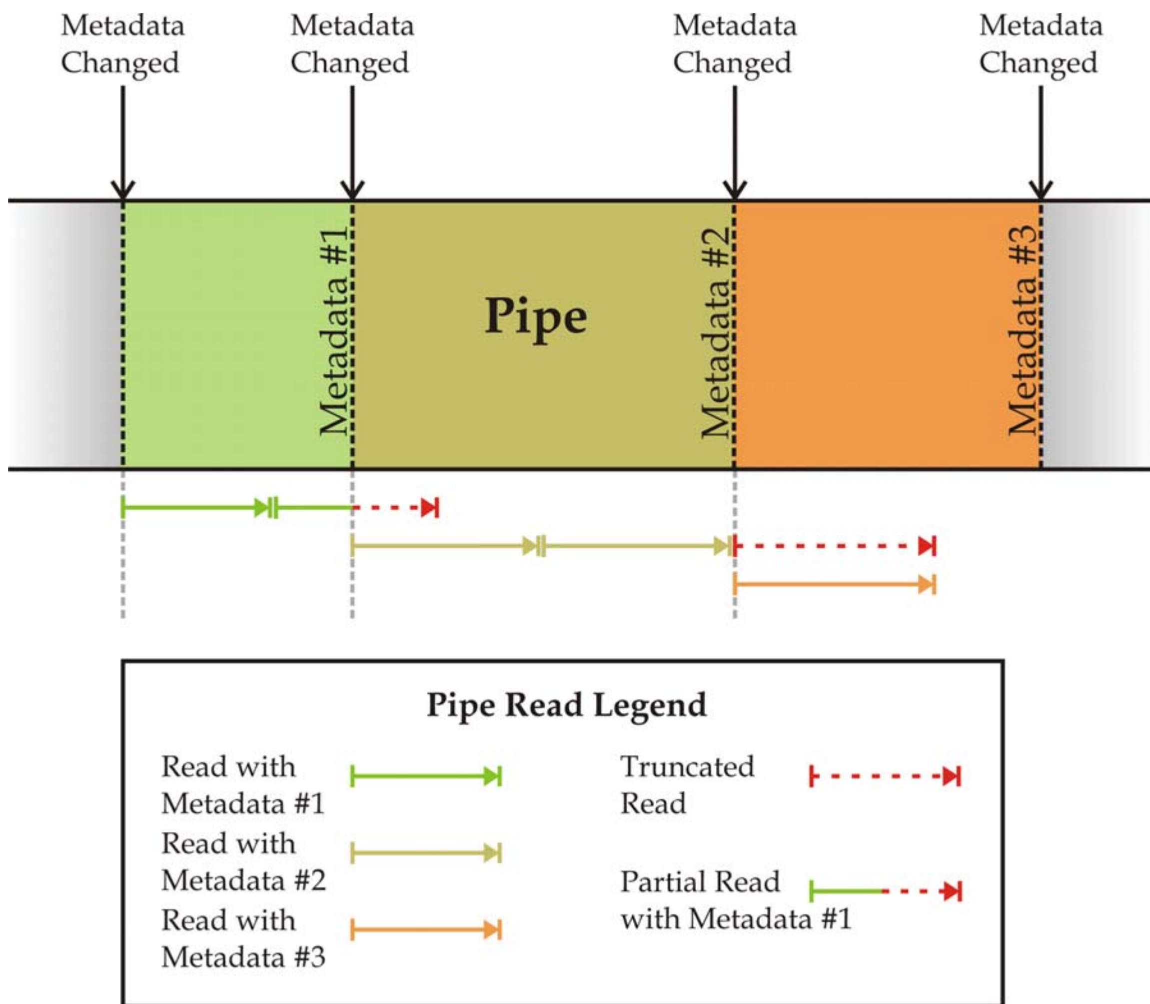

Figure 4 – Reading From A Pipe With Metadata Transitions

The contents of pipe metadata is mostly left up to the module and application programmer, but QuickXP provides a generic structure that should always be the first element of the metadata. This structure (**`tms_meta_t`**, which is described later in this manual) contains useful DSP information, flow control, and, most importantly, the data type used for determining the element size of data when performing pipe reads and writes. This is not required, nor is writing any metadata a requirement. However, most applications will benefit from having some metadata to describe the data.

1.5 Using QuickXP

Once the pipe mechanism is understood, using it to realize problem solutions is an extremely easy programming task. Libraries are provided in C and Python, based around the familiar open, read/write programming model used extensively as the data transfer backbone of applications over mediums as diverse as disk-based storage, shared memory, and network sockets. By using this model, QuickXP programmers will not have to learn new techniques for controlling hardware. Providing QuickXP functionality in C allows for the realization of maximum performance, while the Python layer provides an interactive environment for debug and rapid application development.

1.5.1 Using QuickXP in C

The C library, libxpq.a, makes up the heart of QuickXP. This library publishes an API that allows application programmers to create and use pipes. Combined with Texas Memory Systems' extensive and

easy-to-use math library, optimized DSP algorithms can be coded in a minimum number of lines of C that fully utilize TMS DSP performance.

The QuickXP C library is based on a familiar and well-established programming model used in all areas of software development: the open, read/write model. Pipes are created, opened, then read or written to just like any other system resource. Writing code that uses Pipes revolves around the following three groups of complementary functions:

1. Functions responsible for creating pipes: **`xpq_create`**/**`xpq_destroy`**
2. Functions responsible for opening pipes: **`xpq_open`**/**`xpq_close`**
3. Functions responsible for transferring data to and from pipes: **`xpq_read_to_host`** and **`xpq_read_to_vp`**/**`xpq_write_from_host`** and **`xpq_write_from_vp`**

You simply call **`xpq_create`** to create a pipe, **`xpq_open`** to open it for reading or writing, then one of the **`xpq_read_to_X`** functions to get data, and lastly, one of the **`xpq_write_from_X`** functions to push data. Two sets of transfer functions exist, since Pipes serve as the intermediary between two different processors: the host processor and a VP. **`xpq_read_to_host`**/**`xpq_write_from_host`** is used to transfer data for direct use by the host processor (displayed to the user, read from disk, etc…) while **`xpq_read_to_vp`**/**`xpq_write_from_vp`** transfers data to a VP for vector processing.

Using the pipe data-transfer abstraction and open/read/write simplicity, problems can be coded to their block-diagram form of decomposition. This makes the application easier to understand and maintain, and allows for a high degree of code reuse.

The only functions a programmer needs to become familiar with to fully utilize QuickXP are summarized below:

1. **`int xpq_create(char *pipename, char *device, char **pipedesc)`**
   (Creates a pipe with the name "pipename.")

2. **`int xpq_destory(char *pipename)`**
   (Removes the pipe named "pipename.")

3. **`int xpq_open(char *pipename, unsigned int flags)`**
   (Opens the named pipe for reading or writing, returning a handle.)

4. **`int xpq_close(int handle)`**
   (Closes the connection to the pipe described by handle.)

5. **`int xpq_write_from_host(int handle, char *hostbuf, int wc)`**
   (Writes data from memory referred to by hostbuf into the pipe specified by handle.)

6. **`int xpq_write_from_vp(int handle, vbuf_t vpbuf, int wc)`**
   (Writes data from VP-memory referred to by vpbuf into pipe specified by handle.)

7. **`int xpq_read_to_host(int handle, char *hostbuf, int wc, int adv_wc, int *wc_read)`**
   (Reads data from pipe specified by handle into memory referred to by hostbuf.)

8. **`int xpq_read_to_vp(int handle, vbuf_t *vpbuf, int wc, int adv_wc, int *wc_read)`**
   (Reads data from pipe specified by handle into VP-memory referred to by vpbuf.)

1.5.2 Using QuickXP from the Python Scripting Layer

The scripting layer supported by QuickXP is based on the Python programming language. All of the functionality of the C library is also available in this scripting layer, using the same function names and calling sequences. This allows programmers to quickly prototype applications, interactively test ideas, and then easily convert the final result to C for maximum performance.

In addition to the QuickXP API described above, QuickXP modules are also available to the Python layer as independent, self-contained sub-processes. Thus, pipes can be dynamically created/destroyed and modules can be started and stopped interactively all at the Python shell prompt. New applications can be quickly and interactively pieced together and debugged without ever having to modify previously written modules. This allows for extremely rapid application development. As with the rest of the QuickXP script API, once the programmer no longer needs the simplicity or flexibility of the scripting layer, the application can be easily converted to C for maximum performance, though script based solutions are robust and fast enough to be used for permanent end-user applications and super-modules. Using the included wxPython extension, this scripting layer can also provide useful GUI tools for existing C modules.

1.6 LIBXP

The ability to take advantage of DSP acceleration hardware is important to the QuickXP programmer. This is done via the LIBXP library. The API to LIBXP described in the *LIBXP User Guide* remains unchanged in regard to QuickXP. Application programmers are expected to manage VP resources at the C layer. Module writing convention will dictate that modules should be written to consume multiple available VPs – thus resulting modules should abstract VP node management from the script user, just not from the module writer.

QuickXP DMAs data into VP nodes using the LIBXP **`vcopy()`** routine. This allows the programmer to predict VP node stalls and DMA overlaps – important operations when optimizing DSP routines, but not required in most applications.

Chapter 2 - Installation

QuickXP is a simple RPM install on supported Linux systems. At this time, QuickXP is supported on Red Hat Enterprise 4 using 32-bit x86 processors or 64-bit Intel or AMD processors.

Prerequisites

QuickXP expects the *libxp* RPM and the *xp-package* RPM to be installed prior to use. To install QuickXP download the latest package for your platform from the Texas Memory Systems website – along with any other required packages.

QuickXP also requires *Python 2.5* or greater and *wxPython 2.8* or greater. These two products are not included with the Red Hat Enterprise Linux distribution. Texas Memory Systems has provided a separate *quickxp-melba* package that will build and install these applications with the main QuickXP package. *Python* and *wxPython* can also be obtained from their respective providers and installed into the standard Linux directory structure.

Installing the QuickXP Package

Install the QuickXP package as root (this example uses Red Hat Enterprise Linux 4 running on AMD 64-bit processors):

```
# rpm -ih ./quickxp-1.0.0-RHELWS4NU3.2.6.9_34.Elsmp.x86_64.rpm
Preparing... ########################################### [100%]
1:quickxp    ########################################### [100%]
```

The package installs software into */opt/texmemsys/quickxp* by default. The QuickXP C header files, including *xpq.h*, *tms_meta.h* and *xpqmodules.h*, are located in the */opt/texmemsys/quickxp/include* directory. The QuickXP C libraries, consisting of the QuickXP core library *libxpq* and the package of TMS supplied Modules *libxpqmodules*, are found as both static and shared-libraries in */opt/texmemsys/quickxp/lib*. All of the Python extension modules, including **xpq**, **xpq.modules** and all of the GUI elements are stored in */opt/texmemsys/quickxp/python*. The directory */opt/texmemsys/quickxp/modules* includes **modules.make** and other utilities used to create user defined packages of C based Modules. The final three directories, */opt/texmemsys/quickxp/docs*, */opt/texmemsys/quickxp/examples* and */opt/texmemsys/quickxp/templates* contain documentation, example source code and usable source-code templates, respectively.

