ECHELON®

Pyxos FT Programmer's Guide

078-0339-01A

EACH USER OF THE PYXOS FT CHIP AND PROTOCOL ASSUMES RESPONSIBILITY FOR, AND HEREBY AGREES TO USE ITS BEST EFFORTS IN, DESIGNING AND MANUFACTURING EQUIPMENT LICENSED HEREUNDER TO PROVIDE FOR SAFE OPERATION THEREOF, INCLUDING, BUT NOT LIMITED TO, COMPLIANCE OR QUALIFICATION WITH RESPECT TO ALL SAFETY LAWS, REGULATIONS AND AGENCY APPROVALS, AS APPLICABLE. THE PYXOS FT CHIP AND PROTOCOL ARE NOT DESIGNED OR INTENDED FOR USE AS COMPONENTS IN EQUIPMENT INTENDED FOR SURGICAL IMPLANT INTO THE BODY, OR OTHER APPLICATIONS INTENDED TO SUPPORT OR SUSTAIN LIFE, FOR USE IN FLIGHT CONTROL OR ENGINE CONTROL EQUIPMENT WITHIN AN AIRCRAFT, OR FOR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE PYXOS FT CHIP OR PROTOCOL COULD CREATE A SITUATION IN WHICH PERSONAL INJURY OR DEATH MAY OCCUR, AND THE USER SHALL HAVE NO RIGHTS HEREUNDER FOR ANY SUCH APPLICATIONS.

Parts manufactured by vendors other than Echelon and referenced in this document have been described for illustrative purposes only, and may not have been tested by Echelon. It is the responsibility of the customer to determine the suitability of these parts for each application.

ECHELON MAKES AND YOU RECEIVE NO WARRANTIES OR CONDITIONS, EXPRESS, IMPLIED, STATUTORY OR IN ANY COMMUNICATION WITH YOU, AND ECHELON SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Echelon Corporation
www.echelon.com

Welcome

The Pyxos™ FT platform is Echelon's embedded control networking platform. The Pyxos FT platform includes two portable ANSI C APIs that you can use in conjunction with the Pyxos FT Interface Developer to create a Pyxos Pilot or a Pyxos Point that uses the Pyxos FT protocol to connect a controller to sensors and actuators in a variety of control networks. Each Pyxos FT network contains a *Pyxos Pilot*, which is the controller that manages the sensors and actuators on the network in a master-slave relationship. The sensors and actuators on the network are called *Pyxos Points*. Each Pyxos FT network contains a Pyxos Pilot, and up to 32 Pyxos Points.

The two APIs are used to implement the Pyxos Pilot application, and the hosted Pyxos Point applications. This includes the following:

- The **Pyxos FT Pilot API**. You can use the Pilot API to create a Pilot application to discover and configure Pyxos Points on the network, read their program types, monitor the reliability of the Pyxos FT network, and read or write Pyxos network variables (PNVs) of any type on up to 32 Pyxos Points at once via the Pyxos FT protocol. A Pilot application can detect Pyxos Point failures, and can also detect degraded communications due to poor network connections.
- The **Pyxos Point FT API**. You can use the Point API to create a hosted Pyxos Point, allowing you to send and receive updates to the Pyxos Pilot using PNVs.

This document describes how to create applications that use the Pyxos Point and Pilot APIs.

Related Documentation

The following manuals describe the Pyxos FT platform:

- *Introduction to the Pyxos Platform*. This manual introduces the concepts, technology, and tools for the Pyxos platform.
- *Pyxos FT Chip Data Book*. This manual provides hardware specifications for working with the Pyxos FT Chip.
- *Pyxos FT EVK User's Guide*. This manual describes the Pyxos FT EVK hardware, and how to use the hardware and software examples that are included with the Pyxos FT EVK. It also describes how to develop devices that incorporate Pyxos FT technology.
- *Pyxos FT EVK Quick Start Guide*. This guide helps you set up and install the Pyxos FT EVK Evaluation Boards, and how to get started with the Pyxos EVK examples and tools. A printed copy of this guide is included with your Pyxos EVK.

The following manual provides additional information that can help you to develop applications for the Pyxos FT platform:

- *Introduction to the LONWORKS System.* This manual provides an introduction to the ANSI/CEA-709.1 (EN14908) Control Networking

Protocol, and provides a high-level introduction to LONWORKS networks and the tools and components that are used for developing, installing, operating, and maintaining them.

After you install the Pyxos FT EVK software, you can view all of these documents from the Windows **Start** menu: point to **Programs → Echelon Pyxos FT EVK → Documentation,** and then click a document to view.

All of the Pyxos documentation, and related product documentation, is available in Adobe PDF format. To view the PDF files, you must have a current version of the Adobe® Reader®, which you can install from the Pyxos FT EVK CD or download from www.adobe.com/products/acrobat/readstep2.html. The English version of the Adobe Reader is included on the Pyxos EVK CD. Additional language versions are distributed by Adobe, and you can use those with the PDF files included with the Pyxos FT EVK.

System Requirements

Software and hardware requirements for computers running the Pyxos FT API development environment, also known as the Pyxos FT EVK, are listed below:

- Intel® Pentium® III 500MHz processor or AMD™ Athlon® 750 MHz processor
- Microsoft® Windows XP or Windows 2000, with all updates from Windows Update
- 128MB RAM minimum (256MB RAM recommended)
- 65MB of available hard-disk space (100MB recommended)
- 1024x768 screen resolution
- CD-ROM drive
- One available USB port

Table of Contents

1

Introduction to Pyxos Programming

This chapter introduces the Pyxos FT platform, and describes the relationship between a Pyxos Pilot application and a Pyxos Point application.

Introduction

A Pyxos FT network uses the Pyxos FT protocol to connect a controller to sensors and actuators in a control network. Each Pyxos FT network contains a *Pyxos Pilot*, which is the controller that manages the sensors and actuators on the network in a master-slave relationship. The sensors and actuators on the network that are connected to the Pyxos FT network are called *Pyxos Points*. Each network contains exactly one Pilot, and up to 32 Pyxos Points. A Pyxos Point may have a single sensor or actuator, or may have multiple sensors and actuators.

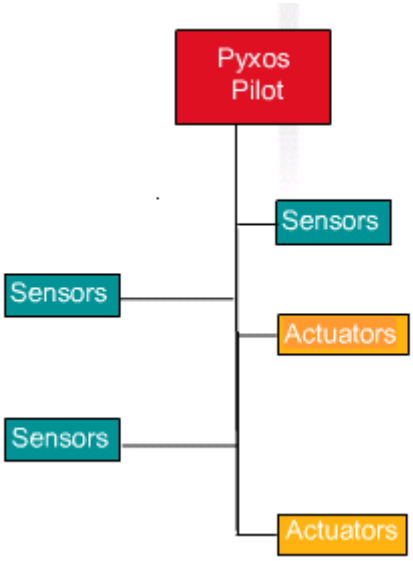

Figure 1 Pyxos FT Network Overview

The Pilot contains a host processor connected to a Pyxos FT Chip. The Pilot is responsible for configuring, maintaining and communicating with the Pyxos Points on the network. The Pilot is also responsible for receiving data from the network, acting on that data, and distributing it to the appropriate Pyxos Points. All communication on the network is either from a Pyxos Point to the Pilot, or from the Pilot to a Pyxos Point. As necessary, the Pilot application will relay data from one Pyxos Point to another. The data is relayed between the Pilot and the Pyxos Points in the form of *Pyxos network variables (PNVs)*, the smallest unit of sensor and actuator data that is shared on the network.

The Pilot can also communicate with devices on other proprietary networks, or with devices on LONWORKS networks. You can use the Pilot API to create a Pilot application that will determine exactly how these tasks will be performed.

The Pilot manages the Pyxos Points on the network in a master-slave relationship. There are two types of Pyxos Points: hosted Pyxos Points, and unhosted Pyxos Points. A hosted Pyxos Point is a device containing a host processor connected to a Pyxos FT Chip.

You can use the Point API to write an application that determines how a hosted Pyxos Point will function on the network. These applications should be designed to initialize and configure the Pyxos FT Chip on each device, as well as process

any I/O attached to the host processor, read PNVs updated by the Pilot, and write PNVs to send to the Pilot.

Unhosted Pyxos Points are slave devices containing Pyxos FT Chips that are not connected to a host processor. Figure 2 shows the architectural difference between an unhosted Pyxos Point and a hosted Pyxos Point.

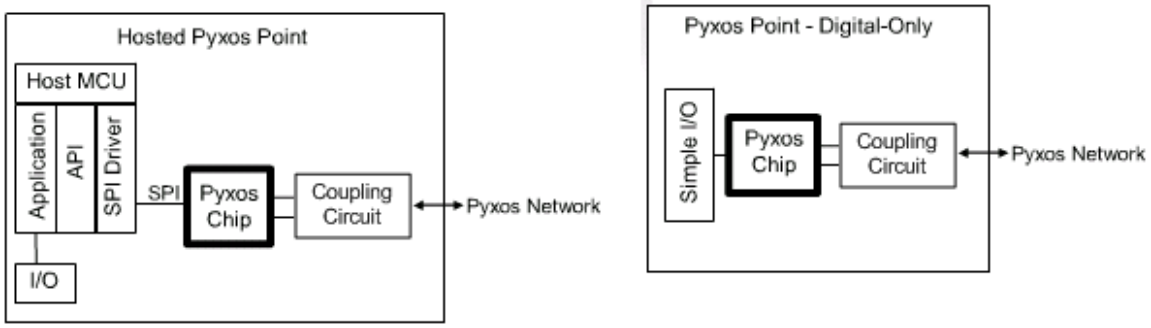

Figure 2 Hosted and Unhosted Pyxos Points

How Pyxos Points Interact With the Pilot

Each Pyxos Point implements its own interface. The interface defines the set of input and output PNVs that the Pyxos Point contains, as well as the location of those PNV values in the Pyxos FT Chip. Hosted Pyxos Points use input PNVs to receive information from the Pilot, and output PNVs to send information to the Pilot. Typically, a Pyxos Point will contain a PNV for each distinct I/O point on the device.

Unhosted Pyxos Points do not have their own set of input and output PNVs as hosted Points do. However, each unhosted Pyxos Point includes four pins, each of which can be used as either a digital input or a digital output. They also contain an optional fifth pin that can be used as a digital input. The Pilot API contains functions you can use to write directly to the digital outputs with your Pilot application. Since an unhosted Pyxos Point has no host, it will be lower cost than a hosted Point, and will consume less power. Therefore, an unhosted Point may be the most efficient way to control a device containing digital I/O.

Generally, you should implement your Pyxos Point as a hosted Point if the Point needs to support any analog data, the Point needs to support more than 4 digital outputs, more than 5 digital inputs, more than five total digital inputs and outputs, or the Point needs to perform any logic that is not performed by the hardware. For example, a hosted Point may transform values, or perform some local function based on inputs that may or may not be published on the Pyxos FT network. Hosted Points support initialization and configuration methods that require no user interaction, which may make using a hosted Point desirable even when using only digital I/O. This is described in more detail in the next section, *How Pyxos Points Join a Network*.

Figure 3 shows an example Pyxos FT network with two hosted Pyxos Points: one embedded inside a switch, and one embedded inside a lamp. In this example network, the Pyxos Point embedded inside a switch sends the Pilot application a message every time the switch is turned on or off by updating an output PNV of type **SNVT_switch**. The Pilot application then sends a message to the Pyxos

Point embedded inside the lamp by updating one of its input PNVs (also a **SNVT_switch**), so that the lamp is turned on or off, as appropriate.

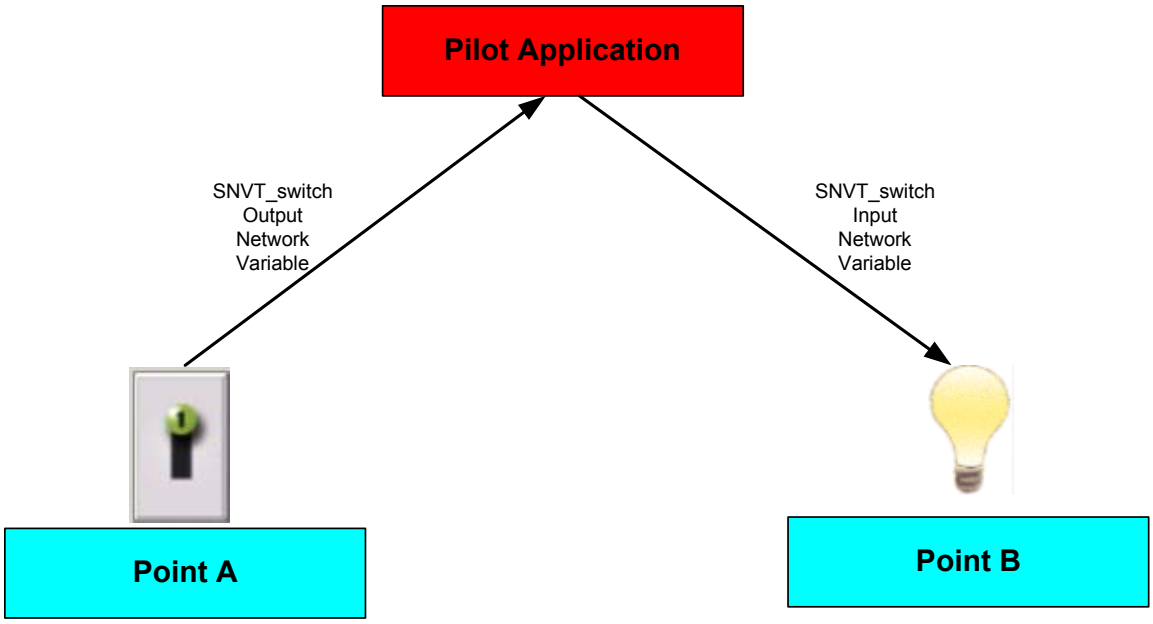

Figure 3 Pyxos Lamp Switch Example

You will use the Pyxos FT Interface Developer, which is included as part of the Pyxos FT EVK, to create Pyxos Point interfaces and define the set of PNVs each hosted Pyxos Point should contain (or the set of digital inputs and outputs that each unhosted Pyxos Point should contain). These tasks are described in Chapter 2 of this document. When you create an application for a Pyxos Point, you will identify the interface it should use when it joins the network. This is described in more detail in the following section.

How Pyxos Points Join a Network

Pyxos FT networks use time-division multiplexing to send and receive messages between the Pilot and the Pyxos Points. TDM is a type of digital multiplexing in which two or more simultaneous bit streams are encoded as sub-channels into a single bit stream by interleaving bits from the different bit streams. The combined bit stream is decoded at the receiving end. For a Pyxos FT network, the different sub-channels are the bit streams from the different Points on the Pyxos FT network, each communicating with the Pilot. Each of the sub-channels has a fixed bit rate, providing deterministic response for each of the sub-channels. The Pyxos FT Chips on the Pilot and the Points manage the interleaving of the channels.

The Pyxos FT protocol divides the time domain into several recurrent *timeslots* of fixed length, one for each sub-channel. A Pyxos *frame* consists of one window of time containing all the timeslots for all the sub-channels. Every frame includes 8 bytes of data sent by the Pilot to each Point, and 8 bytes of data from each Point to the Pilot.

Each Pyxos Point on a network is assigned a timeslot when it joins the network. The timeslot identifies the segment during which data (i.e. PNV updates or

segments of PNV updates) can be sent to that particular Point, and the Point can send data to the Pilot.

Figure 4 shows a representation of a network frame that contains four timeslots numbered 0 to 3. This network contains two Points labeled A and B, each of which has been allocated a timeslot. The Pilot sends messages to Point A in write timeslot 0, and receives messages from the Point in read timeslot 0. The Pilot sends messages to Point B in timeslot 1, and receives messages in read timeslot 1. There are two free timeslots in the frame (numbered 2 and 3) that are available for the next two Pyxos FT Points that join the network. Although the Pyxos FT Chip physically writes to a write timeslot and reads from a read timeslot, the Pilot application sees the write and read timeslots as a single timeslot that is assigned to the Point.

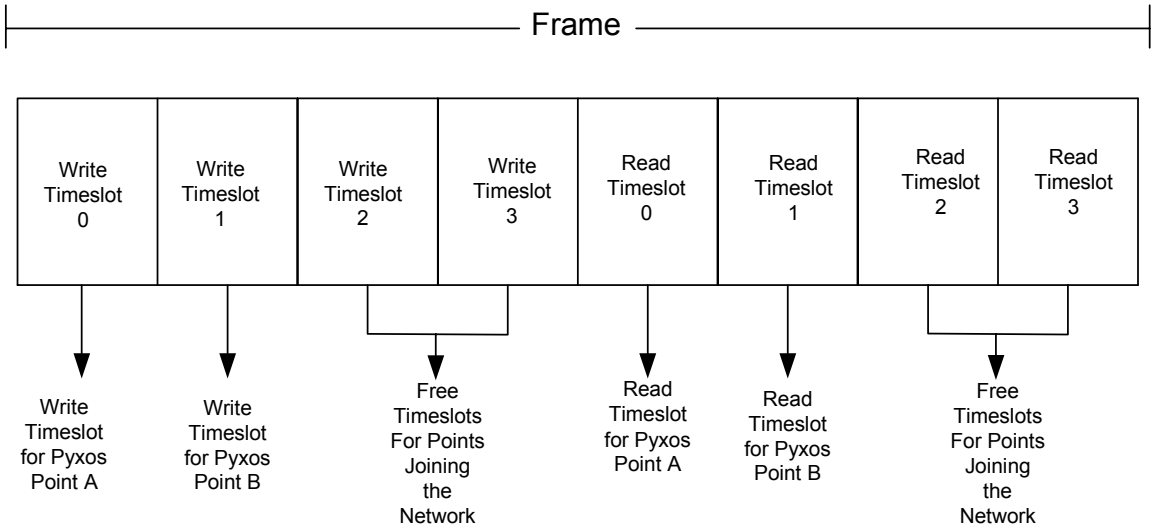

Figure 4 Timeslots

When a Point joins the network, the Pilot assigns the Point a timeslot. After assigning the Point a timeslot, the Pilot must specify the interface the Point will implement. The process of assigning a Pyxos Point a timeslot and specifying the interface it should implement when it joins the network is referred to as *registration.* After a Point has been successfully registered, the Pilot places the Point online so the Point can send or receive data.

There are three methods you can use to register a Pyxos Point. Depending on the level of user interaction your application will require, any of three registration methods may be suitable.

- *Automatic*. In the automatic registration method, the Pyxos Point begins requesting a timeslot as soon as it is initialized (i.e. as soon as the application loaded into the Pyxos Point starts running). It will continue making requests until the Pilot assigns it a timeslot.

  The Pyxos Point application does not need to perform any additional steps to receive a timeslot, as the Point API will begin requesting a timeslot automatically as soon as it is initialized. The Pilot application will typically read the Point's program ID, and identify its interface after allocating a timeslot. Therefore, no user interaction is required to complete an automatic registration.

- *Hardwired*. In the hardwired registration method, the timeslot and interface the Pyxos Point should use are hardcoded into the Pyxos Point and Pilot applications. After the Pyxos Point is initialized, its application selects a timeslot using data that is hardcoded into the application, read from a wiring harness or serial identification tag, or determined via some other application-dependent means. The Pilot application receives a registration request when this occurs, and then specifies the Pyxos Point's interface using hardcoded knowledge of what types of Pyxos Points use each timeslot.
- *Manual*. In the manual method, the user initiates the registration. This begins with the Pilot, where a user informs the Pilot application that a new Point will be registered. The way this step is performed will vary depending on the Pilot user interface. For example, it could be done by selecting from a menu on an LCD display or by pressing a sequence of buttons.

  The Pyxos Point application then sends a registration request to the Pilot. This is typically triggered by some user action on the Pyxos Point, such as pressing a button on the device containing the Point.

  The Pilot application then receives the registration request and allocates the Pyxos Point a timeslot, using the data entered previously to determine the timeslot to assign to the Pyxos Point and the interface implemented by the Pyxos Point.

  Unhosted Points must use the manual registration method.

Getting Started

You can use the Pyxos FT APIs to develop custom Pyxos Pilot or Pyxos Point applications. To develop a Pyxos application, follow the steps below. The steps are described in more detail later in the document:

1. Define the Pilot and Point interfaces using the Pyxos FT Interface Developer. In this step, you will specify how your Pilot application should interoperate with the Pyxos FT network. You will also create a Pyxos Point interface definition for each type of Pyxos Point on your network, and declare the PNVs that each type of Pyxos Point will contain.

   This step is described in Chapter 2, *Using the Pyxos FT Interface Developer*.

2. If you are developing any custom hosted Points, create applications for the custom hosted Pyxos Points on your network. For descriptions of the functions and callbacks included in the Point API, as well as guidelines and recommendations on how you should use those functions, see Chapter 3, *Creating a Pyxos Point Application*.

   Chapter 5, *Including the Pyxos FT API*, describes how to include the Point API in your application.

3. If you are developing a custom Pilot, create the Pilot application for your network. For descriptions of the functions and callbacks included in the Pilot API, as well as an overview of how you should use them with your Pilot application, see Chapter 4, *Creating a Pilot Application*.

   Chapter 5, *Including the Pyxos FT API*, also describes how to include the Pilot API in your application.

4. Load the custom Pyxos Pilot and Point applications into the host processors attached to the Pyxos FT Chips on the network, and begin operating the network.

Several additional chapters you may find useful follow Chapter 5. Table 1 describes these chapters.

Table 1 Pyxos Programmer's Guide Supplemental Chapters

Title	Description
Chapter 6, *Porting the Pyxos FT API*	This chapter describes how to port the Pyxos FT API to a new processor or compiler. This includes hardware considerations and details on the Pyxos Serial API (psAPI), and on modifications you can make to the **platform.h** file to suit your microprocessor.
Chapter 7, *Pyxos FT Protocol*	This chapter describes the Pyxos FT Protocol and how to directly access the Pyxos FT Chip to use the protocol. The simplest way to develop a Pyxos Point or Pyxos Pilot application is to use the APIs provided with the Pyxos software. These APIs implement most of the required host code for the Pyxos FT Protocol.
Appendix A, *Advanced Topics*	This chapter discusses advanced topics, including descriptions of the functions you can use to read and write registers that control the operation of a Pyxos FT Chip, and discussion on the data space and code size consumed by a Pyxos application.
Appendix B, *Pyxos FT Network Gateways*	This appendix provides background information that you can use when extending your Pyxos FT network to connect to other Pyxos FT networks, or to a LONWORKS network.

2

Using the Pyxos FT Interface Developer

This chapter describes how to install the Pyxos FT EVK software, and how to use the Pyxos FT Interface Developer to design your Pyxos FT network.

Installing the Pyxos Software

Follow the steps below to install the Pyxos EVK software. Once you have installed the software, you can use the Pyxos FT Interface Developer to design your Pyxos FT network project, as described in the *Using the Pyxos FT Interface Developer* section on page 10.

1. Log on to your Windows computer with a user ID that is a member of the Administrators group or that has equivalent administrator privileges.
2. Insert the Pyxos FT EVK CD-ROM into a CD-ROM or DVD drive. The setup application should start automatically. If it does not, use Windows Explorer to open the CD-ROM drive, and double-click **Setup**. The Pyxos EVK setup application's main window opens.
3. From the Pyxos EVK setup application's main window, click **Install Products**. The Install Products window opens.
4. From the Install Products window, click **Pyxos FT EVK 1.0**. Follow the installation dialogs to install the Pyxos FT EVK software onto your computer.
5. If you do not already have the Microsoft .NET 2.0 (or later) runtime components on your computer, return to the Pyxos EVK Install Products window and click **Microsoft .NET Framework 2.0** to install the Microsoft .NET Framework. The .NET runtime components are required to run the Pyxos Network Example HMI application.
6. If prompted to restart Windows, click **OK** to restart.
7. If you do not already have the Adobe Reader installed on your computer, return to the Pyxos EVK setup application's Install Products window and click **Adobe Reader 7.0.8** to install the Adobe Reader. The Adobe Reader is required to view the Pyxos product documentation. You can also download the latest reader from the Get Adobe Reader link at www.adobe.com. The Adobe Reader included with the Pyxos EVK is the English version. You can get other language versions from www.adobe.com.
8. If you plan to connect the Pyxos FT network to a LONWORKS network, you can install the LONMARK Resource Editor 3.13 and the OpenLDV™ 3.3 driver from the Pyxos EVK setup application's Install Products window. These products are optional. You can also download the latest version of the OpenLDV 3.3 driver from www.echelon.com/downloads.

Using the Pyxos FT Interface Developer

You can use the Pyxos FT Interface Developer to configure the Pilot application for your Pyxos FT network project, and to create Point interface definitions for the Pyxos Points in your project. To do so, follow these steps:

1. Create a network project. For more information on this task, see *Step One: Creating a Network Project* on page 11.

2. Create or import the Point interface definitions for the Pyxos Points in your project. Create (or import) a separate Point interface definition for each type of Pyxos Point that exists in your project.

   When you create a Point interface definition for a hosted Point, you will define the PNVs that the Points implementing that interface will contain. When you create a Point interface definition for an unhosted Point, you will define the I/O bitmasks (used to read and write the digital inputs and digital outputs on the Pyxos FT Chip) that the Points implementing the interface will contain. For hosted Points, you will also select the registration method that Pyxos Points using the definition will use. For more information on this task, see *Step Two: Creating and Importing Pyxos Point Interface* on page 13. Unhosted Points always use the manual registration method.

3. Set advanced options for the Pyxos FT network project, including the network project settings and the Pilot options. For more information on this task, see *Step Three: Setting Advanced Options* on page 22.

4. Generate the include files for the network project you have created. These include files are used to customize the Pyxos Pilot and Point APIs. You will include these files when you create your Pyxos Point and Pilot applications.

   For more information on this task, see *Step Four: Generating Include Files* on page 29.

5. Begin developing the Pilot and Pyxos Point applications for your network project, as described in Chapters 3 and 4 of this document.

Step One: Creating a Network Project

The first step to perform when using the Pyxos FT Interface Developer is to create a network project. To do so, follow these steps:

1. Click the **Start** button, point to **Programs** → **Echelon Pyxos FT EVK** and then click **Pyxos FT Interface Developer**. The Pyxos FT Interface Developer – Startup Options dialog opens.

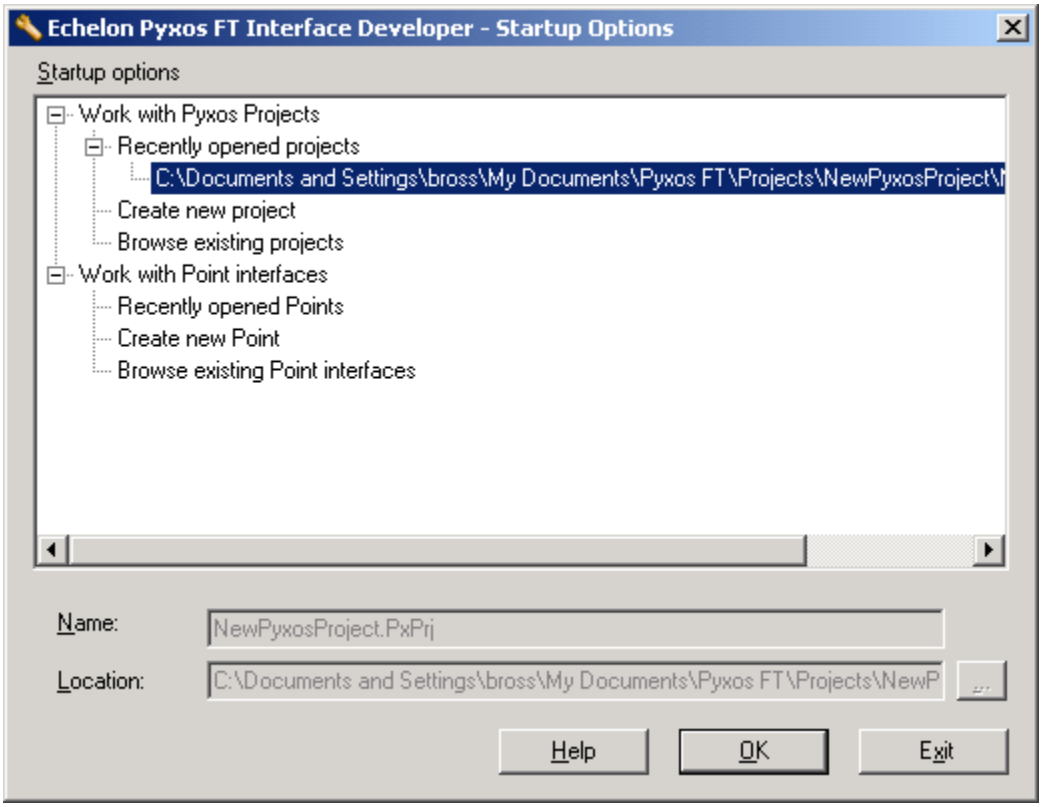

Figure 5 Pyxos FT Interface Developer – Startup Options

2. Click **Create New Project** to start a new Pyxos FT network project, and then enter a name for the project in **Name**. Specify the location where the project files are to be stored in **Location**, and then click **OK** to create the project. The window shown in Figure 6 opens.

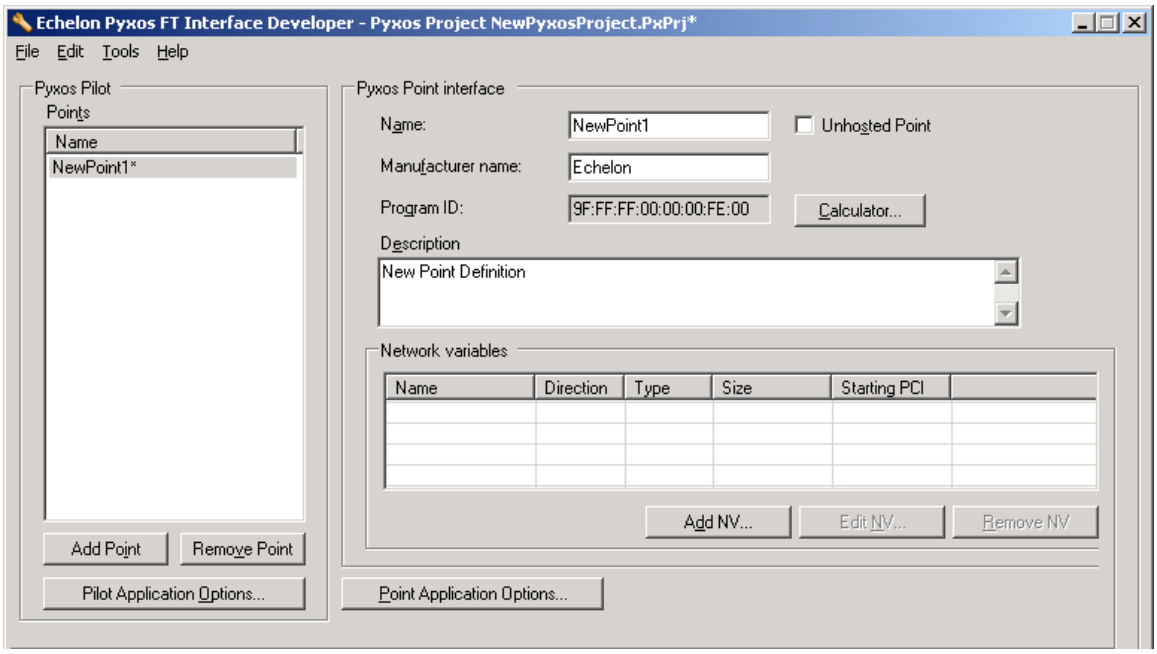

Figure 6 Pyxos FT Interface Developer Window

3. Create or import the Point interface definitions for your project, as described in the next section.

Step Two: Creating and Importing Pyxos Point Interface Definitions

After you have created a network project, you can begin creating or importing the Point interface definitions for the project. Each Pyxos Point will use a Point interface definition. The Point interface definition determines how the Pyxos Point should be registered to join a network, and what PNVs the Pyxos Point should contain.

To create a Point interface definition, follow the steps below. You can also import Point interface definitions created for another Pyxos FT network project into your project, as described in the *Importing Point Interface Definitions* section on page 19.

1. From the **File** menu, point to **New** and then click **Point Interface Definition** (or, click **Add Point**). You will be prompted to save the project files you have created, and then you will return to the Pyxos FT Interface Developer – Startup Options dialog shown in Figure 5.

   Enter the name of the Point interface definition in **Name** and the location of the project files for the Point interface definition in **Location**, and then click **OK**. The name must be between 1 and 20 characters long. You can use alphanumeric characters, as well as the underscore character (_), in the name.

2. Select the **Unhosted Point** check box to implement the Points that use this interface definition as unhosted Points. Otherwise, leave the check box clear.

If you are implementing the Point interface definition for unhosted Points, proceed to step 4.

3. If you are implementing the Point interface definition for hosted Pyxos Points, meaning that you left the **Unhosted Point** check box clear in step 2, click **Point Application Options** to open the Point Application Options dialog.

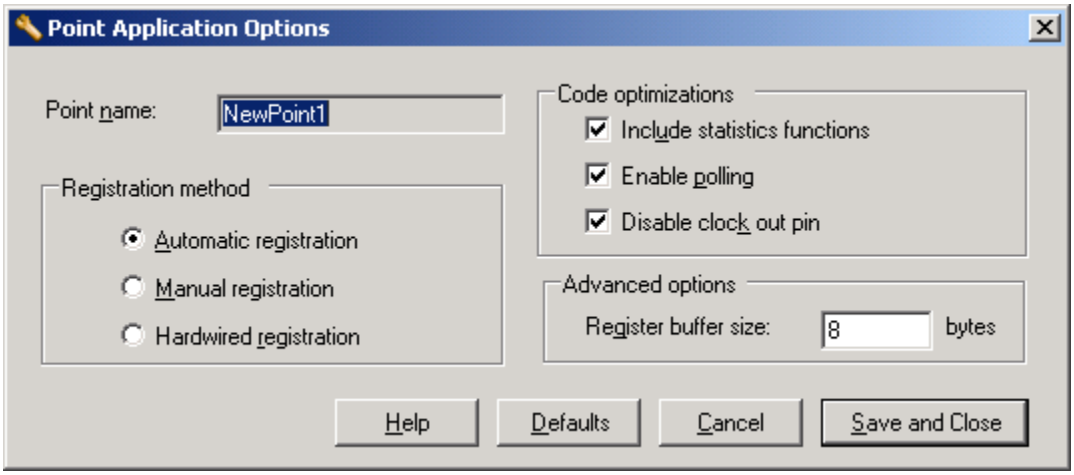

Figure 7 Point Application Options Dialog

Set the properties on the Point Application Options dialog. Table 2 describes these properties.