The following environment variables must be set prior to using QuickXP:

- **XPQ** – Set to the base directory of the QuickXP installation:
  /opt/texmemsys/quickxp
  This should be already setup by the quickxp RPM.
- **MATHXP** – Set to the base directory of the LIBXP installation:
  /opt/texmemsys/mathxp
  This should be already setup by the mathxp RPM.
- **PYTHONPATH** – Must include the QuickXP installation Python directory:
  $XPQ/python:$PYTHONPATH
  This should be already setup by the quickxp RPM.

Installing Python and wxPython with Melba

NOTE: If suitable versions of *Python* and *wxPython* are already installed into standard locations on the Linux system, then this package is not required for QuickXP.

The quick*xp-melba* package will build and install *Python* and *wxPython* into the standard installation location (*/opt/texmemsys/quickxp-melba*). Copy the compressed package into a temporary directory and run the following commands as root to install it:

```
# tar xvzf ./quickxp_melba-1.0.0.tar.gz
<… List of Files Extracted …>
# cd quickxp-melba
# ./install.sh
Would you like to install the TMS support version of
Python and wxPython on this local system under /opt?
 ([yes]/no)? yes
make: Entering directory `/tmp/quickxp-melba'

***** Starting 'native' build at Thu Sep  6 17:36:34 CDT 2007 *****
***** Installing into '/opt/texmemsys/quickxp-melba' *****

***********************************************************************
** Building 'Python'
***********************************************************************
** Uncompressing (Python-2.5)
** Configuring
** Building
** Installing

***********************************************************************
** Building 'wxPython-src'
***********************************************************************
** Uncompressing (wxPython-src-2.8.4.0)
Applying patch '/tmp/quickxp-melba/package/wxPython-src/patches-
2.8.4.0/wxPython-2.8.4.0_001.patch'
patching file configure
** Configuring wxWidgets
** Building wxWidgets
** Installing wxWidgets
** Building and installing wxPython

real    16m50.054s
user    14m55.704s
sys     1m57.387s

'native' was successfully built

***** Finished 'native' build at Thu Sep  6 17:53:24 CDT 2007 *****
Cleaning up build directories...done.
make: Leaving directory `/tmp/quickxp-melba'
No errors detected
```

Because *quickxp-melba* does not install the applications into standard Linux locations, the following environment variables must be modified so that QuickXP can find and use the appropriate versions of *Python* and *wxPython*:

- **PATH** – Must add the correct version of *Python* to the executable path:
  /opt/texmemsys/quickxp-melba/bin:$PATH
  This will be automatically set for new sessions by */etc/profile.d/quickxp.sh* & *.csh*.
- **LD_LIBRARY_PATH** – Must put *wxPython* shared objects in the library path:
  /opt/texmemsys/quickxp-melba/lib:$LD_LIBRARY_PATH
  This will be automatically set for new sessions by */etc/profile.d/quickxp.sh* & *.csh*.

Chapter 3 - Programming Guide

Programming with QuickXP can be divided into two distinct aspects, module development and super-module/application development. Modules are independent, atomic processing units, designed to receive (or generate) data, manipulate it in some fashion, and pass the result along. On their own, they are not particularly useful, but they form distinct pieces of the final puzzle, the application. Given a collection of modules, solving a particular DSP task becomes a matter of stringing instantiations of the desired modules together, connected by pipes. Modules can be reused in different applications, significantly decreasing development time of complicated new algorithms. As a concrete example, assume you wanted to sum the results of the vectors contained in two data files and plot the result. Modules already exist to read data from a file into a pipe, sum the data from two pipes into another pipe, and to plot data read from a pipe. Creating this application would be a simple matter of using the QuickXP scripting interface to instantiate two file-reader modules, one vector addition module, and one plotter module, then connect each module with a Pipe. If all of the Module building blocks necessary to realize an application are already available, no new module code ever need be created. Yet new modules can be coded as needed for specific functionality, adding to the user's suite of DSP tools. The considerations and recommended technique for these two distinct aspects of programming in QuickXP will be examined in the following subsections.

3.1 Module Programming

Modules can be written in either C or Python. An equivalent API is provided for both (as detailed below in the various API chapters). Code in this chapter will be given in C. They map to Python, with the appropriate syntactic changes to account for the different languages, and the QuickXP API function names mapped as described in the Scripting API chapter. The techniques and considerations are primarily the same for both. Any differences will be highlighted in the respective subsections when appropriate.

The user implements a module as a function in C, or as a function or a class in Python. This function (or class) is then declared as a module, a process that associates a user-defined name with that function (or class). The resulting module can be instantiated by calling the function created by the declaration process. Modules created in C are accessible from C or Python super-modules, but currently, modules created in Python are only accessible from Python super-modules.

For the sake of brevity, error checking of all API functions' return values has been omitted from the following examples.

3.1.1 Implementing Modules

The user implements a module as a function in C, or as a function or a class in Python. C function implementations take an array of pointers to strings as input and return an integer status (0 on success, less than 0 on error), analogous to the ubiquitous "main" function. Python function implementations take standard Python arguments, as defined by the user. Python class implementations must be a subclass of **`xpq.ClassModule`**, with a method named **`main`** which will act as the Module function.

Module instances share the same global memory space (they are implemented as asynchronously running operating system threads executing the code specified by the module declaration) so the appropriate multi-threaded programming care must be taken if accessing any possibly shared data. This is of primary concern when using C function implementations. It is highly recommended that any "global" variables required by a module C implementation be stored in a single block of memory unique to each module instance. A C struct declared on the stack of the module's associated main function is perfectly suited for such a task. Given Python's inherent multi-threaded support, this is generally not a concern but should be taken into account, as some python extension modules are not thread-safe (most GUI packages, for example). Note that with Python class modules, the class instance will be created in the calling context, but the 'main' method will be run in the new module instance's context.

Regardless of the implementation chosen, the Module function should be designed as an atomic processing unit, meaning that it is only concerned with reading the data necessary for it to accomplish its designated functionality then write its results for the next module in the chain to work with. A module should never issue a read on the same pipe that it has itself written data into (for example, **`xpq_write_from_host`** to "pipeA" then **`xpq_read_to_vp`** from "pipeA".)

The following is the minimal Module example. Specific points of user customization are designated by **`/*** description ***/`** comment sections.

```
int minimal_module_template(int argc, char ** argv)
{
    rp = xpq_open(..., XPQ_READ);
    xpq_module_cleanup_push(xpq_close, rp);
    wp = xpq_open(..., XPQ_WRITE);
    xpq_module_cleanup_push(xpq_close, wp);
    while (xpq_active()) {
        xpq_read_to_X(rp, BUF, xfer_len, xfer_len, &read);

        /*** Process Data ***/

        xpq_write_from_X(wp, BUF, xfer_len);
    }
    return 0;
}
```

This example can be used to quickly accomplish a given task, and is all that is absolutely necessary to start processing data with QuickXP. However, it lacks the features that make it more generically useful, such as metadata, Control and dynamic flow-rate. It is easily extended to include these capabilities, as is shown with the next example.

In general, most modules will adhere to the following template. Modules may omit the read or write elements of this template, depending on whether or not the source or destination for their data is a pipe. A fully functional C-code version of this template is included in the QuickXP distribution. Specific points of user customization are designated by **`/*** description ***/`** comment sections.

```
int module_template(int argc, char ** argv)
{
    /*** Process input arguments specified in argc/argv ***/

    /* Open all pipes (both readers and writers) */

    rp = xpq_open(..., XPQ_READ);
    xpq_module_cleanup_push(xpq_close, rp);
    wp = xpq_open(..., XPQ_WRITE);
    xpq_module_cleanup_push(xpq_close, wp);

    while (xpq_active()) {

        /* Propagate metadata from the input pipe to our outputs */

        xpq_meta_read(rp, &prmeta, &rmetasz, 0);
        xpq_module_cleanup_push(xpq_meta_free, prmeta);

        /*** Determine output metadata ***/

        xpq_meta_write(wp, &wmeta, sizeof(wmeta), 0);
```

```
        /* Data processing loop */
        xpq_control_init(ctrl);
        while (xpq_active() && !ctrl->update) {

            st = xpq_read_to_X(rp, BUF, xfer_len, xfer_len, &read);
            if (st == XPQ_ST_METADATA_CHANGED) {
                /* Our metadata changed.  We've actually received
                   'read' elements (which will be between 0 and
                   xfer_len, inclusive).  This data can be processed,
                   thrown away, or whatever is appropriate for the
                   module... */
                ctrl->update = 1;
            }

            /*** Process Data ***/

            xpq_write_from_X(wp, BUF, xfer_len);

            /* Defer to xpq_control_process to make sure our data is
               output at the rate requested by the user, but to
               also ensure we stay responsive to input control. These
               two steps are combined to keep the module responsive to
               control, yet have minimum impact on the processor when idle.*/
            if((st = xpq_control_process(process_param, &tp, ctrl,
                                    prmeta->datarate, readlen)) < 0) {
                 xpq_log_error("failed processing control");
                 return st;
            }        }

        /* Free Metadata */
        xpq_modules_cleanup_pop(1);
    }

    return 0;
}
```

As always, the first step in any module is to process its input arguments. It is suggested that the same format be used for the input arguments as is supported by the module's control, but this is not a requirement. This step establishes the initial values for any data used by this instance of the module. The names of the pipes used by this instance are generally passed as input parameters.

After performing initial setup (including opening all required pipes), the primary module loop is entered. This template assumes that the module will process data ad infinitum. This need not be the case. Specific control could be designated to terminate processing. The "endless" loop could be replaced with code to operate on a specific amount of data, or for a specific amount of time, or whatever else the task at hand may require. This is simply an example of a more common usage.

The first step of data processing, metadata propagation, however, is essential. While not strictly required, this allows modules connected in a processing chain to establish characterizing information about the data being exchanged. Their actual processing can then be coded more generically, adapting to the data provided, giving them more interoperability and a higher degree of code-reuse. Modules reading from pipes should first read the metadata of each input pipe (this step will obviously be skipped if the module is a pure data source). This input metadata, along with the algorithm being implemented and any current control/configuration settings should be used to determine then write the metadata describing data generated to the various output pipes (again, this step will be skipped if the module does not write to any pipes and does not care to persist this information).