Table 2 Pyxos Point Application Options Dialog

Property	Description
Registration Method	Select **Automatic Registration**, **Manual Registration**, or **Hardwired Registration** to specify how Pyxos Points that use this interface definition should be registered to join a network. Chapter 1 introduces these registration methods, and Chapter 3 provides details on how your Point application will perform the registration when using each method.
Include Statistics Functions	Select this check box to enable the Pyxos Point application to access functions used to read network statistics such as the total number of frames received on the Pyxos Point's timeslot, and the number of CRC errors on the timeslot.
Enable Polling	Select this check box to enable the Pyxos Point application to request that the Pilot update one of the Point's input variables.

Property	Description
Disable Clock Out Pin	Every Pyxos FT Chip has a clock out pin that the Point can use to drive the host processor's clock, or anything else that might need a clock. If the Point's hardware uses the clock, the clock out pin must be enabled. If the Point's hardware does not use the clock, it can be disabled to conserve power. On a hosted Point and on the Pilot, the clock out pin will be disabled by the API based on the application options settings set on the Point Application Options and Pilot Applications Options dialogs, respectively. The Pilot disables/enables the clock out pin on unhosted Points, based on the setting of the **Clock Out** field on the unhosted Point's interface. Select the **Disable Clock Out Pin** check box to disable the clock out pin on the Point.
Register Buffer Size	Enter the size of the Pyxos Point's register buffer, in bytes. This buffer is used when performing any register operations during a callback. If the application asks to read or write a local register larger than this value during a callback, the function will fail and an error will be returned. The default value is 8.

4. Click **Save and Close** to save your settings and return to the main Point Interface Developer window. You can return to the Point Application Options window at any time to review and change your settings.

5. Configure the rest of the properties on the window. The properties you need to configure will vary depending on whether the Point interface definition is for an unhosted Point, or for a hosted Point.

   Table 3 lists the properties required for unhosted Pyxos Points.

Table 3 Unhosted Pyxos Point Properties

Property	Description
Clock Out	Select **Enabled** to enable the clock out pin on the unhosted Point.
I/O Mode	Select **Polled** to specify that the Pyxos Point only send updates for its digital inputs onto the network when it is polled by the Pilot application. Select **Send Updates On Change** to enable the Pyxos Point to both respond to polls, and to send updates on the network whenever the values of the digital inputs change.

Table 4 lists the properties required for hosted Pyxos Points.

Table 4 Hosted Pyxos Point Properties

Property	Description
Program ID	Click **Calculator** to open the LonMark Standard Program ID Calculator. Use the Program ID Calculator to set the program ID that will be assigned to Pyxos Points using this interface definition. You can set the program ID by manually entering it in the Program ID box at the bottom of the dialog, or you can set the other fields on the dialog to appropriate values for your application, and calculate a program ID based on those values. For information on how you should set these fields, see *Using the Standard Program ID Calculator* on page 19.

Table 5 lists the properties required for both unhosted Pyxos Points and hosted Pyxos Points.

Table 5 Properties for Unhosted and Hosted Pyxos Points

Property	Description
Manufacturer Name	Optionally, enter the name of the manufacturer (or an abbreviation of the name) of the device containing the Pyxos Points that use this interface definition in the **Manufacturer Name** box. The name must be between 1 and 10 characters long. You can use alphanumeric characters, as well as the underscore character (_), in the name. This field may also be useful if you plan on taking advantage of the interoperability provided by Pyxos FT networks. For example, multiple manufacturers may produce interfaces for Pyxos Points for use on a single network. In that case, you can use this field to identify the manufacturer of the device containing each Pyxos Point and to avoid naming conflicts in the generated header files.
Description	Enter a description of the interface definition in the **Description** box.

6. Add PNVs to the interface definition for hosted Points, or I/O bitmasks for unhosted Points. If you are creating a hosted Point interface definition, click **Add NV** to open the Add Pyxos NV dialog. If you are creating an unhosted Point interface definition, click **Add Bitmask** to open the Add I/O Bitmask for Unhosted Point dialog.

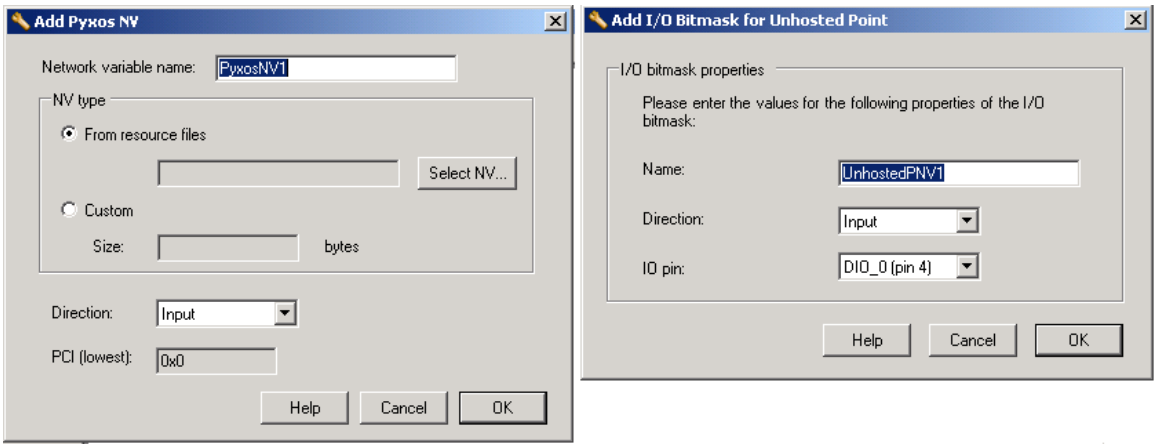

Figure 8 Add Pyxos NV and Add I/O Bitmask for Unhosted Point Dialogs

7. If you are adding an I/O bitmask to an unhosted Pyxos Point, configure the fields on the Add I/O Bitmask for Unhosted Point dialog, and then click **OK** to return to the main Pyxos FT Interface Developer window.

   When you return to the main Pyxos FT Interface Developer window, the I/O Bitmasks list will include the I/O bitmask you've just created. You can edit its configuration later by selecting it in the list, and then clicking **Edit Bitmask** (you will not be able to change its direction). You can remove it by selecting it in the list and clicking **Remove Bitmask**. You can add more I/O bitmasks by clicking **Add Bitmask** again. When you have finished, proceed to step 11.

   Table 6 describes the fields on the Add I/O Bitmask for Unhosted Point dialog.

Table 6 Add I/O Bitmask for Unhosted Point Dialog

Property	Description
Name	Enter the name of the I/O bitmask.
Direction	Specify whether the I/O pin will be used as an input or output. An input pin should be connected to a digital input, and an output pin should be connected to a digital output.
IO Pin	Select the IO pin that this I/O bitmask applies to in **IO Pin**. The bitmask selected for the **DI (pin 3)** I/O pin must be an input.

   The Pyxos FT Chip reserves 4 I/O pins for digital I/O. If one or more of these pins is not connected to either an input or output, you do not have to define an I/O bitmask for that pin. The interface generated by the Pyxos FT Interface Developer will tell the Pilot to configure the pin as an output. If the pin is tied high or low, you must configure the pin as an input. To accomplish this, you can define dummy bitmasks for these unused pins that are defined as inputs. Assign bitmask names and a Point description that make it clear that these pins are not used by the Point.

8. If you are adding a PNV to a hosted Pyxos Point interface definition, enter the name of the PNV in **Network Variable Name**.

   Then, select the PNV's type by setting **NV Type**. To assign a custom type to the PNV, select **Custom**, enter the size of the PNV in **Bytes**, and then click **OK**.

   To select a type from the resource file catalogs on your computer, select **From Resource Files**, and then click **Select NV**. The window shown in Figure 9 opens.

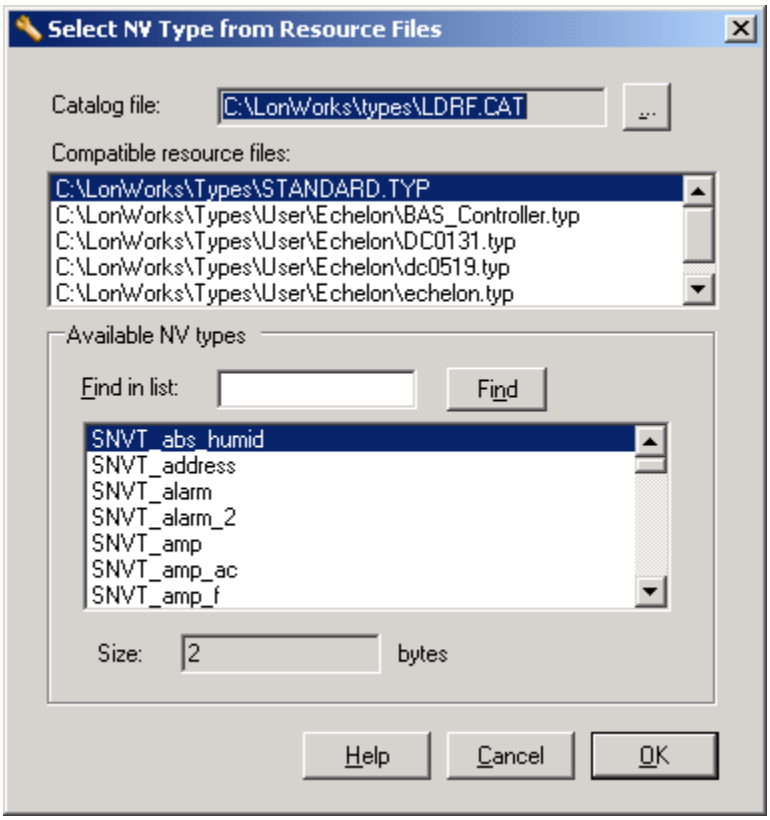

Figure 9 Select NV Type From Resource Files Dialog

By default, the window will list the resource files available in the standard LONWORKS Resource Files catalog on your computer. You can enter the name of a type in the **Find in List** box, and then click **Find** to scroll to that type's entry in the list. You can click the button to the right of **Catalog File** to choose a different resource file catalog.

If a suitable standard type is not available, you can choose a resource file containing user-defined types. Using standard types simplifies integration of Points and Pilots by different developers or manufacturers.

You can use the NodeBuilder Resource Editor to browse the resource files and edit the types available on your computer. To start the NodeBuilder Resource Editor, select **NodeBuilder Resource Editor** from the **Tools** menu.

Select a resource file in **Compatible Resource Files**. Then, select the PNV type in **Available NV Types**. Click **OK** to return to the **Add NV** dialog.

9. Set **Direction** to specify whether the PNV is an **Input** or **Output** network variable. An input PNV is used by the Pyxos Point to receive information from the Pilot. An output PNV is used to send information to the Pilot.

9. Click **OK** to return to the main Pyxos FT Interface Developer window. The newly created PNV will be listed in **Network Variables** at the bottom of the dialog. You can edit the PNV's configuration later by selecting it in the list and clicking **Edit NV** (you will not be able to change the PNV's direction). Or, you can remove it by selecting it in the list and clicking **Remove NV.**

10. Repeat steps 6-9 to create more PNVs, as your application design requires.

11. From the **File** menu, click **Save NewPoint.PxIntf**, and then specify a file name for the Point interface definition. You can use the default directory selected by the Pyxos FT Interface Developer to store the file.

12. The Point interface definition will now be listed in **Points** on the Pyxos FT Interface Developer window. You can modify its settings later. To do so, re-open the network project by selecting **Open > Network Project** from the **File** menu, and then select **Open > Point Interface Definition** to edit the Point interface definition.

    You can remove the Point interface definition by selecting it in the list, and clicking **Remove Point**. If you create a new Point interface definition and then try to remove it after making any changes to it, a dialog will open asking if you want to save the changes to the Point. This behavior is expected, as removing the Point interface definition does not cause its underlying files to be deleted.

Importing Point Interface Definitions

You can import Point interface definitions created for another Pyxos FT network project into your project. This may be useful if you plan to add devices created by other manufacturers to your project. To do so, follow these steps:

1. Add the **.PxIntf** file for the Point interface to a local directory on the computer running the Pyxos FT Interface Developer.

2. From the **File** menu, click **Add Existing Point**. The **Add Point Interface Definition File to Project** window opens.

3. Browse to the **.PxIntf** file selected in step 1, and click **Open**.

4. The imported Point interface definition will be listed in **Points** on the right side of the Pyxos FT Interface Developer window. Apply the interface definition to the Pyxos Points in your project. The settings selected on the Point Application Options dialog for the interface may no longer be valid, as the settings are not imported.

Using the Standard Program ID Calculator

The program ID is a 64-bit (16-hex-digit) identifier that uniquely identifies the application contained within a hosted Pyxos Point. A program ID is typically

presented as eight pairs of hexadecimal encoded digits, separated by colons. When formatted as a standard program ID, the 16 hexadecimal digits are organized as 6 fields that identify the manufacturer, classification, usage, channel type, and model number of the device. Every standard program ID uses the following format:

FM:*MM*:*MM*:*CC*:*CC*:*UU*:*TT*:*NN*

You will use the Standard Program ID Calculator shown in Figure 10 to select a program ID for each hosted Pyxos Point interface definition in your project. You can set the program ID by manually entering it in the Program ID box at the bottom of the dialog, or you can set the other boxes on the dialog to appropriate values for your application, and calculate a program ID based on those values. Table 7 lists and describes the program ID fields. When you have defined your program ID, click **OK** to return to the main Pyxos FT Interface Developer window.

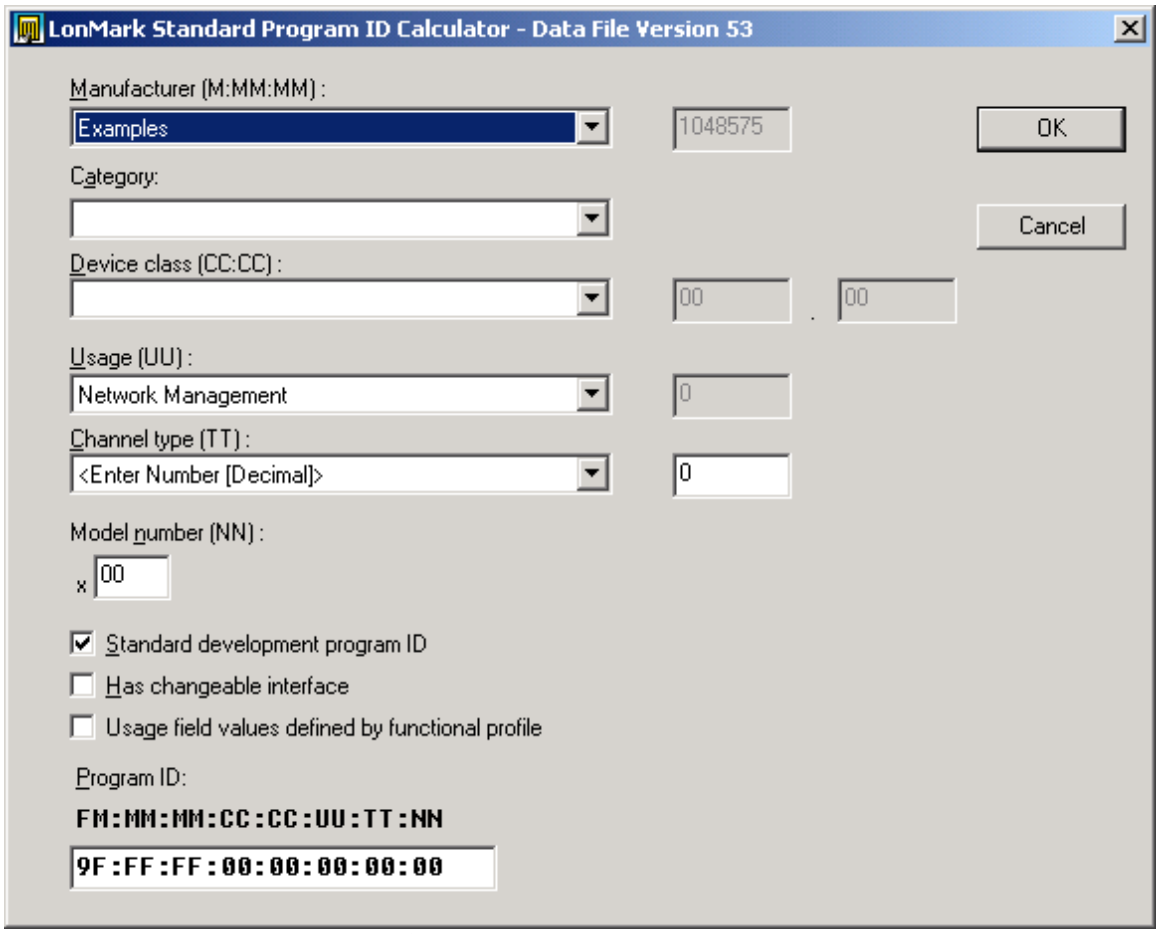

Figure 10 LonMark Standard Program ID Calculator

Table 7 Program ID Fields

Program ID Segment	Field	Description
F	**N/A**	A 4-bit format identifier. Must be set to 9.
M:MM:MM	**Manufacturer**	A 20-bit identifier for the device manufacturer. Click the arrow to select from a list of all the device manufacturers who are members of LONMARK International. If your company is a member of LONMARK International but is not included in the list, download the latest program ID data from www.lonmark.org/spid. If your company is not a member of LONMARK International, get a temporary manufacturer ID from www.lonmark.org/mid. If your company is a LONMARK member, but not listed in the updated program ID list, or if you have a temporary manufacturer ID, select **<Enter Number [Decimal]>** in the **Manufacturer** list, then enter your manufacturer ID in the field to the right of the **Manufacturer** box. Enter the value in decimal, the calculator converts it to hex for the program ID. You do not have to join LONMARK International to get a temporary manufacturer ID, the information required to get one is very minimal, and there is no fee to get one. For more information about membership in LONMARK International, see www.lonmark.org.
CC	**Category**	The general purpose or industry of the device. The **Category** selected determines the device classes that will be available in **Device Class**. Select **ALL** to have **Device Class** show all existing device classes. Select **Profiles By Name** to have **Device Class** show an alphabetical list of all device classes with a standard functional profile. Select **Profiles By Number** to have **Device Class** show a numerical list (sorted by device class number) of all device classes with a standard functional profile.
CC	**Device Class**	A 16-bit identifier for the primary function of the device. The primary function of the device is determined by the primary function implemented by your device.
UU	**Usage**	Select the setting that most appropriately describes the usage of your device. You can enter any value in this field if you set the **Usage Field Values Defined By Functional Profile** check box.
TT	**Channel Type**	Must be set to **Pyxos-Ft** (0xFE)

Program ID Segment	Field	Description
NN	**Model Number**	An 8-bit identifier that you assign to specify the product model for your device. Assign a unique model number for the specified manufacturer, device class, usage, and channel type. You can use the same hardware for multiple model numbers, depending on the program that is loaded into the hardware. The model number within the program ID does not have to conform to your published product model number.
N/A	**Standard Development Program ID**	This check box must be selected—this chooses a format 9 standard program ID.
N/A	**Has Changeable Interface**	This check box must be cleared.
N/A	**Usage Field Values Defined By Functional Profile**	Select this check box to enter any value in the **Usage** field.
N/A	**Program ID**	This box is automatically updated when changes are made to the other fields on the dialog. You can also manually enter or change a program ID here.

Step Three: Setting Advanced Options

The default settings included with the Pyxos FT Interface Developer utility will be suitable for most applications. This section describes additional options for advanced developers.

Setting the Pilot Application Options

To set the Pilot Application Options, follow these steps:

1. Open the Pyxos FT network project as described previously in this chapter, and then click **Pilot Application Options**. The Pilot Application Options dialog opens.

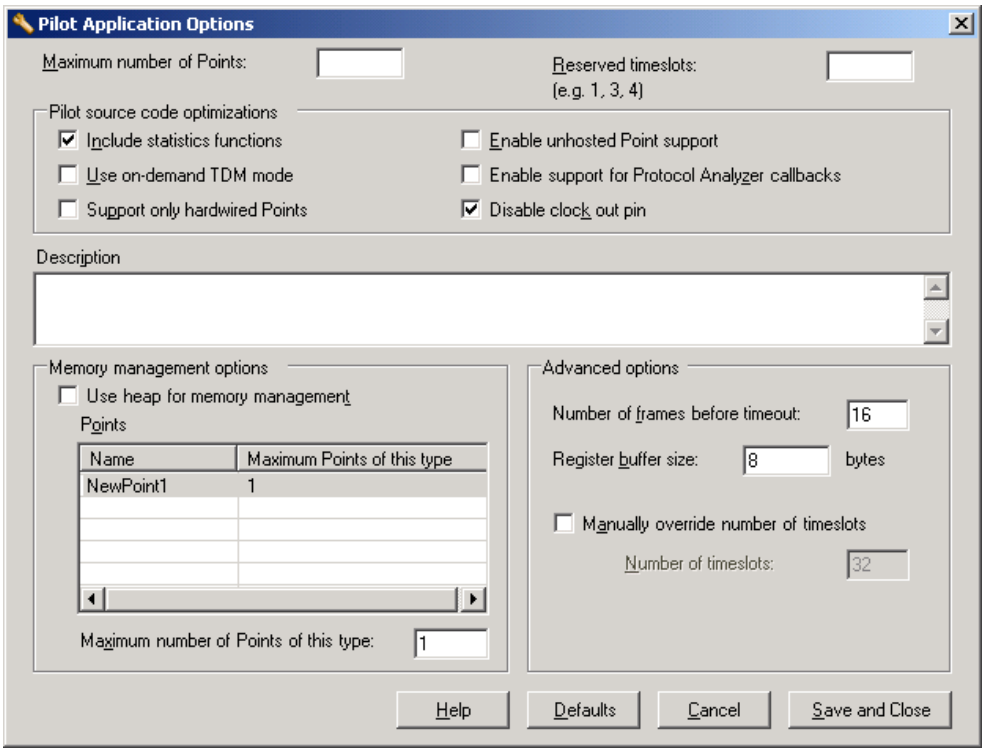

Figure 11 Pilot Application Options Dialog

2. Set the fields on the Pilot Application Options dialog, and then click **Save and Close** to save your settings. Table 8 describes the fields on the Pilot Application Options dialog.

Table 8 Pilot Application Options

Field	Description
Maximum Number of Points	Enter the maximum number of Pyxos Points that can be added to the network. Each network must contain at least one Pyxos Point, and can contain a maximum of 32 Pyxos Points. This setting defines the number of timeslots for the network, and therefore affects the response time for the network. By default, the number of timeslots is set to the minimum even number that is greater than or equal to this value. For a network with two timeslots (two write plus two read timeslots), that is, up to two Points, response time is less than 2 ms; for a network with a full 32 timeslots (32 write plus 32 read timeslots for 32 Points), response time is about 25 ms. The default value is 32.
Reserved Timeslots	Enter any timeslots that should be marked as reserved in a comma-separated list. The Pilot application will not announce timeslots that are marked as reserved, so that Pyxos Points that are registered with the manual and automatic registration methods will not have access to these timeslots. If you are using the hardwired registration for any Pyxos Points on your network, you should reserve the timeslots you plan to assign to those Pyxos Points. For example, to mark the first five timeslots as reserved, enter **0,1,2,3,4** By default, no reserved timeslots are selected.
Include Statistics Functions	Select this check box to enable the Pilot application to read and clear network statistics for the Pyxos Points on the network with the **PyxosReadNetworkStats()** and **PyxosClearNetworkStats()** functions. See Chapter 4, *Creating a Pilot Application*, for more information on these functions. By default, this check box is selected.
Enable Unhosted Point Support	Select this check box if you will be adding any unhosted Pyxos Points to the network. By default, this check box is cleared.
Use On-Demand TDM Mode	Select this check box to place the network in on-demand mode, meaning that write frames used to update the PNVs on the network will only be sent when the **PyxosPilotEventHandler()** function is called. By default, the Pilot runs in continuous TDM mode, rather than in on-demand TDM mode. You can use on-demand mode to save power on very slow networks. In on-demand mode, the Pilot application controls how frequently frames are sent. By default, this check box is cleared.

Field	Description
Support Only Hardwired Points	Select this check box if all of the Pyxos Points on the network are to join the network using the hardwired registration method. If this is selected, all functions that can be used to allocate free timeslots and send free timeslots to a Pyxos Point will be excluded from the Pilot API. By default, this check box is cleared.
Enable Support for Protocol Analyzer Callbacks	When debugging a Pyxos application, it may be useful to implement a protocol analyzer that logs the packets sent on the Pyxos FT network. You may use the **PyxosPilotProtocolAnalyzerCallback()** to aid in this development. Select the **Enable Support for Protocol Analyzer Callbacks** check box to enable the use of this feature. For more information on the **PyxosPilotProtocolAnalyzerCallback()**, see *Analyzing Pyxos FT Network Communication* on page 92. By default, this check box is cleared.
Disable Clock Out Pin	Every Pyxos chip has a clock out pin that the Point can use to drive the host processor's clock, or anything else that might need a clock. If the Point's hardware uses the clock, the clock pin must be enabled. If the Point's hardware does not use the clock, it can be disabled to conserve power. On a hosted Point and on the Pilot, the clock out pin will be disabled by the API based on the application options settings set on the Point Application Options and Pilot Applications Options dialogs, respectively. The Pilot disables/enables the clock out pin on unhosted Points, based on the setting of the **Clock Out** field on the unhosted Point's interface. Select the **Disable Clock Out Pin** check box to disable the clock out pin on the Pilot. By default, this check box is selected.
Description	Enter an optional description of the Pilot.
Manually Override Number of Timeslots	If you want to specify how many timeslots can be allocated on the network, select the **Manually Override Number of Timeslots** check box and then enter the number of timeslots in the **Number of Timeslots** field. This must be an even number between 2 and 32, and must be greater or equal to than the **Maximum Number of Points** value. **Use the default value for this property.** By default, this check box is not selected, and the **Number of Timeslots** value is set to the minimum even number that is greater than or equal to the value of the **Maximum Number of Points** property.

Field	Description
Number of Frames Before Timeout	Enter the number of read frames that the Pilot application will wait for an acknowledgment before timing out after sending an update or poll request to a Point. The default value for this property is 16 frames. You should only increase this value if you have some Pyxos Points on the network that are very slow. You can decrease this value if you have a large network and you want very fast failure responses. You must keep the value of this property in the range of 2-254.
Register Buffer Size	Enter the size of the register buffer, in bytes. This buffer is used when performing any register operations during a callback. If the application attempts to read or write a local register larger than this value during a callback, the function will fail and an error will be returned. **Use the default value (8 bytes) for this property.**
Memory Management Settings	Select the **Use Heap For Memory Management** check box to specify that the Pyxos FT API will use the heap to allocate the data used to control the Pyxos Points on the network. Depending on the flexibility of the network, using the heap may or may not result in better use of resources. If you do not select the **Use Heap For Memory Management** check box, then data must be declared to support each instance of each Point and you must specify the maximum number of each type of Point supported. You can do this by selecting a Pyxos Point interface definition in **Points**, and then entering the maximum number of Pyxos Points that will use that interface definition in **Max Number of Points of This Type**.

Setting the Network Project Settings

To set the project settings for your Pyxos FT network project, follow these steps:

1. Create (or open) the project as described in the previous sections.
2. From the **File** menu, select **Project Settings**. The Network Project Settings window opens.

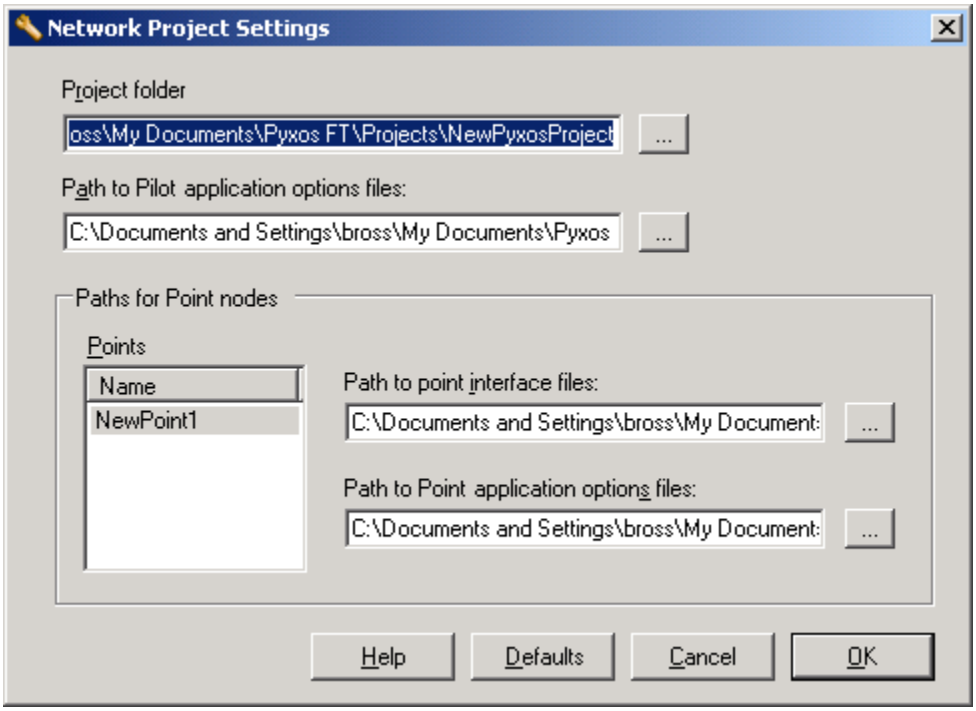

Figure 12 Network Project Settings

3. Fill in the fields on the Network Project Settings dialog. These fields determine where the include files for your network project will be stored. You will reference these include files when creating the Pyxos Point and Pilot applications for your network.

   Table 9 describes these settings.

Table 9 Network Project Settings

Property	Description
Project Folder	Select the folder that will contain all of the project files for your Pyxos FT network project.
Path to Pilot Application Options Files	Select the file path for the Pilot application options file for your network project. This file defines the operating parameters that affect how the Pilot will interoperate with and manage the Pyxos FT network.

Property		Description
Paths for Point Nodes	**Path to Point Interface Files**	The name and path of the interface definition file for the Pyxos Point type currently selected in **Points**. The interface definition file contains definitions of the PNVs that Pyxos Points using the interface definition file should include.
	Path to Point Application Options Files	The name and path of the Point application options file for the Pyxos Point type currently selected in **Points**. This file references the Point interface definition file, and contains information each hosted Pyxos Point application will need. If a Point implementation does not use the API, no Point Application Options file is required. While the Point Application Options button in the Pyxos FT Interface Developer will be enabled for such a Point, the settings displayed in the corresponding Point Application Options dialog may not be representative of the actual settings for the Point.

4. Click **OK** to save your changes and return to the main Pyxos FT Interface Developer window.

Setting Project Options

You can use the Options dialog to configure project settings for the Pyxos FT Interface Developer. To use the Options dialog, follow these steps:

1. Create (or open) the Pyxos FT network project as described in the previous section.
2. From the **Tools** menu, select **Options**. The Options dialog opens.

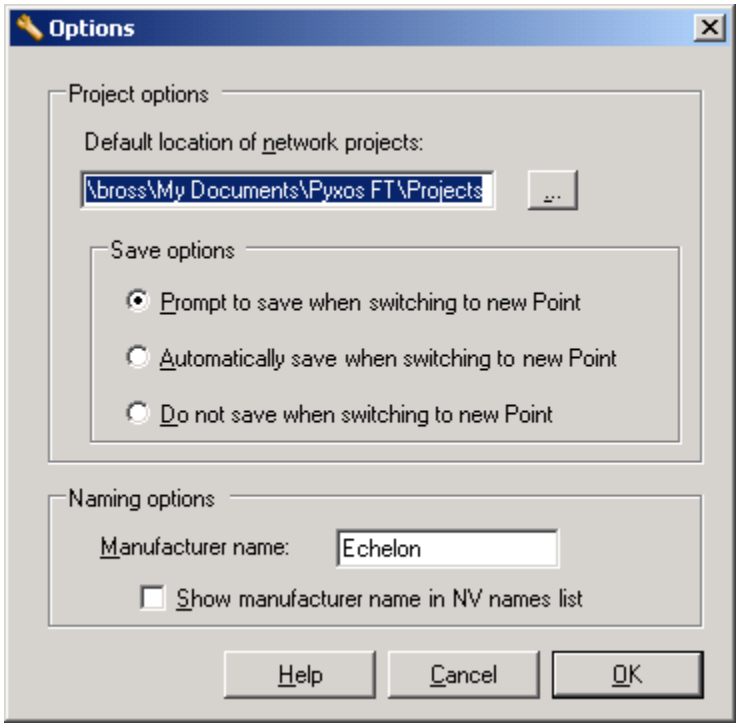

Figure 13 Options Dialog

3. Configure the fields on the Options dialog, and then click **OK** to save your changes. Table 10 describes the fields on the Options dialog.

Table 10 Options Dialog

Property	Description
Default Location of Network Projects	Sets the default location of the network projects. You can change the location when you create a Pyxos FT network, as described in the *Step One: Creating a Network Project* section on page 11.
Save Options	Determines whether or not you will be prompted to save the interface definition you are editing when you choose to create a new interface definition.
Naming Options	Specifies the manufacturer name to be applied to your Pyxos Points in the **Manufacturer Name** box. Select the **Show Manufacturer Name in NV Names List** check box to display the manufacturer name in the list of PNVs on the main Pyxos FT Interface Developer window.

Step Four: Generating Include Files

When you have finished designing your network project, you can generate the include files for the project and for each Pyxos Point, and then begin developing your applications. The following sections provide an overview of the files you will generate. This is followed by instructions you can use to generate the include files.

Generated XML Files

You will generate a Point interface definition file for each type of Pyxos Point in your project. These files use the following naming convention: **<*PointName*>.PxIntf**, where **<*PointName*>** represents the name of the Pyxos Point. The Point interface file name depends on the name of the Point, so the only way to change the name of the file is by changing the name of the underlying Point. When you create a Pyxos Point application, you will reference this file to identify the interface the Pyxos Point should implement.

You will also generate a **MyPyxosApplication.PxOpts** file for each Pyxos Point and Pilot. This file stores implementation specific options for the Point or Pilot.