Once the metadata has been propagated to the output pipes, the data processing loop is entered. The primary goal of this loop is to retrieve input data, operate on it, and then propagate the results. It is expected to operate as fast as possible (or desired based on requested output data rates). The primary processing loop should only be exited if it is determined that significant changes have occurred such that basic assumptions should be reevaluated and down-stream modules notified of any changes. This generally occurs when up-stream metadata changes, with certain incoming control, or some situation uncovered by the processing itself. Up-stream metadata changes are indicated when data is read from a pipe (via any of the **`xpq_read_to_X`** routines) by a return value equal to XPQ_ST_METADATA_CHANGED. In this situation, it is possible that less data than requested was actually transferred. The actual amount of data available after the read is stored in the memory referenced by the 'wc_read' parameter to the read function. It is up to the module designer to decide what to do with this data.

After the current "chunk" of data has been processed, the module should check for any control, and possibly defer further processing for a variable amount of time to maintain the requested output data rate. These two steps are combined to keep the module responsive to incoming control, yet have minimal impact on processor utilization when idle. The API function **`xpq_control_process`** has been designed to cooperate for this purpose. **`xpq_control_process`** combines **`xpq_control_flow`** and **`xpq_control_receive`** that allows for efficient and complete parsing of incoming commands. **`xpq_control_flow`** cleanly defers processing until the designated time slice has expired or control has been received. If control has potentially been received, **`xpq_control_receive`** is called successively to empty the queue of pending messages. If **`xpq_control_flow`** terminated due to the receipt of a control message, the pattern is repeated until the time slice has expired and the next "chunk" of data should be processed.

Finally, when it has been determined that the module instance should terminate (either by the instance itself or the module sub-system), the module instance must release any resources it may have previously acquired (including closing pipes, closing any open file-descriptors, freeing dynamically allocated memory, releasing any resource locks, etc…) and return from its "main" function.

3.1.2 Metadata

Since data stored in a pipe is accessed as linear chunks of bytes, important information about more complex data structures, such as vectors, matrices or even simple scalars, is lost. To reconstruct a simple scalar read from a pipe, the module must know how many bytes comprise that scalar, and what type of data those bytes represent. To reconstruct a vector of scalars, the module must not only know this information about the elements of the vector, but how many elements are in the vector. To reconstruct a matrix, the module must know how many rows and how many columns are in the matrix. Some algorithms fundamentally modify the data stream by producing different amounts of data than they consume (such as resamplers, decimators, etc…). In these situations, it would be nice to know how output elements correlate to the inputs, since a simple one-to-one correspondence can no longer be assumed.

TMS DSP hardware and software are designed for vector processing. The fundamental type of data that will be operated upon, and thus read or written to or from pipes, will be vectors of numerical data. Certain information is required to describe such data. The TMS metadata structure contains elements that are used to describe these fundamental aspects of that data, as well as elements useful for describing a particular data element's position in the processing stream.

All modules should read and interpret incoming metadata to provide guidance on how the data they receive should be interpreted. All modules should also construct and write metadata to describe the data they are providing for downstream modules.

TMS metadata structure

All metadata written to Pipes should either contain a **`tms_meta_t`** structure starting at byte 0 of the data written (from C) or be a subclass of **`xpq.MetaData`** (from Python). All of the pre-existing Modules provided

by TMS expect this information in the metadata. Pipes opened with the **XPQ_WC_ELEMENTS** flag require this information in the metadata. Note that the **tms_meta_t** structure must always be initialized before use via one of the API functions, such as **tms_meta_init** or **tms_meta_init_from_file**.

The **dimensions** field is an integer indicating the number of dimensions of the data. A value of one indicates that the data is considered a never-ending vector of numerical data. This is analogous, for example, to a digitally sampled signal varying in time. Successive elements will simply be the next elements sampled. A value of two indicates that data is considered to be a never-ending succession of frames of fixed size data, like a matrix with an infinite number of rows (or columns, depending on how you look at it). This is analogous, for example, to the successive views of the frequency spectrum of a digitally sampled band-limited time-variant signal as it varies over time. Currently, TMS metadata only supports **dimensions** up to and including two. This may be addressed in future versions.

Since data in a pipe is logically unbounded, the outermost dimension could also be considered unbounded. Data will continue to be available as long as the writer to the pipe continues to provide it. Successive data will represent new data along that axis. As such, the outermost dimension of data processed with QuickXP is most often time, though this is not a requirement.

The **datatype** field encodes the type of a single element in the data stream, and is analogous to the fundamental data types in C. Valid values are any of the typecodes defined in LIBXP. Please refer to that manual for a complete list. The data type defines the number of bytes a single data element spans, whether it is integer or floating-point, real or complex, signed or unsigned, and various other factors. This field is used by QuickXP when issuing reads or writes on pipes opened with the **XPQ_WC_ELEMENTS** flag. The total number of bytes transferred for those calls will depend on the **datatype** field of the metadata valid for that transfer.

The **framesize** field is an integer indicating the number of elements (as defined by **datatype**) that comprise a single frame of the outermost dimension. This number will be one if **dimensions** is equal to one and greater than or equal to one if **dimensions** is equal to two.

The **xstart**, **xdelta** and **xunits** fields describe the first dimension of the data and how successive elements correlate to points in that dimension. The **xunits** field determines the unit of measurement for that dimension. Currently supported units are time (**TMS_META_UNIT_TIME**) and frequency (**TMS_META_UNIT_FREQUENCY**). Time is measured in seconds, and frequency in hertz. The **xdelta** field specifies how far apart (in **xunits**) successive elements are, and the **xstart** defines the value (in **xunits**) corresponding to the first element. So, for example, the following values:

```
xunits = TMS_META_UNIT_TIME
xstart = 3.2
xdelta = 0.0005
```

describe the data as follows:

Element Index	Time Associated with Element
0	3.2000
1	3.2005
2	3.2010

and so on…

This could correspond to a signal digitally sampled at 2kHz, starting at time 3.2s (from some arbitrary time zero).

The **ystart**, **ydelta** and **yunits** fields are identical to their matching **'x'** counterparts explained above, except they describe the second dimension of the data, and how successive frames of data elements correlate to points in that dimension. They are only valid if **dimensions** is greater than or equal to two. So, for example, the following values:

```
framesize = 100
yunits = TMS_META_UNIT_TIME
ystart = 3.2
ydelta = 0.0005
```

describe the data as follows:

Frame Index	1st Element Index	Time Associated with Frame
0	0	3.2000
1	100	3.2005
2	200	3.2010

and so on…

The **starttime** and **endtime** fields are primarily meaningful for looping data generation and playback modules. Given a fixed size set of data, these two fields in conjunction represent the range of that fixed size set of data being looped upon, in the units of the outermost dimension (as specified by the combination of **dimensions**, **xunits** and **yunits**). The fact that the two fields' names end in "time" does not imply that the outer units must be time. As previously stated, they often are, but this is not a requirement. So, for example, if the file playback module is being used to play a file representing samples of time domain data, and the user has selected to restrict the playback of data from that file to the range 1.0s to 5.2s, the **starttime** and **endtime** fields will be 1.0 and 5.2, respectively. These fields should only be set by the originator of the data they describe. Processing modules will simply pass this information to the next module in the chain. Modules that interact with the user can use these fields to correlate the data they are displaying with the time (or other appropriate unit) to which that data corresponds in the originating data stream.

The **datarate** field is used to instruct a module how fast (in real time) it should process data, and to communicate to downstream modules how fast data is being pushed their way. The data rate is represented in samples per second, with a rate of 0 representing no flow control. It is used in conjunction with the **xpq_control_flow** API function to determine how long a module should suspend processing to maintain the desired rate, and by the **xpq_elements_per_time** API function to determine how much data should be processed by a module in one iteration of its processing loop. Since QuickXP automatically suspends a module while it waits for data from its source pipes, only data generators (whose source data does not come from a pipe) should use the **xpq_control_flow** API function. Modules should set the **datarate** of the metadata for all of their output pipes to represent the rate at which they are writing data into that pipe.

Providing user-defined metadata

Module developers may use the metadata mechanism to provide any extra arbitrary information describing the data in a pipe that they wish. Note that a metadata transition on a pipe will generally cause a reading module to break out of its inner processing loop, and depending on the module design, potentially drop some fixed amount of data. As such, metadata transitions are intended to be low volume traffic.

Metadata is assumed to be a block of arbitrarily sized binary data. The first N bytes of that data must be the predefined TMS metadata structure (where N is the size of that structure, in bytes). Anything else is simply passed through the QuickXP subsystem.

User defined modules may append extra bytes, or modify the elements that they are responsible for (either in the TMS metadata portion, or in any extended portion they may be aware of). They should propagate untouched any extra, unknown metadata bytes they receive to downstream modules.

The recommended way to add extra metadata is to encapsulate the extra information in a C structure whose first element is the **`tms_meta_t`** structure. This guarantees that the required core elements will be accessible by the QuickXP subsystem and other modules. Note that the **`tms_meta_t`** structure must always be initialized before use via one of the API functions, such as **`tms_meta_init`** or **`tms_meta_init_from_file`**. Future versions of QuickXP may provide API functions to automate a more stack-like facility for metadata extensions.

3.1.3 Control

QuickXP provides a control mechanism where module instances may communicate asynchronously with each other. QuickXP control channels are unconnected, packet oriented channels. Each module instance has its own independent receive queue. Control messages are simple fire-and-forget name and value pair datagrams (or simply name only), both elements being arbitrary length, NULL terminated strings. No restrictions or assumptions are placed on the content of either of these elements by the QuickXP subsystem; both are up to the discretion of the module implementer. The value element of the pair may be omitted, in which case the receiving module will be given a NULL pointer for that element.

QuickXP control is designated for a particular module instance using that instance's fully qualified name. When control is received, the fully qualified name of the sender is also returned. This fully qualified name is a NULL terminated string that uniquely identifies any given module instance amongst all running module instances. The fully qualified name is a hierarchical name generated automatically when the module instance is created. It is the fully qualified name of the module that instantiated the one in question (the toplevel application is treated as an implicit module instantiation given the instance name "main") and the module in question's instance name, separated by the XPQ_MODULE_PATH_SEPERATOR (a single dot; **`'.'`**). The user may set the instance name of a module when it is created with the **`"NAME"`** input argument to the module instantiation function, described in further detail in the next section "Super-Module and Application Programming". In the absence of an explicitly set instance name, the QuickXP subsystem automatically generates one based on the module name being instanced and the order of module instances created. So for example, if a single module instance is created by the application and given the name "MyMod1", its fully qualified name will be "main.MyMod1". If that module instance itself creates two other module instances named "FirstSubMod" and "SecondSubMod", those instances' fully qualified names will be "main.MyMod1.FirstSubMod" and "main.MyMod1.SecondSubMod", respectively. Using these fully qualified names, each of these four module instances (remember that the application is an implicit module instance) could be targeted for control.

3.1.4 Declaring Modules

After a module's main function (or class) has been implemented, it must be declared as a module. This process associates a user-defined **`NAME`** with that function (or class). The resulting module can be

instantiated by calling the function created by the declaration process. In C, module instantiation functions will be callable as **`xpq_modules_NAME(…)`**. In Python they will be callable as **`xpq.modules.NAME(…)`**.