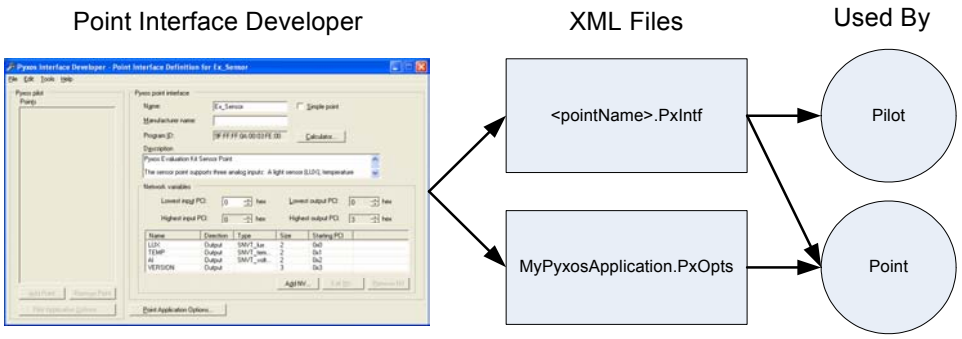

Figure 14 Generated XML Files

Generated Header Files

The Pyxos FT Interface Developer also generates a **<*PointName*>PointInterface.h** header file from each Point interface definition file. This file defines constants and macros for the Point interface definition, and will be included by both Pyxos Point and Pilot APIs and applications.

A group of **Resources.h** files will also be generated, one for the Pilot and one for each Point interface definition file. This file defines PNV types, and may require customization for a particular host.

A **MyPyxosApplication.h** file will be generated from the **MyPyxosApplication.PxOpts** files for each Pyxos Point and Pilot, so that there is also one **MyPyxosApplication.h** file for each Pyxos Point and Pilot. The **MyPyxosApplication.h** file contains macros used to control the compilation of the Pyxos Point and Pilot APIs.

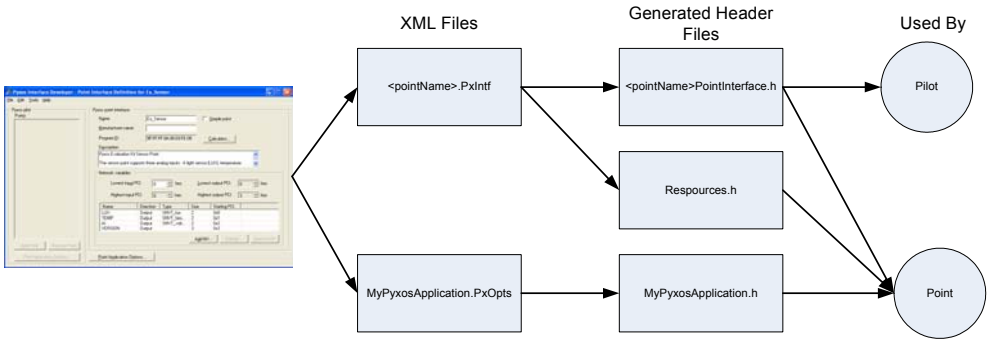

Figure 15 Generated XML Files

Generating Include Files

To generate the include files, follow these steps:

1. From the **File** menu, select **Open** and then select **Network Project** to open a network project.

2. From the **File** menu, select **Generate All Include Files** to generate the include files for the project.

3. Each Point or Pilot application needs to include the **Pyxos.h** file, which will in turn include its respective include file (the **MyPyxosApplication.h** file). For Pyxos Points, this file automatically includes the interface header file for the Point interface definition selected for the Point. For the Pilot application, this file includes the interface include file for each Point included in the network.

3

Creating a Pyxos Point Application

This chapter describes how to create a Pyxos Point application using the Pyxos Point API.

Overview

This chapter describes how to create a Pyxos Point application using the Pyxos Point API. A Pyxos Point application using the Pyxos Point API performs the following tasks:

1. Initialize the Pyxos FT Chip and the Pyxos FT API. For more information on this task, see *Initializing the Pyxos FT API* on page 36.
2. If the Pyxos Point uses the hardwired registration method, announce its timeslot after initializing the Pyxos Point. For more information on this task, see *Allocating Timeslots* on page 36.
3. Periodically call **PyxosPointEventHandler()** to send PNV updates to the Pilot and to receive PNV updates from the Pilot. For more information on this task, see *Handling Events* on page 38.
4. Read and write to PNVs on the Pyxos Point, as your application requires. For more information on this task, see *Reading, Writing and Receiving PNV Updates* on page 39.

Table 11 lists the functions you will use to perform each of these tasks. These functions are described in more detail later in the chapter. The *Pyxos Point Example Application* section on page 43 includes sample code you may find useful when following these steps to create your own Pyxos Point application.

Table 11 Point API Functions

Function	Description	For More Information, See…..
psInit()	Initializes the Pyxos Serial API. This function must be called before **PyxosPointInit()**. The implementation of **psInit()**is processor-specific, and generally must be called before I/O is enabled on the processor.	*Initializing the Pyxos FT API* on page 36
PyxosPointInit()	Initializes the Pyxos FT API and the Pyxos FT Chip.	

Function	Description	For More Information, See…..
PyxosPointSendUniqueId()	Registers a Pyxos Point with the Pilot application. This function is typically only used with manual registration without using a **Join** button. It is not required when using the automatic and hardwired registration. It is also not required for manual registration if your Pyxos Point uses the **Join** button to send registration requests—so in many cases your application will not need to use this function.	*Allocating Timeslots* on page 36
PyxosPointAnnounceTimeslot()	Assigns a timeslot to a Pyxos Point when using the hardwired registration method.	
PyxosPointGetTimeslot()	Determines the timeslot that has been assigned to a Pyxos Point.	
PyxosPointIsOnline()	Returns True if the Point is online, False otherwise.	
PyxosPointEventHandler()	Sends any pending updates to the Pyxos Pilot and processes any updates received by the Pyxos Point. Your application must call this function periodically to determine whether there are any Pyxos events to process.	*Handling Events* on page 38
PyxosPointPollPnv()	Requests the Pilot to update one of the Pyxos Point's input PNVs.	*Reading, Writing and Receiving PNV Updates* on page 39
PyxosPointUpdatePnv()	Writes the value of a PNV on the Pyxos Point, and sends the new value to the Pyxos Pilot.	
PyxosPointIsPnvUpdatePending()	Determines if there are any updates for a PNV cached in the memory of the Pyxos FT Chip or in the API that have not been sent to the Pilot.	

Initializing the Pyxos FT API

The first step to take when creating a Pyxos Point application is to initialize the Pyxos serial driver and the Point API. You can do so with the **psInit()** and **PyxosPointInit()** functions.

psInit()

Initializes the Pyxos serial driver.

Syntax:

```
void psInit(void);
```

Remarks: The Pyxos Point application must call this function once when it starts up, after every power-up or reset. The implementation of this function depends on the host processor, and is often called before processor I/O has been enabled. See Chapter 6, *Porting the Pyxos FT API*, for more information concerning **psInit()**. After this function has been called and processor I/O has been enabled, the application must call **PyxosPointInit()**.

PyxosPointInit()

Initializes the Pyxos FT API and the Pyxos FT Chip.

Syntax:

```
PyxosSts PyxosPointInit(void);
```

Remarks: A Pyxos Point application must call the **PyxosPointInit()** function once when the device starts up after a power-up or a reset, immediately after calling the **psInit()** function and processor I/O has been enabled. This initializes the Pyxos FT API and the Pyxos FT Chip, and writes the Point's program ID to the Pyxos FT Chip. You can specify the Point's program ID with the Pyxos FT Interface Developer, as described in Chapter 2, *Using the Pyxos FT Interface Developer*.

Allocating Timeslots

If the Pyxos Point is using the hardwired registration method, then the application must assign the Pyxos Point a timeslot after initializing the Pyxos Point with the **PyxosPointInit()** function. This is accomplished by calling the **PyxosPointAnnounceTimeslot()** function.

If the Pyxos Point is not using the hardwired registration method, then the Pilot will assign the Pyxos Point a timeslot, and you can begin handling events and reading and writing PNV values with your Pyxos Point application immediately after initialization. These tasks are described in the *Handling Events* and *Reading, Writing and Receiving PNV Updates* sections later in the chapter.

Timeslot Functions

This section describes the functions that are used to allocate a Pyxos Point a timeslot.

PyxosPointAnnounceTimeslot()

Assigns a Pyxos Point a timeslot when using the hardwire registration method. The timeslot to use can be hardcoded into the Pyxos Point application, read from a wiring harness or serial identification tag, or determined via some other application-dependent means.

Syntax:

```
PyxosSts PyxosPointAnnounceTimeslot(PyxosTimeslot timeslot);
```

Remarks: Call this function only once after start-up, reset, or power up. It sets the timeslot in the Pyxos Point, sets the Pyxos Point into configured mode, and announces the timeslot to the Pilot application.

To use this function, the **PYXOS_REGISTRATION_MODE** macro must be defined as **PYXOS_REGISTRATION_MODE_HARDWIRED** in the **MyPyxosApplication.h** file. This will be the case if you selected **Hardwired Registration** as the **Registration Method** on the Point Application Options dialog when you created the Point Interface Definition file for this type of Point with the Pyxos FT Interface Developer.

PyxosPointGetTimeslot()

Returns the timeslot currently assigned to the Pyxos Point. You can use this to verify that the Pyxos Point has been assigned a timeslot.

Syntax:

```
PyxosTimeslot PyxosPointGetTimeslot(void);
```

Remarks: This function returns the Pyxos Point's timeslot, if the Point is currently configured. If it is not configured, it returns **PYXOS_NO_TIMESLOT**.

If the **PYXOS_REGISTER_BUFFER_SIZE** macro in the **MyPyxosApplication.h** file is set to a value less than 4, calling this function from a callback will fail and return **PYXOS_NO_TIMESLOT**, even if the Pyxos Point has been assigned a timeslot. The value of the **PYXOS_REGISTER_BUFFER_SIZE** macro corresponds to the value you selected for the **Register Buffer Size** field on the Point Application Options dialog when you created the Point Interface Definition file for this type of Point with the Pyxos FT Interface Developer.

This function does not depend on calling the **PyxosPointEventHandler()** function. For example, the application can wait until the Point has been configured to perform a task as shown in the following example:

```
PyxosTimeslot timeslot = PYXOS_NO_TIMESLOT;

while ((timeslot = PyxosPointGetTimeslot ()) == PYXOS_NO_TIMESLOT);
```

However, it is not generally necessary for a Point to wait until it has been configured to perform network tasks. The application can update PNVs and the API will write the value into its cache. The value will be written to the Pyxos FT Chip during a subsequent call to the **PyxosPointEventHandler()** function after the Point has been configured and set online.

PyxosPointIsOnline()

Returns True if the Point has been set online,

Syntax:

```
Bool PyxosPointIsOnline(void);
```

Remarks: This function returns True if the Pilot has set the Point online, and False otherwise.

The application can update PNVs before the Pilot has set the Point online, but the API will not write them out to the Pyxos FT Chip until a subsequent call is made to **PyxosPointEventHandler()**, after the Point has been configured and set online.

PyxosPointSendUniqueId()

Sends the Pyxos Point's unique ID to the Pilot application.

Syntax:

```
PyxosSts PyxosPointSendUniqueId(void);
```

Remarks: This function is used in the manual registration method to allow the Pyxos Point to announce itself to the Pilot application. If the Pyxos Point has not yet been allocated a timeslot, the message will be sent on a random free timeslot. Typically, the **Join** button on the Pyxos Point is used to send registration requests to the Pilot with the manual registration method, and your application will not need to use this function.

If the **PYXOS_REGISTER_BUFFER_SIZE** macro in the **MyPyxosApplication.h** file is set to a value less than 8**,** calling this function from a callback will fail and return the **PyxosSts_NotAllowed** error. The value of the **PYXOS_REGISTER_BUFFER_SIZE** macro corresponds to the value you selected for the **Register Buffer Size** field on the Point Application Options dialog when you configured the Point application with the Pyxos FT Interface Developer.

Handling Events

After you have initialized the Pyxos Point and (if necessary) allocated the Pyxos Point a timeslot, you can begin calling the **PyxosPointEventHandler()** function to read incoming data from the network, and call the appropriate callbacks.

PyxosPointEventHandler()

Reads incoming data from the network, and calls the appropriate callbacks. Your Pyxos Point application must call this function periodically. If there are any

pending updates to be sent on the network, they are written to the Pyxos FT Chip when this function is called.

Syntax:

```
PyxosSts PyxosPointEventHandler(void);
```

Remarks: All Pyxos Point applications must call this function periodically. Otherwise, the application will not be able to receive or send data to the network. The application may call this in response to a Pyxos interrupt. For more information on Pyxos interrupts, see *Interrupt Driven Pyxos Programs* on page 166.

The Point API includes one callback, **PyxosPointPnvUpdateOccurred()**. This callback will be fired each time a PNV on the Pyxos FT network is updated. For more information on this, see the next section, *Reading, Writing and Receiving PNV Updates*.

Reading, Writing and Receiving PNV Updates

Your application can read and write the values of PNVs on the Pyxos Point with the **PyxosPointUpdatePnv()** and **PyxosPointPollPnv()** functions. It can send these PNV updates onto the network, and receive updates from the Pilot, with the **PyxosPointPnvUpdateOccurred()** callback. These functions and callbacks are described in this section.

PyxosPointUpdatePnv() Function

Writes the value of an output network variable and sends the value to the Pyxos Pilot.

Syntax:

```
PyxosSts PyxosPointUpdatePnv(PyxosPci pci, Byte *pData);
```

Remarks: The PNV to be updated must be referenced by its Pyxos Chip index (*pci* parameter) as defined in the Point interface definition file used by the Point, and the PNV must be defined as an output PNV. You can specify the value to be written to the PNV with the *pData* parameter.

The API buffers the output value and writes the value to the Pyxos FT Chip when **PyxosPointEventHandler()** is called, unless an unacknowledged update for this PNV has already been sent to the chip. In that case, the value will be written to the Pyxos FT Chip on a subsequent call to **PyxosPointEventHandler()** after the previous write requests have been acknowledged. Once the value has been written to the Pyxos FT Chip, the Pyxos FT Chip will send the value to the Pilot when it can. Updates for multiple PNVs are sent out on a round-robin basis. If the **PyxosPointUpdatePnv()** function is called multiple times for the same PNV while an update is in progress, all of the intermediate values for the PNV will be lost, meaning that only the last updated value will be sent onto the network.

The application may update PNVs before the Point is configured and set online. The API will cache the value, but will not write the value to the Pyxos FT Chip

until a subsequent call is made to **PyxosPointEventHandler()**, after the Point has been configured and set online.

The order in which PNVs are sent on the network **is not necessarily the same** as the order that your application uses to update them.

If you attempt to update an input PNV with this function the operation will fail.

All PNVs are sent in big endian format. If your host processor is a little endian processor, your application may have to swap the bytes of multi-byte fields prior to calling the **PyxosPointUpdatePnv()** function. Macros defined in the **Platform.h** file are provided to aid in this transformation. See the *Modifying the Platform.h File* section on page 115 for more information on this.

You cannot use this function if the **PYXOS_NO_OUTPUTS** macro is defined in the **MyPyxosApplication.h**. This is only the case if the Pyxos Point does not have any output PNVs.

The <*PointName*>**PointInterface.h** file includes a **#define** for each Pyxos FT PNV, which includes the lowest PCI value for the PNV. You can use the **#define** as the *pci* parameter. For example, refer to the excerpt from the **ACMESensorPointInterface.h** file below. For more information on the **ACMESensorPointInterface.h** file, see the *Pyxos Point Example Application* section on page 43.

```
/* PNV definitions */

/* type name: SNVT_switch           */
#define ACME_SENSOR_SWITCH                      0x8
/* type name: SNVT_switch           */
#define ACME_SENSOR_SWITCH_FEEDBACK             0x0
#define PYXOS_ACME_SENSOR_NETWORK_VARIABLES   \
      PYXOS_DEFINE_PNV(PYXOS_POINT_OUTPUT,    \
                       ACME_SENSOR_SWITCH, 2) \
      PYXOS_DEFINE_PNV(PYXOS_POINT_INPUT,     \
                       ACME_SENSOR_SWITCH_FEEDBACK, 2)
```

The Point can send the value of the switch as follows:

```
SNVT_switch switchValue;
if (ButtonPressed())
{
   /* Button was pressed.  Toggle the switch, update the
      status led and send the update to the Pilot.
    */
   switchValue.state = !switchValue.state;
   switchValue.value = switchValue.state ? 200 : 0;
   SetLed(switchValue.state);

   PyxosPointUpdatePnv(ACME_SENSOR_SWITCH,
                       (Byte *)&switchValue);

}
```

PyxosPointPollPnv() Function

Sends a request to the Pilot to update the value of one of the Pyxos Point's input PNVs.

Syntax:

```
PyxosSts PyxosPointPollPnv(PyxosPci pci);
```

Remarks: The PNV to be polled must be referenced by its Pyxos Chip index (*pci* parameter), as defined in the Point interface definition file used by the Point. The poll request will be written to the Pyxos FT Chip when **PyxosPointEventHandler()**is called, unless an unacknowledged poll request already exists. In that case, the poll request will be written to the Pyxos FT Chip on a subsequent call to **PyxosPointEventHandler()** after the previous poll request has been acknowledged. Once the poll request has been written to the Pyxos FT Chip, the Pyxos FT Chip sends a poll request to the Pilot as soon as possible. Sending the poll request to the Pilot competes with sending updates to the Pilot on a round-robin basis. The Pyxos FT protocol supports sending up to four poll requests at a time. If more than four poll requests are pending, they will also be processed in round-robin order.

The **<***PointName***>PointInterface.h** file includes a **#define** for each PNV, which includes the lowest PCI value for the PNV. You can use the **#define** as the *pci* parameter. For example, refer to the excerpt from the **ACMESensorPointInterface.h** file below. For more information on the **ACMESensorPointInterface.h** file, see *Pyxos Point Example Application* on page 43.

```
/* PNV definitions */

/* type name: SNVT_switch           */
#define ACME_SENSOR_SWITCH                      0x8

/* type name: SNVT_switch           */
#define ACME_SENSOR_SWITCH_FEEDBACK             0x0

#define PYXOS_ACME_SENSOR_NETWORK_VARIABLES   \
      PYXOS_DEFINE_PNV(PYXOS_POINT_OUTPUT,    \
                       ACME_SENSOR_SWITCH, 2) \
      PYXOS_DEFINE_PNV(PYXOS_POINT_INPUT,     \
                       ACME_SENSOR_SWITCH_FEEDBACK, 2)
```

The Point can poll the value of the switch feedback as follows:

```
PyxosPointPollPnv(ACME_SENSOR_SWITCH_FEEDBACK);
```

It is generally not necessary for a Point to poll any of its input PNVs. Instead, it should process them when the Pilot decides to send them. However, if the Point has some need to refresh the value, this function can be used to do so.

If you attempt to poll an output PNV with this function the operation will fail.

The application can poll PNV values before the Point is configured and set online. The API will cache the request, but will not write the request to the Pyxos Chip

until a subsequent call is made to **PyxosPointEventHandler()**, after the Point has been configured and set online.

To use this function, the **PYXOS_ENABLE_POLLING** macro must be defined in the **MyPyxosApplication.h** file. This means that you must select the **Enable Polling** check box on the Point Application Options dialog when you configure the Point Interface Definition file for the Pyxos Point with the Pyxos FT Interface Developer.

PyxosPointIsPnvUpdatePending()

Determines if there are any updates for a PNV cached in the memory of the Pyxos FT Chip, or in the API that have not yet been sent to the Pilot.

Syntax:

```
Bool PyxosPointIsPnvUpdatePending(PyxosPci pci);
```

Remarks: The PNV to be updated must be referenced by its Pyxos Chip index (*pci* parameter) as defined in the Point interface definition file used by the Point. The function will return True if the PNV has any updates pending, and False otherwise. You cannot call this function on an input PNV.

This function cannot be used if the **PYXOS_NO_OUTPUTS** macro is defined in the **MyPyxosApplication.h** file.

PyxosPointPnvUpdateOccurred() Callback

Handles a PNV update. Your application must implement this callback if it defines any input PNVs. It will be called by **PyxosPointEventHandler**() for each PNV update that is received by the Point.

Syntax:

```
void PyxosPointPnvUpdateOccurred(PyxosPci pci,
                                 const Byte *pPnvValue,
                                 Byte length);
```

Remarks: This function is called from **PyxosPointEventHandler()** whenever an input variable is updated. You can use the Pyxos Chip index (*pci* parameter) to identify the PNV that was updated. The update may have been initiated independently by the Pilot or may be in response to calling **PyxosPointPollPnv()**.

The function also provides the value currently assigned to the PNV (*pPnvValue* parameter) and the PNV's length (*length* parameter).

All PNVs are sent in big endian format. If your host processor is a little endian processor, your application may have to swap the bytes of multi-byte fields prior to calling **PyxosPointUpdatePnv()**. Macros defined in the **Platform.h** file are provided to aid in this transformation. See the *Modifying the Platform.h File* section on page 115 for more information on this.

This function will not be called if the **PYXOS_NO_INPUTS** macro is defined in the **MyPyxosApplication.h**. This is only the case if the Pyxos Point does not have any input PNVs.

The **<*PointName*>PointInterface.h** file includes a **#define** for each PNV, which includes the lowest PCI value for the PNV. You can use the **#define** as the *pci* parameter. For example, refer to the excerpt from the **ACMESensorPointInterface.h** file below. For more information on the **ACMESensorPointInterface.h** file, see *Pyxos Point Example Application* on page 43.

```
/* PNV definitions */

/* type name: SNVT_switch           */
#define ACME_SENSOR_SWITCH                      0x8
/* type name: SNVT_switch           */
#define ACME_SENSOR_SWITCH_FEEDBACK             0x0

#define PYXOS_ACME_SENSOR_NETWORK_VARIABLES    \
      PYXOS_DEFINE_PNV(PYXOS_POINT_OUTPUT,     \
                       ACME_SENSOR_SWITCH, 2)  \
      PYXOS_DEFINE_PNV(PYXOS_POINT_INPUT,      \
                       ACME_SENSOR_SWITCH_FEEDBACK, 2)
```

When the Pilot updates the sensor feedback, the Point can receive the update as follows:

```
/* This function is called by the Point API whenever a new
   value is received.
 */
void PyxosPointPnvUpdateOccurred(PyxosPci pci,
                                 const Byte *pPnvValue,
                                 Byte length)
{
   if (pci == ACME_SENSOR_SWITCH_FEEDBACK) {
      /* Update the switch state based on the feedback, and
         then light the LED accordingly
       */
      memcpy(&switchValue, pPnvValue, sizeof(switchValue));
      SetLed(switchValue.state);
   }
}
```

Pyxos Point Example Application

This section provides a simple example Pyxos Point application that has one output PNV, ACME_SENSOR_SWITCH, and one input PNV, ACME_SENSOR_SWITCH_FEEDBACK. The sensor has a single push-button switch and an LED that is used to reflect the state of the switch.

Whenever the button is pushed, the switch's state is toggled and the value is sent to the Pilot. The Pilot can also update the state of the switch by updating the ACME_SENSOR_SWITCH_FEEDBACK PNV. When the Pyxos Point receives an update of the ACME_SENSOR_SWITCH_FEEDBACK PNV, it updates the state of the switch and the LED accordingly.

This example (**ACMESensorPointInterface.h**) uses the following functions whose implementations are not shown:

- **HostIoInit():** Initializes the host I/O, including calling **psInit()**.

- **ButtonPressed():** Returns a Boolean indicating that the switch has been pressed.
- **SetLed(Bool on):** Turns the LED on or off, based on the value specified.

The current state of the switch is stored in a global variable of type **SNVT_switch** called **switchValue**. The next section shows example code you can use for Pyxos Points that use the automatic or manual registration methods. Following that is example code for a Pyxos Point that is using hardwired registration. The registration method that each Pyxos Point should use is specified in the Point's **MyPyxosApplication.h** file, and the Point API sets the mode in the Pyxos FT Chip accordingly.

When you use the Point API, you must include the appropriate Pyxos files, as described in Chapter 5, *Including the Pyxos FT API*.

Example for Manual or Automatic Registration

```
#include "Pyxos.h"
/* Current state of the switch. This is modified locally when the
   user presses the sensor's button, and is also updated based on
   the feedback from the Pilot.
 */
SNVT_switch switchValue;

/* This is the application entry Point and main control loop. */
int main(void)
{
   /* Initialize switch state to FALSE. */
   memset(&switchValue, 0, sizeof(switchValue));

   /* Initialize I/O including Pyxos Serial Driver */
   HostIoInit();

   /* Initialize the Point API. */
   PyxosPointInit();

   /* This is the main control loop, which runs forever. */
   while (TRUE) {
      /* Run the Pilot API event handler to process Pyxos Events.
       */
      PyxosPointEventHandler();
      if (ButtonPressed()) {
         /* Button was pressed.  Toggle the switch, update the
            status led and send the update to the Pilot.
          */
         switchValue.state = !switchValue.state;
         switchValue.value = switchValue.state ? 200 : 0;
         SetLed(switchValue.state);

         PyxosPointUpdatePnv(ACME_SENSOR_SWITCH,
                             (Byte *)&switchValue);
      }
   }

   return 0;
}
```

```
/* This function is called by the Point API whenever a new value
   is received.
 */
void PyxosPointPnvUpdateOccurred(PyxosPci pci,
                                 const Byte *pPnvValue,
                                 Byte length)
{
   if (pci == ACME_SENSOR_SWITCH_FEEDBACK) {
      /* Update the switch state based on the feedback, and then
         light the LED accordingly
       */
      memcpy(&switchValue, pPnvValue, sizeof(switchValue));
      SetLed(switchValue.state);
   }
}
```

Example for Hardwired Registration

This example is identical to the one in the previous section except for the first few lines of initialization, during which the Point determines the timeslot it should use and configures itself. This example uses the **DetermineTimeslot()** function (not shown), which returns the timeslot that the Point should use. This function may read a serial device, dip switches, or use some other external means of customization to determine the timeslot that the Point should use.

The first several lines of the main program are shown below with the modifications in bold. The rest of the example is the same as in the previous section:

```
/* This is the application entry Point and main control loop. */

int main(void)
{
   PyxosTimeslot timeslot;

   /* Initialize switch state to FALSE. */
   memset(&switchValue, 0, sizeof(switchValue));

   /* Initialize I/O including Pyxos Serial Driver */
   HostIoInit();

   /* Initialize the Point API. */
   PyxosPointInit();

   /* Retrieve the timeslot from the user settings. */
   timeslot = DetermineTimeslot();

   /* Configure the Point in the desired timeslot, and announce
      the Point to the Pilot.
    */
   PyxosPointAnnounceTimeslot(timeslot);

   /* This is the main control loop, which runs forever. */
   while (TRUE) {
```

-
-
-

4

Creating a Pilot Application

This chapter describes how to create a Pilot application to register, monitor, and control the Pyxos Points on a network.

Overview

This chapter describes how to create a Pilot application using the Pyxos Pilot API to register, monitor, and control the Pyxos Points on a network. A Pyxos Pilot application performs the following tasks:

1. Initialize the Pyxos FT Chip and the Pyxos FT API. This task must be performed before any of the others. For more information on this, see *Initializing the Pyxos Pilot* on page 50.

2. Implement a control loop that periodically calls the Pyxos event handler to handle Pyxos events, and to send PNV updates to the Pyxos Points on the network. For more information on this, see *Handling Events* on page 51.

3. Register the Pyxos Points on the network. This involves receiving registration requests from the Pyxos Points, allocating timeslots to them, reading their program IDs, and identifying the interfaces they should use. For more information on this, see *Registering Points* on page 55.

4. Receive updates for PNVs from the Pyxos Points on the network, and send PNV updates to Pyxos Points. For more information on this task, see *Reading and Writing PNVs* on page 67.

5. Monitor the health of the network, and recover from network errors by resetting, reconfiguring, or replacing Pyxos Points when necessary. For more information on this, see *Detecting, Reporting, and Correcting Communication Errors* on page 77.

Table 12 lists and describes the functions you will use to perform each of these tasks. These functions are described in more detail in the sections listed above. This chapter also includes example code that will help you get started developing your own Pilot application.

Table 12 Pilot API Functions

Function	Description	For More Information, See…..
psInit()	Initializes the Pyxos Serial API. This function must be called before **PyxosPilotInit()**. The implementation of **psInit()**is processor-specific, and generally must be called before I/O is enabled on the processor.	*Initializing the Pyxos Pilot* on page 50
PyxosPilotInit()	Initializes the Pyxos FT Chip and the Pyxos FT API. The Pilot application must call this function when it starts, after calling **psInit()**.	
PyxosPilotEventHandler()	Processes Pyxos events. The Pilot application must call this function periodically.	*Handling Events* on page 51
PyxosPilotAllocateTimeslot()	Allocates a timeslot to a Pyxos Point.	*Registering Points* on page 53
PyxosPilotFreeTimeslot()	Frees a timeslot that has been previously allocated to a Pyxos Point.	
PyxosPilotGetTimeslot()	Determines which timeslot a Pyxos Point is currently using.	
PyxosPilotGetUniqueId()	Determines the unique ID of the Pyxos Point associated with a particular timeslot.	
PyxosPilotGetProgramId()	Gets the program ID of a Point.	
PyxosPilotGetPointInterface()	Obtains a pointer to the interface being used by the Pyxos Point assigned to a particular timeslot.	
PyxosPilotSetPointInterface()	Identifies the interface a Pyxos Point should use.	
PyxosPilotGetNumberOfPoints()	Reads the number of Pyxos Points that have been allocated timeslots, and the number of Pyxos Points whose Point interfaces have been specified.	
PyxosPilotSetPointOnline()	Sets a Pyxos Point online. Prior to being set online, the Point will not send any PNV values to the Pilot and the Pilot will not send any PNV values to the Point.	

Function	Description	For More Information, See…..
PyxosPilotUpdatePnv()	Updates the value of an input PNV on a Pyxos Point, and schedules the propagation of the value onto the network.	*Reading and Writing PNVs* on page 67
PyxosPilotIsPnvUpdatePending()	Checks if any updates for a PNV are cached in the API but have either not yet been sent to or have not been acknowledged by the Point.	
PyxosPilotUpdateUnhostedPointIo()	Updates digital output values on an unhosted Point.	
PyxosPilotPollRegister()	Polls the value of a Register on a Pyxos Point.	
PyxosPilotGetPnvValue()	Reads the cached value of an input PNV on a Pyxos Point.	
PyxosPilotResetPoint()	Resets a Pyxos Point. This resets the Point's Pyxos FT Chip and host processor. If the Point uses manual or automatic registration, this will reconfigure the Point. If it uses hardwired registration, this will verify that the Point has configured itself.	*Detecting, Reporting, and Correcting Communication Errors* on page 75
PyxosPilotReplacePoint()	Replaces a Pyxos Point with another of the same type.	
PyxosPilotReconfigurePoint()	Reconfigures a Pyxos Point that may have been reset.	
PyxosPilotCheckConfiguration()	Checks the configuration of the Pyxos FT Chip used by the Pilot application.	
PyxosPilotReInitPyxosInterface()	Re-initializes the Pyxos FT Chip used by the Pilot.	

Initializing the Pyxos Pilot

The first step to complete when creating a Pyxos Pilot application is to initialize the Pyxos serial driver and the Pilot API. You can do so by calling the **psInit()** and **PyxosPilotInit()** functions.

psInit()

Initializes the Pyxos serial driver.

Syntax:

```
void psInit(void);
```

Remarks: The Pilot application must call this function once after every reset. The implementation of this function depends on the host processor, and is often called before processor I/O has been enabled. See Chapter 6, *Porting the Pyxos FT API*, for more information concerning **psInit()**. After this function has been called and processor I/O has been enabled the Pilot must call **PyxosPilotInit()**.

PyxosPilotInit()

Initializes the Pyxos FT API and the Pyxos FT Chip.

Syntax:

```
PyxosSts PyxosPilotInit(void);
```

Remarks: The Pilot application must call **PyxosPilotInit()** once after every reset. You must call this function after **psInit()** has been called and the processor I/O has been enabled. This initializes the Pyxos FT Chip being used by the Pilot application, and the Pyxos FT API.

Handling Events

The Pyxos Pilot application must periodically call **PyxosPilotEventHandler()** to handle Pyxos events, and to send PNV updates to the Pyxos Points on the network. Typically, this is done within a control loop that is always running.

Alternatively, you may invoke the event handler from an interrupt routine. While this may be useful in some cases, it complicates the Pyxos Pilot application. For more information on Pyxos interrupts, see *Interrupt Driven Pyxos Programs* on page 166.

The *Example Code for Event Handling, and Registering Pyxos Points* section on page 62 includes sample code you may find useful when creating your own event handler.

PyxosPilotEventHandler()

Reads incoming data from the network, and calls the appropriate callbacks. Your Pilot application must call this function periodically. If there are any pending PNV updates or poll requests to be sent on the network, they are written to the Pyxos FT Chip when this function is called.

Syntax:

```
PyxosSts PyxosPilotEventHandler(void);
```

Remarks: The Pilot application can call this function in response to a Pyxos interrupt. For more information on Pyxos interrupts, see *Interrupt Driven Pyxos Programs* on page 166.

There are multiple callbacks that can be called by the event handler. Table 13 lists these callbacks. These callbacks are described in more detail in the sections following Table 13.

Table 13 Pilot API Callbacks

Callback	Description
`PyxosPilotRegistrationRequestReceived()`	Handles a registration request to the Pilot from a Pyxos Point. For more information on this callback, see *Registering Points* on page 53.
`PyxosPilotPointConfigured()`	Provides notification that a Point has been successfully configured. This may be the result of a call to **PyxosPilotAllocateTimeslot()**, **PyxosPilotReconfigurePoint()**, or **PyxosPilotReplacePoint()**. Or, if the Point uses hardwired registration, this may occur because the Point has reset and reconfigured itself. For more information on this callback, see *Registering Points* on page 53.
`PyxosPilotPointConfigurationFailed()`	Provides notification that configuration of a Point initiated via a call to **PyxosPilotAllocateTimeslot()**, **PyxosPilotReconfigurePoint()** or **PyxosPilotReplacePoint()**has failed. For more information on this callback, see *Registering Points* on page 53.
`PyxosPilotFreeTimeslotCompleted()`	Provides notification that a call to **PyxosPilotFreeTimeslot()**has completed, meaning that the Pilot application has successfully freed a timeslot. For more information on this callback, see *Registering Points* on page 53.
`PyxosPilotSetPointOnlineCompleted()`	Provides notification that a request to set the Point online has either successfully completed, or timed out. For more information on this callback, see *Callbacks for Registering Points* on page 60.
`PyxosPilotUpdatePnvCompleted()`	Provides notification that a PNV update request has either successfully completed, or timed out. For more information on this callback, see *Reading and Writing PNVs* on page 67.
`PyxosPilotPollRegisterCompleted()`	Provides notification that a register poll request has successfully completed or timed out. For more information on this callback, see *Reading and Writing PNVs* on page 67.