Declaring a C Module

To declare a function in C as a module, use the XPQ_MODULE macro. For example:

```
int my_module_function(int argc, char **argv) {
    printf("running my module\n");
}

XPQ_MODULE(MyModule, my_module_function, "An example module.")
```

This will create the module **`MyModule`**, which will run the C function **`my_module_function`** when instantiated. To instantiate this module from C, call the function **`xpq_modules_MyModule`** with the desired initial arguments passed as parameters. To instantiate this module from Python, call the function **`xpq.modules.MyModule`** with the desired initial arguments passed as parameters.

Declaring a Python function Module

To declare a function in Python as a module, use the **`xpq.FunctionModule`** decorator. For example:

```
import xpq
@xpq.FunctionModule
def MyModule(*args, **kwds):
    print "running my module"
```

This will create the module **`MyModule`**, which will run the Python function **`MyModule`** when instantiated.

Declaring a Python class Module

To declare a class in Python as a module, use the **`xpq.ClassModule`** as a base class for the Python class, and implement a **`main`** method in that class with the desired module functionality. For example:

```
import xpq
class MyModule(xpq.ClassModule):
    def main(self, *args, **kwds):
        print "running my module"
```

This will create the module **`MyModule`**, which will run the Python class method **`MyModule.main`** when instantiated.

3.1.5 Building Modules

This section only pertains to modules implemented in C. The Python interpreter automatically compiles Python code; therefore no specific action is required of the user to be able to instantiate modules written in that language.

Modules implemented in C are compiled and linked into a shared library (called a Module Package, or simply package). Multiple modules may exist inside a single package, and multiple packages may be used by an application. Packages provide a convenient mechanism to manipulate an arbitrary collection of modules as a single entity.

A GNU Make compatible makefile has been provided with the distribution that should be included in the user's makefile. It automates the process of compiling and linking source code to create the final shared library, and other internal bookkeeping that allows the modules defined in the package to be exported to the Python scripting layer. This file is named 'modules.make' and is located in the 'modules' directory of the QuickXP installation. An example user defined makefile is outlined and explained below. Concrete examples are also provided with the distribution in the 'examples' directory of the QuickXP installation.

```
# This assumes the default installation location…

XPQ                     = /opt/texmemsys/quickxp
XPQ_MODULES_PKGNAME     = mymodules
XPQ_MODULES_SOURCES    += mymodules1.c mymodules2.c

.PHONY: all
all: xpq_modules

.PHONY: clean
clean: xpq_modules_clean

include $(XPQ)/modules/modules.make
```

Executing make at the command line with this makefile would produce the **mymodules** package from the two C source files **mymodules1.c** and **mymodules2.c**. This package would consist of a header file **mymodules.h** and a shared library **libmymodules.so**. The header file would contain the function prototypes for all of the module instantiation functions declared in the C source files, and the shared library would contain the actual module instantiation functions.

The **XPQ** variable must be set to the path of the QuickXP installation. Often, the user will define this as an environment variable given its static nature, and as such it may be omitted from the makefile.

The **XPQ_MODULES_PKGNAME** variable must be set to the desired package name. This name will be used to uniquely identify the package. It is used to generate the names for the header file and shared library file, as well as other internal elements necessary to allow the modules defined in the package to be exported to the Python scripting layer.

The **XPQ_MODULES_SOURCES** variable must be set to the list of C source files that contain the user code. These files will be compiled and linked into the final shared library. All Modules declared in all of the files specified will be exported from the final package.

3.2 Super-Module and Application Programming

The application is the conductor of the orchestra of modules operating to produce a symphony of data processing. It creates the pipes that connect the modules it instantiates. After everything has been created, it may intermittently communicate with the running modules (via QuickXP control) to either receive information about or directly manipulate their state.

QuickXP Super-Modules and applications can be written in either C or Python. The API for instantiating modules in either is symmetric. However, modules implemented in Python are currently only accessible from Super-Modules and applications written in Python. As of this release, that includes all of the GUI elements currently provided by TMS.

After a Super-Module or application has created all of the pipes and module instances that it must, it may simply return. The module runtime system will prevent a Super-Module or application from completely terminating until all modules it has instantiated terminate.

3.2.1 Super-Module and Application Programming in C

Pipes are created and control sent or received with the API functions documented in the C API chapters below. Modules are instantiated by calling their module instantiation functions as mentioned in the previous section "Declaring Modules". From C, the module instantiation functions take a variable number of arguments, each a pointer to a NULL terminated string. The list of arguments must end with a NULL. These arguments are used as the **argv** parameter to the module implementation function. The special argument **"NAME"** is recognized by the Module runtime system. It must be followed by another parameter that will be used as the module instance name for the instance created.

```
xpq_modules_MyModule("NAME", "MyMod", "PARAM1", "VALUE1", NULL);
```

This will create an instance of the module MyModule, assigning it the module instance name **"MyMod"**, calling the associated module implementation function with three entries in the **argv** array; **"MyMod", "PARAM1", "VALUE1"**.

The following is a basic C application template. The source for this example is included with the distribution.

```
#include "xpq.h"
#include "xpqmodules.h"
/*** include custom Module Package header files ***/

#define SYSTEM "/dev/xp/0"

int XPQ_C_application_template(int argc, char ** argv)
{
    char * pipename1;

    xpq_create(NULL, SYSTEM, &pipename1);

    xpq_module_MODULE("NAME", "MODULE1", pipename1, NULL);

    /*** Either implement custom user-interface to control code
         or simply fall off the end, relying on the module runtime
         system to correctly synchronize module termination. ***/

    return 0;
}
```

3.2.2 Super-Module and Application Programming in Python

Pipes are created and control sent or received with the API functions documented in the Python API chapters below. Modules are instantiated by calling their module instantiation functions as mentioned in the previous section "Declaring Modules". From Python, the module instantiation functions may take both a variable number of arguments and arbitrary keyword arguments. The parameters are passed as-is to modules implemented in Python. For modules implemented in C, the positional arguments are stored in the **argv** module instance parameter first, one array entry per argument, followed by the keyword arguments. Each keyword argument corresponds to two entries in the **argv** array, one for the keyword and one for its value. Currently, both the variable arguments and keyword argument values sent to C module implementations must be Python strings. Future versions of QuickXP may attempt to automatically convert these values to strings. Note that due to Python implementation details, the order of keyword parameters when passed to C module implementations is non-deterministic. The special argument "NAME" (either positional or keyword) is recognized by the module runtime system. It must be followed by

another parameter if positional or given a value if keyword that will be used as the module instance name for the instance created.

```
xpq.modules.MyModule("NAME", "MyMod", "PARAM1", "VALUE1")
```

This will create an instance of the module MyModule, assigning it the module instance name **`"MyMod"`**, calling the associated module implementation function with three entries in the **`argv`** array; **`"MyMod", "PARAM1", "VALUE1"`**.

The following is a basic Python application template. The source for this example is included with the distribution.

```
import xpq
import xpq.modules
### import custom Module Packages ###

SYSTEM = "/dev/xp/0"

if __name__ == '__main__':
    pipename1 = xpq_create(None, SYSTEM)

    xpq.module.MODULE(pipename1, NAME="MODULE1")

    ### Either implement custom user-interface to control code
    ### or simply fall off the end, relying on the module runtime
    ### system to correctly synchronize module termination.

    # The following is required for any wxPython GUIs
    xpq.modules.control.main.run()
```

Importing C based Module Packages

User defined module packages implemented in C can be imported into the Python scripting layer either explicitly using the **`xpq.modules.LoadQuickXPCModules`** function or implicitly using the **`XPQ_MODULES_PKGS`** environment variable. To explicitly load a module package, call the **`xpq.modules.LoadQuickXPCModules`** function with a single argument, the filename of the shared library implementing the package. This will load the package and export all modules declared within. Packages can be implicitly loaded at Python startup by setting the **`XPQ_MODULES_PKGS`** environment variable to a colon-separated list of the shared library filenames implementing the packages. This behaves exactly as if **`xpq.modules.LoadQuickXPCModules`** were called on every filename in the list. The module instantiation functions defined in the shared library files will be available to the Python scripting layer as **`xpq.modules.NAME`** where **`NAME`** is the module name as declared in the C source files.

QuickXP Graphical User Interface

QuickXP includes various graphical user interface components to provide a variety of information and control to the user. The user may also implement custom GUI interfaces to interact with pre-existing modules or custom modules they define. The pre-existing interfaces will be described in more detail in "Chapter 7 - Pre Existing Modules." This section will focus on defining custom interfaces and instantiating interfaces.

NOTE: Applications must currently call the **`xpq.modules.control.main.run()`** function after instantiating all modules and module GUI's. This explicit step may be alleviated in future versions of QuickXP.

The GUI in QuickXP is based on the WX cross-platform GUI library, utilizing the wxPython extension package for accessibility from the scripting layer. Custom interfaces are created at the Python layer by subclassing an appropriate WX widget, and building from there. Documenting WX and wxPython are beyond the scope of this document. Please refer to their respective manuals and examples for more information. All that is needed to interact with the QuickXP engine is to assign primary interface widget classes' **`__metaclass__`** variable to **`xpq.modules.control.main.QuickXPGuiMetaClass`**. These classes may also define a **`GetCaption`** method that returns the desired title as a string for that widget instance. The widget may also register to receive the Control generated by specific Modules with the **`RegisterForQuickXPControl`** function of the **`wx`** application object. It is recommended that interface widgets designed to compliment a particular module accept the same input arguments to its __init__ function that the module accepts as its input arguments. Future versions of QuickXP may automate the instantiation of a module's GUI when the module itself is instantiated, passing the same arguments to both. The following is an example widget.

```
import xpq
import xpq.modules.control.main
import wx

class MyModuleControlPanel(wx.Panel):

    __metaclass__ = xpq.modules.control.main.QuickXPGuiMetaClass

    def GetCaption(self):
        return 'MyModule controlling %s' % self.modulename

    def __init__(self, parent=None, ID=-1, NAME=None):
        wx.Panel.__init__(self, parent, ID)
        self.modulename = NAME
        wx.GetApp().RegisterForQuickXPControl(self.modulename,

        ### Create widgets and register for events ###
                                                     self.QuickXPControlReceived)

    def QuickXPControlReceived(self, command, value, addr):
        print '%r=%r from %r' % (command, value, addr)
```

The interface widget is created and displayed by simply creating an instance of the widget. Given the above example, the widget would be created with the following command.

```
MyModuleControlPanel(NAME="SomeModuleInstance")
```

Chapter 4 - Pipe API

The following functions comprise the API needed by C application programmers to create and use QuickXP pipes. All functions are available from the QuickXP C library (libxpq.a or libxpq.so) and their prototypes are defined in xpq.h.