Callback	Description
`PyxosPilotPnvUpdateOccurred()`	Provides notification that a PNV value has been updated. This may be the response to a poll request, or it may be the result of a Pyxos Point application updating the value. For more information on this callback, see *Reading and Writing PNVs* on page 67.
`PyxosPilotResetPointCompleted()`	Provides notification that a call to **PyxosPilotResetPoint()** has completed, meaning that a Pyxos Point has been successfully reset or a reset operation has timed out. For more information on this callback, see *Detecting, Reporting, and Correcting Communication Errors* on page 77.
`PyxosPilotProtocolAnalyzerCallback()`	Provides notification that a the Pilot API has read or written a packet. For more information on this callback, see *Analyzing Pyxos FT Network Communication* on page 92.

pyxosPilotWriteFrameCount Global Variable

The Pyxos FT API maintains a global variable (**pyxosPilotWriteFrameCount**) that represents the number of write frames processed by the API. This value can be used to efficiently keep track of the number of write frames to implement network-based timers.

Syntax:

```
Dword pyxosPilotWriteFrameCount;
```

Remarks: The **pyxosPilotWriteFrameCount** variable is set to 0 by **PyxosPilotInit()**, and is incremented by **PyxosPilotEventHandler()**each time it processes a write frame. This value can be used to efficiently keep track of the number of write frames to implement network-based timers. Use of this variable is substantially more efficient than reading the network statistics, as no SPI transaction is required.

This value may be less than the actual number of frames if **PyxosPilotEventHandler()** is not called frequently enough. This value wraps at 0xFFFFFFFF. When using this value to implement frame relative timing, the application must handle wrap-arounds.

The application must never modify this value, as it is used by the Pyxos FT API to detect update failures.

Registering Points

The Pilot must allocate timeslots to all the Pyxos Points on the network, and inform the Pilot API which interface each Pyxos Point is implementing. Once a Point has been configured, the Pilot must set the Point online. This process is called *registration*.

Typically, the first time the Pilot starts, it does not have any pre-programmed knowledge about any of the Pyxos Points on the network, although it does have knowledge about the program IDs and interface types supported on the network from the application's include files. The Pilot allocates timeslots and identifies interfaces as it receives the registration events that are called when each Pyxos Point announces itself.

You can program your Pilot application to remember the unique IDs of the Pyxos Points that it has discovered in the past by storing that information in non-volatile memory. Such applications can allocate timeslots immediately upon initialization, without waiting for those Pyxos Points to announce themselves. This is particularly important in systems with Points using the manual registration method.

After the Pilot has successfully configured a Point and set its interface, the Pilot must set the Point online in order to start receiving PNV values from the Point, and propagating PNV values to the Point.

The typical steps you will follow to allocate timeslots and identify interfaces are listed below. Depending on how you chose to register a particular Pyxos Point (i.e. which registration method you selected for the Point), some of these steps may vary.

1. The **PyxosPilotRegistrationRequestReceived()** callback is triggered. This occurs whenever a Pyxos Point announces itself to the Pilot. Pyxos Points using the automatic registration method announce themselves on startup after calling **PyxosPointInit()**, and continue until the Pilot has configured them. Pyxos Points using the manual registration method typically announce themselves when the Pyxos FT Chip's **Join** button is pressed. Alternatively, a Pyxos Point can announce itself by calling **PyxosPointSendUniqueId**(). Pyxos Points using the hardwired registration method announce themselves when they call **PyxosPointAnnounceTimeslot()**. In all of these situations, the **PyxosPilotRegistrationRequestReceived()** callback is triggered.

   From the callback, the Pilot application must call **PyxosPilotAllocateTimeslot()** to assign the Pyxos Point a timeslot. It should use the unique ID provided by the **PyxosPilotRegistrationRequestReceived()** callback to identify the Pyxos Point that is being assigned a timeslot.

   Depending on the registration method selected for the Pyxos Point, the application may specify a specific timeslot for the Pyxos Point, or select the first available timeslot. This is described later in the chapter, where **PyxosPilotAllocateTimeslot()** is described in more detail.

   Alternatively, if the Pilot knows about a Pyxos Point, it may allocate it a timeslot during initialization without waiting for the Point to announce itself. This is the case if the application has stored the Pyxos Point's unique ID in non-volatile memory, or if the Pyxos Point uses the hardwired registration method and the Pilot does not need the Point's unique ID. Regardless of whether the timeslot is allocated in response to a registration request or during initialization, the remaining steps required to allocate the timeslot and identify the Point's interface are the same.

2. The **PyxosPilotPointConfigured()** callback is triggered when the Point has been successfully configured. From this callback the application will typically perform the following steps:

    a. Get the Point's program ID by calling **PyxosPilotGetProgramId()**.

    b. Use the program ID to identify the Pyxos Point's interface with **PyxosPilotSetPointInterface()**. This function informs the Pyxos FT API which interface is implemented by the Pyxos Point, and allocates a cache of PNVs for the Point based on that interface.

    c. Set the Point online with **PyxosPilotSetPointOnline()**.

Functions for Registering Points

This section describes the functions listed in the previous section in more detail. It also describes additional functions you will find useful when allocating timeslots and identifying interfaces, such as **PyxosPilotFreeTimeslot()**, which you can use to free a timeslot that has been previously assigned to a Pyxos Point, and **PyxosPilotGetTimeslot()**, which you can use to determine the timeslot that is currently assigned to a Pyxos Point.

Some of the functions described in this section, such as **PyxosPilotSetPointOnline()**, are used when reconfiguring or replacing a Point, as well as during the initial registration process.

PyxosPilotAllocateTimeslot()

Assigns a timeslot to a Pyxos Point. The Pyxos Point to be assigned the timeslot is referenced by its unique ID.

Syntax:

```
PyxosSts PyxosPilotAllocateTimeslot(PyxosUniqueId *pUniqueId,
                                    PyxosTimeslot timeslot,
                                    Word timeout,
                                    Bool hardwired);
```

Remarks: Use this function to assign a timeslot to a Pyxos Point. You will typically call this function from the **PyxosPilotRegistrationRequestReceived()** callback, or during your application initialization routine (if the Point uses hardwired registration or the unique ID is already known). Most of the other functions included in the Pilot API require a timeslot as input to identify the Pyxos Point to be affected by the function. Therefore, you will need to call this function before using many of the others.

Use the *timeslot* parameter to indicate which timeslot should be assigned to the Pyxos Point. The valid range for timeslots is 0 to (**PYXOS_MAX_POINTS** – 1). The value for **PYXOS_MAX_POINTS** is set to the maximum number of Points you defined for the network with the Pyxos FT Interface Developer. You can set the **timeslot** parameter to **PYXOS_TIMESLOT_ANY** (0xFF) to allow the API to choose a free timeslot to assign to the Pyxos Point.

If the *hardwired* parameter is set to False, the Pilot will use the unique ID to configure the Point's Pyxos FT Chip with the allocated timeslot. If the *hardwired*

parameter is set to True, the API will not send the timeslot to the device, since this indicates that the Point is using the hardwired registration method and the timeslot should have already been assigned by the Point application. In either case, the API calls the **PyxosPilotPointConfigured()** callback after the Pilot has confirmed that the Point has been successfully configured, and calls the **PyxosPilotPointConfigurationFailed()** callback if the configuration process times out.

The *timeout* parameter indicates the number of frames that the Pilot should wait before timing out the configuration process. The time it takes for this process to complete varies depending on the Point's processor and application, especially if the Point has just been reset. Be sure to specify a sufficient timeout. A value of 0 indicates that the Pilot should use the default time period you specified with the **Number of Frames Before Timeout** property on the Pilot Application Options dialog when you configured the Pilot with the Pyxos FT Interface Developer. This value corresponds to the **PYXOS_MAX_UNACKD_FRAME_COUNT** macro in the **MyPyxosApplication.h** file.

If you define the **PYXOS_HARDWIRED_ONLY** macro in the **MyPyxosApplication.h** file, then you must use the hardwired registration method for all of the Pyxos Points on the network. This means that the *hardwired* parameter must be set to True, and the *timeslot* parameter cannot be **PYXOS_TIMESLOT_ANY**. You can define the **PYXOS_HARDWIRED_ONLY** macro in the **MyPyxosApplication.h** file by selecting the **Support Only Hardwired Points** checkbox in the Pilot Application Options dialog of the Pyxos FT Interface Developer, but only if you plan on using hardwired registration for every Pyxos Point on the network. This reduces the code size of the Pilot API.

In systems with hardwired Points, the Pilot application may allocate the timeslot before it receives the registration request from the Pyxos Point. However, in this case, the Pilot may not know the Pyxos Point's unique ID. For this reason, if the **hardwired** parameter is True, the *pUniqueId* parameter may be NULL. If it is NULL, the Pyxos FT API will record the unique ID as all zeros.

Neither the **PyxosPilotPointConfigured()** or the **PyxosPilotConfigurationFailed()** callbacks will be called unless **PyxosPilotAllocateTimeslot()**returns **PyxosSts_Good**. If this is the case, the timeslot has been successfully allocated and the API will call either the **PyxosPilotPointConfigured()** callback or the **PyxosPilotConfigurationFailed()** callback, depending on whether the Point successfully configured.

The **PyxosPilotAllocateTimeslot()** function may return the following errors:

- **PyxosSts_NoTimeslotsAvailable**: Returned when the *timeslot* parameter is set to **PYXOS_TIMESLOT_ANY**, but no timeslots are available.
- **PyxosSts_TimeslotNotAvailable**: Returned when the *timeslot* parameter is not set to **PYXOS_TIMESLOT_ANY**, meaning that you requested a specific timeslot, and that timeslot is already in use by another Point.
- **PyxosSts_InvalidTimeslot**: Returned when the *timeslot* parameter is not set to **PYXOS_TIMESLOT_ANY**. This means that the specified timeslot is greater than or equal to **PYXOS_MAX_POINTS** (the maximum number of Points on the network).

- **PyxosSts_AlreadyAllocatedToPoint**: Returned when the specified Point already has a timeslot allocated to it.
- **PyxosSts_UniqueIdIsRequired**: Returned when the unique ID pointer is NULL, and the *hardwired* flag is FALSE.

PyxosPilotFreeTimeslot()

Frees the timeslot assigned to a Pyxos Point.

Syntax:

```
PyxosSts PyxosPilotFreeTimeslot(PyxosTimeslot timeslot);
```

Remarks: This function frees the memory associated with the Pyxos Point using the specified timeslot, and resets the Point. Upon completion, the **PyxosPilotFreeTimeslotCompleted()** callback will be triggered, and the timeslot will be available to other Pyxos Points. The timeslot will not be available for reallocation until the Pilot API has called the **PyxosPilotFreeTimeslotCompleted()** callback.

PyxosPilotGetTimeslot()

Determines the timeslot that is currently assigned to a Pyxos Point. The Pyxos Point is referenced by its unique ID.

Syntax:

```
PyxosTimeslot PyxosPilotGetTimeslot(const PyxosUniqueId
                                    *pUniqueId);
```

Remarks: This function returns the timeslot assigned to a Pyxos Point. If the Pyxos Point has not been assigned a timeslot, the function returns **PYXOS_NO_TIMESLOT**. You can use this function if you want to check if a Pyxos Point has been configured or not.

PyxosPilotGetUniqueId()

Determines the unique ID of the Pyxos Point that is using a particular timeslot.

Syntax:

```
const PyxosUniqueId *PyxosPilotGetUniqueId(PyxosTimeslot
                                           timeslot);
```

Remarks: If the timeslot passed in as the *timeslot* parameter has been assigned to a Pyxos Point, the function returns a pointer to the Pyxos Point's unique ID. This pointer will be valid until the timeslot has been freed with the **PyxosPilotFreeTimeslot()** function. If the timeslot is not associated with a Pyxos Point, the function returns NULL.

PyxosPilotGetProgramId()

Determines the program ID of the Pyxos Point that is using a particular timeslot.

Syntax:

```
PyxosSts PyxosPilotGetProgramId(PyxosTimeslot timeslot,
                                PyxosProgramId *pProgramId);
```

Remarks: If the timeslot passed in as the *timeslot* parameter has been assigned to a Pyxos Point and the Point has been successfully configured, the function returns the program ID that the Pilot API cached during the configuration process. If the program ID is not available, the function returns **PyxosSts_ProgramIdNotAvailable**.

PyxosPilotSetPointInterface()

Identifies the interface a Pyxos Point should use. The Pyxos Point is specified by its timeslot.

Syntax:

```
PyxosSts PyxosPilotSetPointInterface(PyxosTimeslot timeslot,
          const PyxosPilotPointInterface *pPointInterface);
```

Remarks: The **PyxosPilotSetPointInterface()** function completes immediately, and is most often called from the **PyxosPilotPointConfigured()** callback. The function allocates and initializes memory to cache PNV values on a Point, and to manage sending and receiving data from the PNVs. You must assign a timeslot to the Pyxos Point with **PyxosPilotAllocateTimeslot()** function before calling **PyxosPilotSetPointInterface()**, and wait for the configuration to complete (indicated when the **PyxosPilotPointConfigured()** callback is called).

This function allocates the Pyxos cache data used for the Point, so any pending operations on the Point are canceled when this function is called. Therefore you should only call this function after all pending updates to the Point have completed. For example, you should never call this function immediately after calling the **PyxosPilotAllocateTimeslot()** function. Instead, you can call it from the **PyxosPilotPointConfigured()** callback.

If the Point interface supplied to the function corresponds to an unhosted Point, the Pilot API will initialize all of the I/O output values to 0. To set some of the digital values on the Point to 1, call **PyxosPilotUpdateUnhostedPointIo()** to set the values after calling **PyxosPilotSetPointInterface()** and before setting the Point online using **PyxosPilotSetPointOnline()**. For more information on this function, see *Reading and Writing PNVs* on page 67.

The Point interface record is typically defined as constant data. It must remain allocated as long as any Points are defined to use it. You can set the *pPointInterface* parameter to NULL to de-associate the Pyxos Point from the interface.

The Point interfaces you can reference with the *pPointInterface* parameter are defined in the Pilot application's **MyPyxosApplication.h** file. For example, the following interface definitions are declared in the example **MyPyxosApplication.h** file:

```
extern const struct PyxosPilotPointInterface echelonEx_NanoPointInterface;
extern const struct PyxosPilotPointInterface echelonEx_SensorPointInterface;
extern const struct PyxosPilotPointInterface echelonEx_ActuatorPointInterface;
```

You could identify the third interface listed with the following call to the **PyxosPilotSetPointInterface()** function:

```
PyxosPilotSetPointInterface(timeslot, &echelonEx_ActuatorPointInterface);
```

PyxosPilotGetPointInterface()

Determines the interface being used by a particular Pyxos Point. The Pyxos Point is specified by its timeslot.

Syntax:

```
const PyxosPilotPointInterface
      *PyxosPilotGetPointInterface(PyxosTimeslot timeslot);
```

Remarks: This function returns a pointer to the interface associated with the Pyxos Point using the specified timeslot. If the designated timeslot has no interface associated with it, the function will return NULL.

The *Example Code for Sending and Receiving PNVs* section on page 73 includes example code that uses **PyxosPilotGetPointInterface()**to determine whether a Pyxos Point is using a certain interface, and then update a PNV on the Pyxos Point if it is using that particular interface.

PyxosPilotGetNumberOfPoints()

Determines the number of Pyxos Points on the network that have been allocated timeslots, and the number of Pyxos Points on the network whose interfaces have been specified.

Syntax:

```
void PyxosPilotGetNumberOfPoints(int *pNumberOfAssociatedPoint,
                                 int *pNumberOfPointsWithInterfaces);
```

Remarks: The **pNumberOfAssociatedPoint* parameter returned by the function indicates the number of Pyxos Points that have been allocated timeslots. The **pNumberOfPointsWithInterfaces* parameter indicates the number of Pyxos Points with defined interfaces.

PyxosPilotSetPointOnline()

Sets the specified Pyxos Point online. The Pyxos Point is specified by its timeslot.

Syntax:

```
PyxosSts PyxosPilotSetPointOnline(PyxosTimeslot timeslot);
```

Remarks: This function must be called each time a Point has been configured or reconfigured. The Pilot application can update PNV values on the Point prior to setting it online with this function, but the updated values will not be propagated until the Pilot sets the Point online. The Point will not send any of its PNVs until the Point is set online. This function must be called after the Point interface has been set.

If the Point is an unhosted Point, the Pilot will set the Point online by configuring the Point's I/O register. The output values are initialized to 0 by **PyxosPilotSetPointInterface()**. To set the initial output values to 1, call **PyxosPilotUpdateUnhostedPointIo()**to update the desired outputs after calling

PyxosPilotSetPointInterface(), and before setting the Point online. When the Point is set online, these output values will be sent to the Point at the same time as the I/O configuration.

The **PyxosPilotSetPointOnline()** function may return the following errors:

- **PyxosSts_PointNotFound**: Returned when the *timeslot* parameter is greater than or equal to **PYXOS_MAX_POINTS**.
- **PyxosSts_PointNotReady**: Returned when Point has not been successfully configured (or reconfigured).
- **PyxosSts_PointAlreadyOnline**: The Point is already online.
- **PyxosSts_PointInterfaceHasNotBeenSet**: The Point's interface has not been set.