4.1 Creating and Opening Pipes

4.1.1 xpq_create

```
int xpq_create(const char *pipename, const char *device,
               char **pipedesc)
```

Description

Create a new pipe on the specified device of a default size (128MB). The pipe descriptor is used to open the pipe and must be unique. If no descriptor is given, an "anonymous" pipe will be created and the pipe descriptor is returned via the *pipedesc* parameter. If the *pipedesc* parameter is NULL and anonymous pipes are used, an error is returned.

Arguments

*pipename - Name descriptor of the pipe. A NULL or empty string will create an anonymous pipe with a unique name.

*device - Name of device to create the pipe on. Example: "/dev/xp/0".

**pipedesc - If not NULL, passes back a copy of the pipe descriptor. This field needs to be freed after the pipe is destroyed and must be valid when using anonymous pipes.

Returns

0 upon success, <0 on error

4.1.2 xpq_create_sized

```
int xpq_create_sized(const char *pipename, const char *device,
                     char **pipedesc, long long pipesize)
```

Description

Create a new pipe on the specified system of the specified size. See **`xpq_create`** for more details.

Arguments

*pipename - Name descriptor of the pipe. A NULL or empty string will create an anonymous pipe with a unique name.

*device - Name of device to create the pipe on. Example: "/dev/xp/0".

**pipedesc - If not NULL, passes back a copy of the pipe descriptor. This field needs to be freed after the pipe is destroyed and must be valid when using anonymous pipes.

pipesize - Size of the pipe in bytes.

Returns

0 upon success, <0 on error

4.1.3 xpq_destroy

```
int xpq_destroy(const char *pipename)
```

Description

Destroy a pipe and free up device resources. If any module has opened connections to the pipe, this routine just schedules the pipe to be destroyed – when the last open connection is closed, the pipe is destroyed.

NOTE: *pipedesc* returned from **`xpq_create*()`** needs to be freed after calling this function if used.

Arguments

*pipename <- Name descriptor of the pipe.

Returns

0 upon success, <0 on error

4.1.4 xpq_open

```
int xpq_open(const char *pipename, unsigned int flags)
```

Description

Open a connection to the pipe for use. The function returns an integer pipe handle on success. A pipe can be opened multiple times, but only tolerates a single writer. The XPQ_READ, XPQ_WRITE, and XPQ_STATUS flags are mutually exclusive.

The XPQ_WC_BYTES and XPQ_WC_ELEMENTS flags are also mutually exclusive. If the pipe is opened with XPQ_WC_BYTES, then the word counts are always specified in bytes for this connection. Otherwise, if opened with XPQ_WC_ELEMENTS (the default), the word counts is based on the element size, which depends on the metadata. If the metadata is missing or invalid, the word count defaults to bytes. See the section on *Metadata* for more info.

The XPQ_READ_LATEST flag is used for processes reading from the pipe that cannot keep up with data stream. On every read, QuickXP will skip to near the front of the pipe and throw away all of the data that it skips. This allows for snapshots of the latest data, albeit with data loss.

Arguments

*pipename - Name descriptor of the pipe

flags - Pipe options:

XPQ_READ : Open pipe for reading (moving data out of the pipe).
XPQ_WRITE : Open pipe for writing (moving data into the pipe).
XPQ_STATUS : Open pipe for monitoring (Default, no data movement).
XPQ_WC_BYTES : Counts specified in bytes.
XPQ_WC_ELEMENTS: Counts based upon datatype in metadata. (Default)
XPQ_READ_LATEST : Reads skip to end of pipe to always keep up. (Data is lost)

Returns

pipe handle upon success, <0 on error

4.1.5 xpq_close

```
int xpq_close(int handle)
```

Description

Closes the pipe connection described by *handle*. If the handle was opened with the XPQ_READ flag, then the pipe will advance the space available pointer to not consider the closed pipe's unread data.

Arguments

handle - Pipe handle.

Returns

0 upon success, <0 on error

4.2 Writing to Pipes

4.2.1 xpq_write_from_host

```
int xpq_write_from_host(int handle, void *hostbuf, int wc)
```

Description

Write data from the host to a pipe. This routine takes a buffer of data pointed to by *hostbuf* and writes into the pipe specified by *handle*. The amount of data to write is specified by the word count. The word size depends on how the pipe was opened and possibly the current metadata. See **`xpq_open()`** for more details.

The handle must describe a pipe opened with the XPQ_WRITE flag. If the handle is not writeable, this routine will return an error.

If there is not sufficient space in the pipe to hold the data, the routine blocks, waiting for space to become available. The routine returns once the buffer of data has been transferred.

Arguments

handle - Pipe handle.
*hostbuf - Host data buffer.
wc - Word count.

Returns

0 upon success, <0 on error

4.2.2 xpq_write_from_vp

```
int xpq_write_from_vp(int handle, vbuf_t *vbuf, int wc)
```

Description

Write data from a Vector Processor bank to a pipe. This routine takes a buffer of data pointed to by *vbuf* and writes into the pipe specified by *handle*. The amount of data to write is specified by the word count. The word size depends on how the pipe was opened and possibly the current metadata. See **`xpq_open()`** for more details.

The handle must describe a pipe opened with the XPQ_WRITE flag. If the handle is not writeable, this routine will return an error.

If there is not sufficient space in the pipe to hold the data, the routine blocks, waiting for space to become available. The routine returns once the buffer of data has been transferred.

Arguments

handle - Pipe handle.
*vbuf - VP data buffer.
wc - Word count.

Returns

0 upon success, <0 on error

4.3 Reading from Pipes

4.3.1 xpq_read_to_host

```
int xpq_read_to_host(int handle, void *hostbuf, int wc, int adv_wc,
                     int *wc_read)
```

Description

Read data from a pipe to the host. This function reads from a pipe and blocks until the *hostbuf* has been written with the amount of data requested or until metadata has changed. The amount of data to read is specified by the word count. The word size depends on how the pipe was opened and possibly the current metadata. See **`xpq_open()`** for more details.

The handle must describe a pipe opened with the XPQ_READ flag. If the handle is not readable, this routine will return an error.

QuickXP does not guarantee that the full word count will always be read as metadata transitions can cause the transfer to be truncated. The actual word count that is transferred is returned via the

wc_read parameter if it is not NULL. If data is truncated and *wc_read* is NULL, an error is generated. For this reason, *wc_read* must be valid unless you are not using metadata.

The *adv_wc* parameter describes how many words to increment the read pointer of the pipe. This means that if *adv_wc* is less than *wc*, some of the data will be read more than once. For example, if *wc* is 4 and *adv_wc* is 2, you would read words 1, 2, 3, 4 the first read. The second read would read words 3, 4, 5, 6.

Likewise, if *adv_wc* is greater than *wc*, some data will be thrown away. However, in this case, if the pipe does not have enough data to satisfy the *adv_wc*, the amount of data skipped after the data read will be silently truncated. So you should only use it this way if you know that the pipe writer will always supply enough data in a single write to satisfy the reader's *adv_wc*.

Arguments

handle - Pipe handle.
*hostbuf - Host data buffer.
wc - Word count of data to read into the host buffer.
adv_wc - Word count to advance read pointer.
*wc_read - Returned actual word count transferred.

Returns

XPQ_ST_METADATA_CHANGED on metadata change detected,
0 upon success, <0 on error

4.3.2 xpq_read_to_vp

```
int xpq_read_to_vp(int handle, vbuf_t *vbuf, int wc, int adv_wc,
                   int *wc_read)
```

Description

Read data from a pipe to a Vector Processor bank. This function reads from a pipe and blocks until the *vbuf* has been written with the amount of data requested or until metadata has changed. The amount of data to read is specified by the word count. The word size depends on how the pipe was opened and possibly the current metadata. See **`xpq_open()`** for more details.

The handle must describe a pipe opened with the XPQ_READ flag. If the handle is not readable, this routine will return an error.

QuickXP does not guarantee that the full word count will always be read as metadata transitions can cause the transfer to be truncated. The actual word count that is transferred is returned via the *wc_read* parameter if it is not NULL. If data is truncated and *wc_read* is NULL, an error is generated. For this reason, *wc_read* must be valid unless you are not using metadata.

The *adv_wc* parameter describes how many words to increment the read pointer of the pipe. This means that if *adv_wc* is less than *wc*, some of the data will be read more than once. For example, if *wc* is 4 and *adv_wc* is 2, you would read words 1, 2, 3, 4 the first read. The second read would read words 3, 4, 5, 6.

Likewise, if *adv_wc* is greater than *wc*, some data will be thrown away. However, in this case, if the pipe does not have enough data to satisfy the *adv_wc*, the amount of data skipped after the data read will be silently truncated. So you should only use it this way if you know that the pipe writer will always supply enough data in a single write to satisfy the reader's *adv_wc*.

Arguments

handle - Pipe handle.
*vbuf - VP data buffer.
wc - Word count of data to read into the host buffer.
adv_wc - Word count to advance read pointer.
*wc_read - Returned actual word count transferred.

Returns

XPQ_ST_METADATA_CHANGED on metadata change detected,
0 upon success, <0 on error

4.4 Writing and Reading Pipe Metadata

This section describes how metadata is written to and read from pipes. See “3.1.2 - Metadata” for more information on how metadata is used.

4.4.1 xpq_meta_write

```
int xpq_meta_write(int handle, void *ptr, int sz8, unsigned int flags)
```

Description

Writes new metadata to a pipe. This metadata is associated with all writes to this pipe until this function is called again. A copy of the metadata is made at the time the function is called, so the calling program may modify/reuse its copy of the metadata at any time.

The handle must describe a pipe opened with the XPQ_WRITE flag. If the handle is not writeable, this routine will return an error.

It is strongly recommended that the first element of the metadata is always **`tms_meta_t`**. This is required to take advantage of element counts (the pipe opened with the XPQ_WC_ELEMENTS flag).

Arguments

handle - Pipe handle
*ptr - New metadata to write (can be NULL).
sz8 - Size of metadata in bytes.
flags - Set to 0 (reserved for future use).

Returns

0 upon success, <0 on error

4.4.2 xpq_meta_read

```
int xpq_meta_read(int handle, void **ptr, int *sz8, unsigned int flags)
```

Description

Grabs a pointer to the current metadata from a pipe. If this is the first read to the pipe, the function will block until data is available. After the pipe has been read, this function will always return immediately with the current metadata.

The **`xpq_read_to*()`** function status will indicate when metadata has changed and needs to be read again.

The handle must describe a pipe opened with the XPQ_READ flag. If the handle is not readable, this routine will return an error.

The metadata pointer is to a shared memory location. The metadata should not be modified under any circumstances, and the pointer should be freed with **`xpq_meta_free()`** when it is no longer being used.

Arguments

handle - Pipe handle.
**ptr - Returned pointer to new metadata.
*sz8 - Returned size of metadata in bytes.
flags - Set to 0 (reserved for future use).

Returns

0 upon success, <0 on error

4.4.3 xpq_meta_free

`int xpq_meta_free(void *ptr)`

Description

Removes a reference to a metadata structure. When all owners have released their reference, the memory is freed. You must call this function for every pointer received from xpq_meta_read() to release the memory.

Arguments

*ptr - Metadata to free.

Returns

0 upon success, <0 on error

4.4.4 xpq_meta_set_cast_func

`int xpq_meta_set_cast_func(cast_meta_func_t func)`

Description

Sets the metadata casting function used when QuickXP detects unfamiliar (not **`tms_meta_t`**) metadata. This function pointer defaults to NULL, and no action is taken in this case. **`cast_meta_func_t`** is defined in **`tms_meta.h`**.

Arguments

func - Pointer to function that converts non-QuickXP metadata.

Returns

0 in all cases

4.5 Metadata Creation

4.5.1 tms_meta_init

`static inline void tms_meta_init(tms_meta_t * meta)`

Description

Initialize a TMS metadata structure (**`tms_meta_t`**). You must call this function at least once to properly setup the magic number. Otherwise, QuickXP will not recognize this structure when determining the datatype for transfer sizes.

This will default to a single dimension of time domain data.

Arguments

meta - Pointer to the **`tms_meta_t`** structure to initialize.

Returns

Nothing

4.5.2 tms_meta_init_from_file

`int tms_meta_init_from_file(tms_meta_t * meta, const char * filename)`

Description

Initialize the metadata structure (**`tms_meta_t`**) from a TMS Metadata XML file. An example of the format used is below:

```
<tms:metadata version="1" xmlns:tms="http://www.texmemsys.org/XML/MetaData">
    <tms:datatype>VF32</tms:datatype>
```

```
        <tms:datarate>1M</tms:datarate>
        <tms:starttime>10M</tms:starttime>
        <tms:endtime>20M</tms:endtime>
        <tms:dimensions>2</tms:dimensions>
        <tms:xstart>100.0</tms:xstart>
        <tms:xdelta>5.0</tms:xdelta>
        <tms:xunits>FREQUENCY</tms:xunits>
        <tms:framesize>32768</tms:framesize>
        <tms:ystart>200.0</tms:ystart>
        <tms:ydelta>20.0</tms:ydelta>
        <tms:yunits>TIME</tms:yunits>
    </tms:metadata>
```

Arguments

meta - The metadata structure to initialize.
filename - The name of the XML file to parse.

Returns

0 on success, <0 on error

4.5.3 tms_meta_datatype_from_string

```
int tms_meta_datatype_from_string(const char * str,
                                  unsigned int * datatype)
```

Description

Convert the string representing the datatype to the actual datatype value. See the *LIBXP* manual for more information on valid datatypes.

Arguments

str - The string representing the datatype.
datatype - Pointer to location to store result (untouched on error).

Returns

0 on success, <0 on error

4.5.4 tms_meta_double_mag10_from_string

```
int tms_meta_double_mag10_from_string(const char * str, double * value,
                                      const char ** endptr)
```

Description

Convert a string to a double precision floating-point interpreting the character immediately following as a magnitude (power of 10). Valid trailing characters are:

```
'p': value * e-12
'n': value * e-9
'u': value * e-6
'm': value * e-3
'k': value * e3
'M': value * e6
'G': value * e9
```

Arguments

str - The string value.
value - Pointer to location to store result (untouched on error).
endptr - Pointer to location to store the character immediately following that last character used in the conversion (untouched on error).

Returns

0 on success, <0 on error

4.5.5 tms_meta_units_from_string

```
int tms_meta_units_from_string(const char * str, unsigned int * units)
```

Description

Convert the string representing the type of units to the actual unit type. Valid unit strings are: TIME, FREQUENCY (or FREQ), and NONE.

Arguments

str - The string value.
units - Pointer to location to store result (untouched on error)

Returns

0 on success, <0 on error

4.6 Retrieving Pipe Information

4.6.1 xpq_get_datasize

```
int xpq_get_datasize(int handle, int * bytes)
```

Description

Retrieves the current element size used by the Pipe to determine the size of transfer word counts. This information is also available by reading the metadata and examining the datatype.

Arguments

handle - Pipe handle.
*bytes - Returned element size in bytes.

Returns

0 in all cases

4.6.2 xpq_get_stats

```
int xpq_get_stats(int handle, xpq_pipe_stats_t *stats)
```

Description

Queries the pipe for usage statistics.
The handle must describe a pipe opened with the XPQ_STATUS flag, otherwise this routine will return an error.

Arguments

handle - Pipe handle.
*stats - Returned stats structure:
read_completed – Total bytes read from pipe.
read_scheduled – Total bytes read and scheduled to be read from pipe.
write_completed – Total bytes written to pipe.
write_scheduled – Total bytes written and scheduled to be written from pipe.
sam_addr – Starting address in SAM memory where the pipe resides.
sam_size – Size of the pipe in bytes.
num_write_handles – Number of write connections to pipe (cannot be greater than 1).
num_read_handles – Number or read connections to pipe.
num_status_handles – Number of status connections to pipe.

Returns

0 upon success, <0 on error

Chapter 5 - Module API

The following functions comprise the API needed by C application programmers to create and use modules. All functions are available from the QuickXP C library (libxpq.a or libxpq.so) and their prototypes are defined in xpq.h.

5.1 Module Creation

5.1.1 XPQ_MODULE

```
#define XPQ_MODULE(NAME, FUNCTION, HELP)
```

Description

This macro creates a module declaration with the given name and uses the *FUNCTION* parameter as the module's main function. See "3.1 - Module Programming" for more information on creating modules.

Arguments

NAME - Name of the module to define.
FUNCTION - Name of the module "main" function.
HELP - String description of the module.

Returns

N/A

5.1.2 XPQ_TOPLEVEL

```
#define XPQ_TOPLEVEL(NAME)
```

Description

This macro creates a "main" program declaration that uses the given module as the toplevel program. This allows a module to be run as a standalone program. See "3.1 - Module Programming" for more information on creating modules.

Arguments

NAME - Name of the toplevel module.

Returns

N/A

5.1.3 xpq_module_get_name

```
const char * const xpq_module_get_name()
```

Description

Return the name of the module from which this function is called.

Arguments

None

Returns

The module name.

5.1.4 xpq_module_get_fullpath

```
const char * const xpq_module_get_fullpath()
```

Description

Return the full module-path of the module from which this function is called.

Arguments

None

Returns

The module-path.

5.2 Module Control

5.2.1 xpq_control_addr_init

```
void xpq_control_addr_init(xpq_control_addr_t * addr,
                           const char * path)
```

Description

Initializes an QuickXP Control Address to refer to the control point associated with *path*. If the path equals "..", *addr* will reference the parent of the current module. If the path starts with "." (and is not equal to ".."), *addr* will reference the module whose full pathname is *path* without the leading ".". If the path starts with "@", *addr* will reference the virtual control pointed named *path* (without the leading "@"). Otherwise, the path is taken to be the name of a sub-module of the current module, and *addr* will refer to that child module.

Arguments

addr - The QuickXP Control Address structure to initialize.
path - The path to reference.

Returns

Nothing

5.2.2 xpq_control_addr_get_name

```
const char * xpq_control_addr_get_name(xpq_control_addr_t * addr)
```

Description

Returns the name of the module referenced by the QuickXP Control Address.

Arguments

addr - The QuickXP Control Address structure to query.

Returns

Module name.

5.2.3 xpq_control_addr_get_fullpath

```
const char * xpq_control_addr_get_fullpath(xpq_control_addr_t * addr)
```

Descirption

Returns the full path of the module referenced by the QuickXP Control Address.

Arguments

addr - The QuickXP Control Address structure to query.

Returns

Module path.

5.2.4 xpq_control_send

```
int xpq_control_send(const xpq_control_addr_t * addr,
                     const char * name, const char * value)
```

Description

Sends the control message *name* = *value* (or just *name* if *value* is NULL) to the module referenced by *addr*. This is an asynchronous fire-and-forget message.

Arguments

addr - References the module to which the message will be sent.
name - The control message-name.
value - The value to associate with the control message-name (may be NULL).

Returns

0 on success, <0 on error

5.2.5 xpq_control_sendto

```
int xpq_control_sendto(const char * path, const char * name,
                       const char * value)
```

Description

Sends a control message to a module referenced by *path*. This function is essentially a shortcut for the sequence:

```
xpq_control_addr_init(&addr, path);
return xpq_control_send(&addr, name, value);
```

Arguments

path - The path (relative or absolute) of the module to reference.
name - The control message-name.
value - The value to associate with the control message-name (may be NULL).

Returns

0 on success, <0 on error

5.2.6 xpq_control_receive

```
int xpq_control_receive(char ** name, char ** value,
                        xpq_control_receive_mode_t mode,
                        xpq_control_addr_t * addr)
```

Description

This function checks for the existence of any QuickXP control messages for the module from which this function is called. If any messages are pending, the oldest in the queue will be retrieved and its name and value will be stored in *name* and *value* (note that the result stored in *value* may be NULL if no value was sent with the message).

mode may be one of XPQ_CONTROL_RECEIVE_NOWAIT, XPQ_CONTROL_RECEIVE_WAIT, or XPQ_CONTROL_RECEIVE_PEEK. XPQ_CONTROL_RECEIVE_NOWAIT (the default) will cause the function to return immediately if no messages are currently pending. XPQ_CONTROL_RECEIVE_WAIT will block indefinitely waiting for a message to arrive. XPQ_CONTROL_RECEIVE_PEEK behaves like XPQ_CONTROL_RECEIVE_NOWAIT, except the message returned will be left at the head of the receive queue so subsequent calls to xpq_control_receive will retrieve that same message again.

If *addr* is non-NULL, it will be filled in with the QuickXP Control address of the module that sent the message.

This function returns 0 if no messages are currently pending, a positive number representing the total number of bytes in the current message if a control message is successfully retrieved, and a negative number indicating the problem on error.

NOTE: **`xpq_control_release()`** MUST be called with the *name* and *value* set by this function after a successful return in order to release any resources allocated. It MUST NOT be called if the return value is <= 0.

Arguments

name - Set to the control message-name (if any message is pending).
value - Set to the control message-value (if any message is pending).
mode - The receive mode (blocking, non-blocking, peek), default is 0 (XPQ_CONTROL_RECEIVE_NOWAIT).
addr - If non-NULL, set to the address of the message's sender.

Returns

Number of bytes when a new message is received, 0 on no message pending, <0 on error

5.2.7 xpq_control_release

`void xpq_control_release(char * name, char * value)`

Description

Release the message resources obtained from a successful **xpq_control_receive()** call.

Arguments

name - The name returned from the previous **`xpq_control_receive()`** call.
value - The value returned from the previous **`xpq_control_receive()`** call.

Returns

Nothing

5.2.8 xpq_active

`int xpq_active()`

Description

Returns the whether or not the module thread has been stopped or not. This is intended to be looped on in the thread's main function. Currently this function is an exit point for the module, but that functionality may change in the future.

Arguments

None

Returns

1 if thread is running, if stopped will return 0 or exit the thread

5.2.9 xpq_stop_all_modules

`void xpq_stop_all_modules()`

Description

This function issues a stop to all QuickXP threads. All calls to **`xpq_active()`** should either return 0 or break out of the thread after this command is issued.

Arguments

None

Returns

Nothing

5.2.10 xpq_wait_all_modules

`void xpq_wait_all_modules()`

Description

This function waits on all QuickXP threads to exit. If a thread fails to exit cleanly, this function will hang indefinitely.

Arguments

None

Returns

Nothing

5.3 Module Cleanup

5.3.1 xpq_module_cleanup_push

`void xpq_module_cleanup_push(module_cleanup_fn, arg)`

Description

This function takes the function pointer and the argument and puts them onto a cleanup stack. Upon the threads exit, the functions are popped off of the stack and executed with the given arguments. The corresponding **`xpq_module_cleanup_pop()`** function also removes the top function from the stack and conditionally executes it. The function supports both 32-bit and 64-bit arguments.

This function provides a clean way to schedule cleanup without messy cleanup routines or having to unroll your cleanup code to handle errors during execution or thread cancellation.

This function is analogous to the Unix **`pthread_cleanup_push()`** function.

Arguments

module_cleanup_fn – Function pointer to a cleanup routine that returns nothing takes a single argument (either 32-bit or 64-bit, depending on the platform).

arg – 32-bit or 64-bit value to pass to the cleanup function.

Returns

Nothing

5.3.2 xpq_module_cleanup_pop

`void xpq_module_cleanup_pop(int execute)`

Description

This function pops the top function (which is the last function put on the stack with **`xpq_module_cleanup_push()`**) off of the cleanup stack. If the execute parameter is set, the function is executed with the previously given argument. If not, the function is simply discarded.

This function is analogous to the Unix **`pthread_cleanup_pop()`** function.

Arguments

execute – runs the function popped of the cleanup stack when set, discards otherwise

Returns

Nothing

5.4 Module Flow Control

5.4.1 xpq_control_init

`static inline void xpq_control_init(xpq_control_flow_t *ctrl);`

Description

Initializes control structure used with **`xpq_control_process`** function.

Arguments

ctrl - Pointer to control structure to initialize

Returns

Nothing

5.4.2 xpq_control_sleep

```
inline int xpq_control_sleep(long long usecs)
```

Description

Sleep for specified microseconds unless control becomes available, in which case the function returns early. The function returns -1 on errors, XPQ_ST_CONTROL_AVAIL if interrupted by control, and 0 if the full sleep occurred.

Arguments

usecs - Time to sleep in microseconds.

Returns

XPQ_ST_CONTROL_AVAIL if interrupted by control,
0 if no interrupt occurred, <0 on error.

5.4.3 xpq_control_flow

```
static inline int xpq_control_flow(long long *tstamp,
                                   long long *tsurplus,
                                   double rate, unsigned int elements)
```

Description

Strategically sleeps to maintain a particular data rate. The *tstamp* and *tsurplus* variables need to be initialized to zero, but otherwise don't need to be modified outside of this function. QuickXP Control will interrupt the sleep and cause the function to return early. *tsurplus* accounts for this on the next call to the function.

Arguments

*tstamp - Persistent microsecond timestamp value.
*tsurplus - Current amount of surplus time in microseconds.
rate - Data rate in elements/second.
elements - Number of elements transferred/processed.

Returns

XPQ_ST_GOOD upon sleeping with no control available,
XPQ_ST_CONTROL_AVAIL if new control is available,
XPQ_ST_POTENTIAL_CONTROL if control *might* be available, <0 on error

5.4.4 xpq_elements_per_time

```
static inline unsigned int xpq_elements_per_time(double rate,
                   unsigned int alignment, unsigned int timeslice_us)
```

Description

Returns the number of elements that should be processed given the data rate and the desired timeslice. The basic equation used is: rate*timeslice_us/1000000. Elements are added to ensure the alignment is maintained.

Arguments

rate - Data rate in elements/second.
alignment - Minimum data alignment in elements.

timeslice_us - Desired time per transaction in microseconds.

Returns

Number of elements to be processed in the given timeslice

5.4.5 xpq_control_process

```
static inline int xpq_control_process(void *hfn, void *hfn_param,
                                      xpq_control_flow_t *ctrl,
                                      double rate, unsigned int elements);
```

Description

Integrates control processing and flow rate control into one function. This allows for efficient command processing and support for execution pausing. All pending commands will be processed before exiting.

The *hfn* parameter is a pointer to a parameter handler that will be called for any pending control. This routine should be of type *xpq_control_handler_fn*, with the first parameter being some user-defined data type that is passed in via *hfn_param*.

The *ctrl->update* field is always passed in to the update parameter of *hfn*.

NOTE: For pausing to work, you must embed the control struct into hfn_param so that the handler has access to the paused variable.

Arguments

*hfn - Pointer to command handler function (of type *xpq_control_handler_fn*)

*hfn_param - User data to be passed into first parameter of the handler function

*ctrl - Pointer to local control structure (defined in xpq.h)

rate - Data rate in elements/second

elements - Number of elements transferred/processed

Returns

0 on success, <0 on error

5.4.6 xpq_control_parse_argv

```
static inline int xpq_control_parse_argv(void *hfn, void *hfn_param,
                                         int argc, char **argv, int argoff);
```

Description

Helper function for parsing command line parameters using the standard command handler. This function depends on the command handler returning 0 when only one argument is taken and 1 when the second argument (value) is consumed.

The *hfn* parameter is a pointer to a parameter handler that will be called for any pending control. This routine should be of type *xpq_control_handler_fn*, with the first parameter being some user-defined data type that is passed in via *hfn_param*.

A NULL will be passed into the update parameter of *hfn*.

Arguments

*hfn - Pointer to command handler function (of type *xpq_control_handler_fn*)

*hfn_param - User data to be passed into first parameter of the handler function

argc, argv - Standard command line values

argoff - Starting offset into argv array

Returns

0 on success, <0 on error

5.5 Module Error Logging

5.5.1 xpq_log_error

```
void xpq_log_error(const char * fmt, ...)
```

Description

This function logs errors using **`printf()`** style arguments and provides additional module debug information.

Arguments

fmt - Standard printf style arguments.

Returns

Nothing

Chapter 6 - Scripting API

In addition to the C programming API, QuickXP also provides a Python scripting interface for many of the QuickXP functions. The following Python functions provide the API needed to create and use pipes and modules. You must use **`import xpq`** in your scripts to have access to the API. Most of these functions are simply Python wrappers for the functions already defined in the C programming API. Refer to the previous chapters for more detailed information on the C function calls.

6.1 Creating and Opening Pipes

6.1.1 xpq.create

`xpq.create([name,] device [, pipesize]) -> pipedesc`

Description

Create a new pipe on the specified device.

Wrapper

If *pipesize* is specified:

`xpq_create_sized()`

Otherwise:

`xpq_create()`

Arguments

name - Name descriptor of the pipe. A missing or empty string will create an anonymous pipe with a unique name.

device - Name of device to create the pipe on. Example: "/dev/xp/0".

pipesize - Size of the pipe to create in bytes.

Returns

Name descriptor of pipe on success, throws exception on error

6.1.2 xpq.destroy

`xpq.destroy(name)`

Description

Destroy a pipe and free up device resources.

Wrapper

`xpq_destroy()`

Arguments

*name <- Name descriptor of the pipe.

Returns

Nothing on success, throws exception on error

6.1.3 xpq.open

```
xpq.open(name, flags = READ|WRITE|STATUS [ + WC_BYTES|WC_ELEMENTS]
                                         [ + READ_LATEST]) -> handle
```

Description

Open a connection to the pipe for use.

Wrapper

`xpq_open()`

Arguments

*name - Name descriptor of the pipe
flags - Pipe options:

READ	: Open pipe for reading (moving data out of the pipe).
WRITE	: Open pipe for writing (moving data into the pipe).
STATUS	: Open pipe for monitoring (Default, no data movement).
WC_BYTES	: Counts specified in bytes.
WC_ELEMENTS	: Counts based upon datatype in metadata. (Default)
READ_LATEST	: Reads skip to end of pipe to always keep up. (Data is lost)

Returns

pipe handle upon success, throws exception on error

6.1.4 xpq.close

`xpq.close(handle)`

Description

Closes the Pipe connection described by *handle*.

Wrapper

`xpq_close()`

Arguments

handle - Pipe handle.

Returns

Nothing on success, throws exception on error

6.2 Writing to Pipes

6.2.1 xpq.write_from_host

`write_from_host(handle, buf, count)`

Description

Write data from an object via its read-buffer-interface to a pipe.

Wrapper

`xpq_write_from_host()`

Arguments

handle - Pipe handle.
*buf - Host data buffer.
count - Word count.

Returns

Nothing on success, throws exception on error

6.2.2 xpq.write_from_vp

`xpq.write_from_vp(handle, vbuf, count)`

Description

Write data from a Vector Processor bank to a pipe.

Wrapper

`xpq_write_from_vp()`

Arguments

handle - Pipe handle.
*vbuf - VP data buffer.
count - Word count.

Returns

Nothing on success, throws exception on error

6.3 Reading from Pipes

6.3.1 xpq.read_to_host

`xpq.read_to_host(handle, buf, count [, advance=count]) -> wc_read`

Description

Read data from a pipe to an object via its write-buffer-interface.

Wrapper

`xpq_read_to_host()`

Arguments

handle - Pipe handle.
*buf - Host data buffer.
count - Word count of data to read into the host buffer.
advance - Word count to advance read pointer.

Returns

Actual word count transferred on success,
throws exception xpq.MetadataChangedNotice on metadata change detected,
throws other exception on error

6.3.2 xpq.read_to_vp

`xpq.read_to_vp(handle, vbuf, count [, advance=count]) -> wc_read`

Description

Read data from a pipe to a Vector Processor bank.

Wrapper

`xpq_read_to_vp()`

Arguments

handle - Pipe handle.
*vbuf - VP data buffer.
count - Word count of data to read into the host buffer.
advance - Word count to advance read pointer.

Returns

Actual word count transferred on success,
throws exception xpq.MetadataChangedNotice on metadata change detected,
throws other exception on error

6.4 Writing and Reading Pipe Metadata

6.4.1 xpq.meta_write

`xpq.meta_write(handle, data [, flags=0])`

Description

Writes new metadata to a pipe.

Wrapper

`xpq_meta_write()`

Arguments

handle - Pipe handle
*data - New metadata to write (can be NULL).
flags - Set to 0 (reserved for future use).

Returns

Nothing on success, throws exception on error

6.4.2 xpq.meta_read

`xpq.meta_read(handle [, flags=0]) -> metadata_buffer`

Description

Grabs a pointer to the current metadata from a pipe.

Wrapper

`xpq_meta_read()`

Arguments

handle - Pipe handle.
flags - Set to 0 (reserved for future use).

Returns

New metadata upon success, throws exception on error

6.4.3 xpq.meta_free

`xpq.meta_free(metadata_buffer)`

Description

Removes a reference to a metadata structure.

Wrapper

`xpq_meta_free()`

Arguments

metadata_buffer - Metadata to free.

Returns

Nothing on success, throws exception on error

6.5 Metadata Creation

6.5.1 xpq.meta_init_from_file

`xpq.meta_init_from_file(filename) -> metadata_buffer`

Description

Initialize the metadata structure from a TMS Metadata XML file.

Wrapper

`tms_meta_init_from_file()`

Arguments

meta - The metadata structure to initialize.
filename - The name of the XML file to parse.

Returns

Metadata buffer on success, throws exception on error

6.6 Retrieving Pipe Information

6.6.1 xpq.get_stats

`xpq.get_stats(handle [, stats=None]) -> PipeStats object`

Description

Query the pipe statistics of the given handle. The optional second parameter must be None or a PipeStats object (default is None). If it is None, a new PipeStats object containing the current statistics will be created and returned. If it is a valid PipeStats object (such as that returned from a previous invocation of this function with this parameter set to None), that object's members will be updated with the current statistics and it will be returned.

Wrapper

`int xpq_get_stats()`

Arguments

handle - Pipe handle.
stats – Updated and returned stats structure (if specified)

Returns

New stats structure if **`stats`**=None, otherwise **`stats`**, throws exception on error

6.7 Module Declaration

6.7.1 @xpq.FunctionModule

```
@xpq.FunctionModule
def NAME(*args, **kwds):
    “HELP…”
    # Module implementation…
```

Description

This Python method decorator is used to declare a standard Python method as a module. When the function wrapped with this decorator is called, a new module instance will be created executing the body of the function. See the “Declaring a Python function Module” section in “3.1.4 - Declaring Modules” for more information.

Wrapper

```
#define XPQ_MODULE(NAME, FUNCTION, HELP)
```

Arguments

N/A

Returns

N/A

6.7.2 xpq.ClassModule

```
class NAME(xpq.ClassModule):
    "HELP…"
    def main(self, *args, **kwds):
        # Module implementation…
```

Description

This Python class is used as the base class for Modules implement as Python classes. When the class descended from **xpq.CLassModule** is instantiated, a new Module instance will be created executing the body of the new instance's **main** function. See the "Declaring a Python function Module" section in "3.1.4 - Declaring Modules" for more information.

Wrapper

```
#define XPQ_MODULE(NAME, FUNCTION, HELP)
```

Arguments

N/A

Returns

N/A

6.8 Module Information

6.8.1 xpq.module_get_name

```
xpq.module_get_name() -> name
```

Description

Return the name of the module from which this function is called.

Wrapper

```
const char * const xpq_module_get_name()
```

Arguments

None

Returns

The module name, throws exception on error

6.8.2 xpq.module_get_fullpath

```
xpq.module_get_fullpath() -> path
```

Description

Return the full module-path of the module from which this function is called.

Wrapper

```
const char * const xpq_module_get_fullpath()
```

Arguments

None

Returns

The module-path, throws exception on error

6.9 Module Control

6.9.1 xpq.control_send

`xpq.control_send(addr, name [, value=None])`

Description

Send Control data to the specified addr.

Wrapper

`int xpq_control_send()`

Arguments

addr - References the module to which the message will be sent.
name - The control message-name.
value - The value to associate with the control message-name (may be None).

Returns

Nothing on success, throws exception on error

6.9.2 xpq.control_receive

`xpq.control_receive([mode=0]) -> None | (name, value, addr)`

Description

Receive Control data.

Wrapper

`int xpq_control_receive()`

Arguments

mode - The receive mode (blocking, non-blocking, peek), default is 0
(XPQ_CONTROL_RECEIVE_NOWAIT).

Returns

The name, value, and sender address if message is available,
Nothing on no message pending, throws exception on error

6.9.3 xpq.control_release

`xpq.control_release(name, value)`

Description

Release the message resources obtained from a successful **xpq.control_receive()** call.

Wrapper

`void xpq_control_release()`

Arguments

name - The name returned from the previous **`xpq.control_receive()`** call.
value - The value returned from the previous **`xpq.control_receive()`** call.

Returns

Nothing

6.9.4 xpq.active

```
xpq.active() -> bool
```

Description

Returns the whether or not the module thread has been stopped or not.

Wrapper

```
int xpq_active()
```

Arguments

None

Returns

True if thread is running, False if stopped

6.9.5 xpq.modules.stop

```
import xpq.modules
xpq.modules.stop()
```

Description

This function issues a stop to all QuickXP threads.

Wrapper

```
void xpq_stop_all_modules()
```

Arguments

None

Returns

Nothing

6.9.6 xpq.modules.wait

```
import xpq.modules
xpq.modules.wait()
```

Description

This function waits on all QuickXP threads to exit.

Wrapper

```
void xpq_wait_all_modules()
```

Arguments

None

Returns

Nothing

6.10 Module Flow Control

6.10.1 xpq.control_sleep

```
control_sleep(usecs)
```

Description

Sleep for specified microseconds unless control becomes available.

Wrapper

```
inline int xpq_control_sleep()
```

Arguments

usecs - Time to sleep in microseconds.

Returns

XPQ_ST_CONTROL_AVAIL if interrupted by control,
0 if no interrupt occurred, throws exception on error

Chapter 7 - Pre Existing Modules

Several Modules have been developed by TMS and distributed with the QuickXP development system as the **`xpqmodules`** Module Package. These are modules that have widespread applicability and usefulness. They include modules that can continuously play data from a file into a pipe, write data to a file from a pipe and display data from a pipe.

7.1 File Reader Module

The file reader module, **`rdfile`**, continuously plays data from a file into a pipe. It supports various control messages to control this playback, allowing for dynamic modification of the playback rate, the range of the data played, pausing, changing the current playback position, and others. It also supports sending control to inform other module's of the current playback range, current playback position, data file name, pipe name, and many others. It is supplied with a GUI control widget providing an intuitive VCR style interface to this functionality. This module expects two instantiation arguments, the filename of the file to playback and the pipename of the pipe to which it will write. Other optional arguments may be supplied. They are the same as the control that may be sent to this module, as documented in the following table. Note that this module ignores case for argument/control command names.

Name	Value Sent	Value Accepted
pipename	Pipe Name	N/A
filename	Data Filename	N/A
metadata	XML Metadata Filename (if any)	XML Metadata Filename or empty string
range	START:END (of entire file in current xunits)	N/A
subrange	START:END (of current playback range in current xunits)	New START:END playback range (in current xunits)
position	POSITION (in current xunits)	New POSITION (in current xunits)
pause	1 if paused, 0 if not	1 to pause, 0 to play
datarate	Playback datarate (in Samples/sec)	Playback datarate (in Samples/sec)
datatype	Type of data in file	Type of data in file
xunits	Metadata xunits	Metadata xunits
xdelta	Metadata xdelta	Metadata xdelta

The pipename, filename and metadata elements currently can only be set when the module is instantiated. Attempts to change them while the module is running will be ignored. This may be changed in future versions of QuickXP. The metadata argument is used to specify a TMS Metadata XML file used as the initial metadata describing the data file. If the metadata argument is not specified (or is an empty string), the module will look for a file in the same directory as the data file, whose name is the same as the data file but with ".metadata" appended.

7.2 File Writer Module

The file writer Module, **`wrfile`**, continuously writes data into a file from a pipe. It supports various control messages to control this writeback, allowing for dynamic modification of the range of the file written to, the current writeback position, and pausing writeback. It also supports sending control to inform other module's of the current writeback range, current writeback position, data file name, pipe name, and others. This module expects two instantiation arguments, the pipename of the pipe to read and the filename of the file to which it will write. Other optional arguments may be supplied. They are the same as the control

that may be sent to this module, as documented in the following table. Note that this module ignores case for argument/control command names.

Name	Value Sent	Value Accepted
pipename	Pipe Name	N/A
filename	Data Filename	N/A
subrange	START:END (of current writeback range in current xunits)	New START:END writeback range (in current xunits)
position	POSITION (in current xunits)	New POSITION (in current xunits)
pause	1 if paused, 0 if not	1 to pause, 0 to record

The pipename and filename elements currently can only be set when the module is instantiated. Attempts to change them while the module is running will be ignored. This may be changed in future versions of QuickXP.

7.3 Pipe Statistics Module

The pipe statistics module **`pipestats`** is a GUI module that will periodically query and graphically display the information returned in the **`xpq_pipe_stats_t`** structure for a specific pipe. It has a single instantiation parameter, the name of the pipe to periodically query.

7.4 Data Plot Module

The real-time plotter GUI module **`rtplot.gui`** will graphically display the data it reads from a pipe. This is shown as a two dimensional line-plot of the data. It requires one instantiation parameter, the name of the pipe to read. It supports the optional instantiation parameter **`framesize`**, which it uses to determine how much data to read and display at a time. This framesize will track any received metadata changes indicating multi-dimensional data in the pipe. This module supports several user-interface actions via the mouse to control zooming on the data displayed. The widget maintains a history of zoom settings, called the zoom stack. Several user-interface actions allow the user to move back and forth through this history of previous views. Any new views pushed on the stack by a zoom-in, zoom-out or zoom-range operation will invalidate the entries that have been popped.

- Zoom-range: Holding the left mouse button down while moving the mouse in the plot window will allow the user to select a region of the display to zoom in on. When the left button is released, the region inside the selection will fill the entire plotter display.
- Zoom-In/Zoom-Out: Rolling the mouse scroll wheel forward or back will zoom both the X and Y axis in or out by a factor of two, centered at the center point of the current view.
- Zoom-In/Zoom-Out Y Only: Rolling the mouse scroll wheel forward or back with the Shift button held down will zoom just the Y axis in or out by a factor of two, centered at the center point of the current view.
- Zoom-In/Zoom-Out X Only: Rolling the mouse scroll wheel forward or back with the Ctrl button held down will zoom just the X axis in or out by a factor of two, centered at the center point of the current view.
- Zoom-History-Previous: Right clicking will restore the previous view in the zoom history.
- Zoom-History-Previous/Next: Rolling the mouse scroll wheel forward or back with the Alt button held down will scroll through the history list, restoring the previous or next view depending on the direction the wheel is rolled.

7.5 Data Grid Module

The data grid module **`rtplot.grid`** will display the data it reads from a pipe in a spreadsheet like grid. Each element in the data (as defined by the current metadata's **`datatype`** setting) will be displayed in a separate cell. It requires one instantiation parameter, the name of the pipe to read. It supports the

optional instantiation parameter **`framesize`**, which it uses to determine how much data to read and display at a time. This framesize will track any received metadata changes indicating multi-dimensional data in the pipe. The number of columns displayed is currently fixed at 8. The number of rows will depend on the current framesize setting. This module is unique in that it does not progress to the next frame of data until the user has clicked the "Next Frame" button. This allows the user to step through successive frames of data.