If **PyxosPilotSetPointOnline()**returns **`PyxosSts_Good`** the API will call **PyxosPilotSetPointOnlineCompleted()** when the online status has been acknowledged or timed out.

It is not possible to set a Point offline, other than to reset the Point and reconfigure it.

Callbacks for Registering Points

This section describes the callbacks that you will use when registering the Pyxos Points on a network. The following callbacks are also used during reconfiguration and replacement: **PyxosPilotPointConfigured()**, **PyxosPilotPointConfigurationFailed()**, and **PyxosPilotSetPointOnlineCompleted()**.

PyxosPilotRegistrationRequestReceived()

Handles a registration request received from a Pyxos Point.

Syntax:
```
void PyxosPilotRegistrationRequestReceived
                              (PyxosTimeslot timeslot,
                               const PyxosUniqueId *pUniqueId);
```

Remarks: When a Pyxos Point application announces itself to the Pilot, this callback is triggered. Pyxos Points using the automatic registration method announce themselves on startup after calling **PyxosPointInit()**, and continue to do so until the Pilot has configured them. Pyxos Points using the manual registration method typically announce themselves when the Pyxos Point's **Join** button is depressed. Alternatively, a Pyxos Point may also announce itself by calling **PyxosPointSendUniqueId()**. Pyxos Points using the hardwired registration method announce themselves when they call **PyxosPointAnnounceTimeslot()**.

Typically, you should call **PyxosPilotAllocateTimeslot()** to assign the Pyxos Point a timeslot from this callback. It is common for this function to be called multiple times for a given Pyxos Point, because the Point will continue to announce itself until it has been successfully configured. If the Pilot has already assigned a timeslot for the Pyxos Point, the **PyxosPilotAllocateTimeslot()** function will do

nothing except return **PyxosSts_AlreadyAllocatedToPoint**. This usually means that the Pilot is still configuring the Point, and can be ignored. If the Pilot fails to configure the Point, error recovery can be performed from the **PyxosPilotPointConfigurationFailed()** callback.

The Pilot must call **PyxosPilotAllocateTimeslot()**, even when the Point uses the hardwired registration method.

The following example allocates a timeslot from the **PyxosPilotRegistrationRequestReceived()** callback:

```
/* With 32 timeslots, 40 frames = 1 second */
#define CONFIGRUATION_TIMEOUT 40
void PyxosPilotRegistrationRequestReceived
         (PyxosTimeslot timeslot,
          const PyxosUniqueId *pUid)
{
   PyxosPilotAllocateTimeslot(pUid, PYXOS_TIMESLOT_ANY,
                              CONFIGRUATION_TIMEOUT,
                              False);
}
```

PyxosPilotPointConfigured()

Provides notification that a Point has been successfully configured.

Syntax:

```
void PyxosPilotPointConfigured
     (PyxosTimeslot timeslot, int reserved);
```

Remarks: This is called when a Point has been successfully configured. This may be the result of the Pilot calling **PyxosPilotAllocateTimeslot()**, **PyxosPilotConfigurePoint()** or **PyxosPilotReplacePoint()**. This is also called when a Point using hardwired registration has been reset and reconfigured itself.

The second parameter, **reserved**, is reserved for future use and should be ignored.

Typically, the next registration steps to complete after a Pyxos Point has been configured are to set the Point's interface if it has not already been set, and then to set the Point online. See the *Example Code for Automatic and Manual Registration* section on page 62 for example code you can use to perform these tasks.

PyxosPilotPointConfigurationFailed()

Provides notification the API failed to configure a Point.

Syntax:

```
void PyxosPilotPointConfigurationFailed
     (PyxosTimeslot timeslot, int reserved);
```

Remarks: This is called when the configuration process has timed out. The timed out process may have been initiated by the **PyxosPilotAllocateTimeslot()**, **PyxosPilotConfigurePoint()** or **PyxosPilotReplacePoint()** functions.

Typically, the Pilot application retries the configuration by resetting the Point and then reconfiguring it in the reset completion handler after a configuration fails. See the *Example Code for Automatic and Manual Registration* section on page 62 for an example of how you could use this callback.

PyxosPilotFreeTimeslotCompleted()

Provides notification that a timeslot has been freed.

Syntax:

```
void PyxosPilotFreeTimeslotCompleted (PyxosTimeslot timeslot,
                                      const PyxosUniqueId *pUniqueId,
                                      Bool success);
```

Remarks: The *success* parameter indicates whether or not the Point has been successfully updated. The timeslot is available for reallocation once the API calls this function.

PyxosPilotSetOnlineComplete()

Provides notification that the Pilot has successfully set a Point online, or provides notification that an attempt to set a Pyxos Point online has timed out.

Syntax:

```
void PyxosPilotSetPointOnline(PyxosTimeslot timeslot,
                              Bool success);
```

Remarks: If the *success* parameter is True, the Point has been successfully configured and set online. If it is False, the Pilot was unable to set the Point online, and the Pilot application should initiate error recovery as described in the *Detecting, Reporting, and Correcting Communication Errors* on page 77.

Example Code for Event Handling, and Registering Pyxos Points

This section provides example code that will help you get started when programming your Pilot application to perform the tasks described previously in the chapter. Your application will need to include the necessary Pyxos files to reference the Pilot API, as described in Chapter 5, *Including the Pyxos FT API*.

Example Code for Automatic and Manual Registration

In this example, the **main()** function performs all of the initialization and implements the main control loop. The bulk of the code to discover Points, allocate timeslots, and identify Point interfaces is implemented in callbacks.

The implementations of some of the functions used in this example are shown in other sections of this chapter, as noted. Some application-specific functions are not provided.

Main Program

This section provides the initialization and main control loop. The main control loop calls the following application functions that are not described in this section.

- **HostIoInit()**: Performs any necessary host I/O initialization, including calling **psInit()**. The implementation of this function is not provided.
- **RestorePointsFromNvData()**. Reads the unique IDs of Points previously registered from non-volatile data and allocates timeslots for these Points. This function is described in the *Reset Example* section on page 90.
- **CheckPoints()**. Checks to make sure that Points that have previously been discovered and configured are still connected. This function is described in the *Checking Point Configuration* section on page 86.
- **CheckPilotConfiguration()**. Checks the Pilot's configuration. This function is described in the *Checking Pilot Configuration* section on page 84.

The **pointStatus** variable that is initialized in the **main()** function is described in the *Checking Point Configuration* section on page 86.

```
int main(void)
{
   /* Initialize global data */
   memset(pointStatus, 0, sizeof(pointStatus));

   /* Initialize I/O including Pyxos Serial Driver */
   HostIoInit();

   /*Initialize Pilot API */
   PyxosPilotInit();

   /* Read non-volatile data and reconfigure any Points that
   were previously registered
    */
   RestorePointsFromNvData();

   /* This is the main control loop, which runs forever. */
   while (TRUE)
   {
      /* Run the Pilot API event handler to process Pyxos
      Events.
       */
      PyxosPilotEventHandler();

      /* Periodically check to make sure that all Points are
      properly configured.
       */
      CheckPoints();

      /* Periodically check to make sure the Pilot is properly
      configured
       */
      CheckPilotConfiguration();
   }
```

```
}
```

Discovering Points and Allocating Timeslots

The Pilot first discovers the presence of a Pyxos Point via the **PyxosPilotRegistrationRequestReceived()** callback, which is called by the Pyxos API whenever a Pyxos Point sends its unique ID to the Pilot. If the Pyxos Point has not already had a timeslot allocated to it, the Pilot application allocates a timeslot in this callback. The next step in the process is carried out by the **PyxosPilotPointConfigured()** callback, which is called as soon as the Point has been successfully configured.

```
/* The configuration timeout depends on how long it takes the
Point to reset.  This is very specific to the Point
application and host processor.  This parameter is expressed
in terms of frames, and so it also depends on the value of
PYXOS_NUM_TIMESLOTS.  In this example, allow 1 second for the
Point to reset and be ready to be reconfigured, and assume 32
timeslots, resulting in a frame every 25 Msec.
1000Msec/25Msec per frame equals 40 frames.
 */
#define CONFIGURATION_TIMEOUT 40 // One second with 32
                                 // timeslots.

/* A registration request has been received from a Point.
Allocate a timeslot if one has not already been allocated.
The interface is set and the Point is placed online in the
PyxosPilotPointConfigured() callback.
 */
void PyxosPilotRegistrationRequestReceived(
          PyxosTimeslot timeslot,
          const PyxosUniqueId *pUniqueId)
{
   /* Allocate a timeslot for the Point.  If it already has a
   timeslot, the API will simply return an error here, and
   there is no need to do anything.
    */
   PyxosPilotAllocateTimeslot(pUniqueId, PYXOS_TIMESLOT_ANY,
                              CONFIGURATION_TIMEOUT, FALSE);
}
```

The **PyxosPilotPointConfigured()** callback is called after the Point has been successfully configured with the **PyxosPilotAllocateTimeslot()** function. The next step in the registration process is to associate a program interface with the Point. During the configuration process, the Pilot can read the Point's program ID with **PyxosPilotGetProgramId()**. The Pilot application can use the Point's program ID to identify the Point's interface. When using manual registration, the Pilot typically requires user input as well as the program ID to determine the Pyxos Point's interface. For example, all unhosted Points have the same program ID (0), but may support different interfaces. In the example below, the **GetPointInterfaceFromProgramId()** function determines the Point's interface with the program ID. If any Points are defined to use manual registration, this function needs to be updated to set the interface correctly based on user input.

After the application has associated a program interface with the Point, the Pilot must set the Point online.

If the configuration fails, the API calls the **PyxosPilotPointConfigurationFailed()** callback and the Pilot application should retry the operation. See the *Detecting, Reporting, and Correcting Communication Errors* section later in this chapter for more information regarding correcting configuration errors.

The **UpdateNvData**() function is used to update non-volatile data to record the unique IDs of Points with allocated timeslots. This function is described in the *Reset Example* section on page 90. The **UpdatePointStatus()** function is used to maintain connection information about a Point, and is described in the *Checking Point Configuration* section on page 86.

```
/* This structure is used to define attributes for each Point.
 */
typedef struct PointTypeDefinition
{
   /* The interface definition defined by the Pyxos FT
      Interface Developer.  These names can be found in the
      Pilot's "MyPyxosApplication.h" file.
    */
   const PyxosPilotPointInterface *pInterface;

   /* True if the Point has a host processor */
   Bool  isHosted;

   /* True if the Point uses manual registration.*/
   Bool  usesManualRegistration;
} PointTypeDefinition;

/* This array contains an entry for each type of Point
supported by the Pilot application.
 */
const PointTypeDefinition PointTypeDefinitions[] =
{
 {&ACME_ActuatorPointInterface, TRUE, FALSE},
 {&ACME_SensorPointInterface, TRUE, FALSE},
   .
   .
   .
 };

#define NUM_PROGRAM_INTERFACES \
   (sizeof(pointTypeDefinitions)/sizeof(PointTypeDefinition))

const PyxosPilotPointInterface *
   GetPointInterfaceFromProgramId(const PyxosProgramId
                                  *pProgramId)
{
   int i;
   for (i = 0; i < NUM_PROGRAM_INTERFACES; i++)
   {
      if (memcmp(pProgramId,
             pointTypeDefinitions[i].pInterface->pid,
             sizeof(PyxosProgramId))== 0) {
```

```
            if (pointTypeDefinitions[i].usesManualRegistration)
            {
               /* find the definition based on user input. */
            }
            return PointTypeDefinitions[i].pInterface;
         }
       }
       return NULL;
   }

   /* The Point has been successfully configured.  This may be
   the result of timeslot allocation, reconfiguration, or
   replacement.  The next step is to set the program interface
   (if not set already) and then set the Point online.

    */
   void PyxosPilotPointConfigured(PyxosTimeslot timeslot,
                                  int reserved)
   {
     /* Update the NV data with the current Points. */
     UpdateNvData();
     const PyxosPilotPointInterface *pInterface =
        PyxosPilotGetPointInterface(timeslot);
     if (pInterface == NULL) {
        /* The Point's interface has not yet been set. */
        PyxosProgramId pid;
        if (PyxosPilotGetProgramId(timeslot, &pid) ==
              PyxosSts_Good) {
           pInterface = getPointInterfaceFromProgramId(pid);
        }
        if (pInterface == NULL) {
           /* Don't recognize this one -  Free the timeslot. */
           UpdatePointStatus(timeslot, FALSE);
           PyxosPilotFreeTimeslot(timeslot);
        } else {
           PyxosPilotSetPointInterface(timeslot, pInterface);
        }
     }
     if (pInterface != NULL){
         /* The Point has a known interface, must set it online
         as the final stage of registration or reconfiguration.
         */
       PyxosPilotSetPointOnline(timeslot);
     }
}

/* The configuration of the Point has failed. */
void PyxosPilotPointConfigurationFailed(PyxosTimeslot timeslot,
                                        int reserved)
{
   UpdatePointStatus(timeslot, FALSE);
   /* Reset the Point to try and recover it. */
   PyxosPilotResetPoint(timeslot);
}

/* The Point has either been successfully put online, or it has
   timed out.
```

```
 */
void PyxosPilotSetPointOnlineCompleted(PyxosTimeslot timeslot,
                                       Bool success)
{
   UpdatePointStatus(timeslot, success);
   if (!success) {
      /* Reset the Point to try and recover it. */
      PyxosPilotResetPoint(timeslot);
   }
}
```

The **PyxosPilotFreeTimeslotCompleted()** callback is called by the API after the timeslot is freed. This function updates the non-volatile data to reflect the current allocation of timeslots.

```
/*  Timeslot has been successfully freed. */
void PyxosPilotFreeTimeslotCompleted(PyxosTimeslot timeslot,
                                     const PyxosUniqueId
                                     *pUniqueId,
                                      Bool success)
{
   UpdateNvData();
}
```

Pyxos Pilot Example Code for Hardwired Registration

When configuring a network with hardwired Points, the Pilot application can allocate the timeslots without talking to the Points beforehand. The function below is used to allocate the hardwired timeslots. This function would be called from the main program after the **PyxosPilotInit()** function is called, but before the main control loop.

```
void InitHardwiredPoints(void)
{
   PyxosTimeslot timeslot;
   for (timeslot = 0; timeslot < NUM_HARDWIRED_POINTS;
         timeslot++) {
      if (hardwiredPoints[timeslot] != NULL) {
         /* Unique ID not required when using hardwired Points.
          */
         PyxosPilotAllocateTimeslot(NULL, timeslot,
                                    CONFIGURATION_TIMEOUT, TRUE);
      }
   }
}
```

The **PyxosPilotPointConfigured()** callback will be called after the Pilot has confirmed that the Point is ready. The implementation of **PyxosPilotPointConfigured()** callback is the same as the one given in the previous section.

Reading and Writing PNVs

Your Pilot application can read and write Pyxos Point PNVs with the functions and callbacks described in this section.

In some cases, an attempt to read or write a PNV on a Pyxos Point will fail even after the Pyxos FT protocol automatically retries the operation for up to **PYXOS_MAX_UNACKD_FRAME_COUNT** frames. If an operation fails persistently, then you should try resetting and reconfiguring the Pyxos Point. For more information on this, see *Detecting, Reporting, and Correcting Communication Errors* on page 77.

PyxosPilotUpdatePnv()

Writes a value to an input PNV on a Pyxos Point.

Syntax:

```
PyxosSts PyxosPilotUpdatePnv(PyxosTimeslot timeslot,
                             PyxosPci pci, const Byte *pData);
```

Remarks: Reference the Pyxos Point containing the PNV to be updated by its timeslot (*timeslot* parameter). Reference the PNV to be updated by its Pyxos Chip index (*pci* parameter).

To use this function, the timeslot must have already been allocated to the Pyxos Point with **PyxosPilotAllocateTimeslot()**, and the interface for the Pyxos Point must have already been specified with **PyxosPilotSetPointInterface()**. The PNV being updated must be an input PNV. The PNV may be updated before the Point is set online. However, the value will not be propagated to the Point until after the Pilot has set the Point online.

Specify the new value to be written to the PNV with the *pData* parameter. If *pData* is NULL, the function will resend the PNV's current value.

When the Pilot application calls this function, the Pilot API will buffer the input value and mark it to be sent. Values are sent at the end of each call to **PyxosPilotEventHandler()**on a round-robin basis. If the application calls **PyxosPilotUpdatePnv()**again before the value has been sent and before the update has started, the old value will be overwritten and the new value will be sent in its place. If the Pilot application is currently sending the previous value, it will complete the update of the original value (which, depending on the size of the PNV, may require multiple frames) and schedule the update of the new value at a later time.

The order in which PNVs are sent out on the network is not necessarily the same as the order that your application calls **PyxosPilotUpdatePnv().**

After the update for the last bytes of the PNV value has been acknowledged, the Pilot API calls the **PyxosPilotUpdatePnvCompleted()** callback with the *success* parameter set to True. If an update is sent and times out before being acknowledged, the Pilot API calls the **PyxosPilotUpdatePnvCompleted()** callback with the *success* parameter set to False. You can specify the timeout value with the **Number of Frames Before Timeout** property on the Pilot Application Options dialog when you configure the Pilot with the Pyxos FT Interface Developer. This corresponds to the **PYXOS_MAX_UNACKD_FRAME_COUNT** macro in the **MyPyxosApplication.h** file.

All PNVs are sent in big endian format. If your host processor is a little endian processor, your application may have to swap the bytes of multi-byte fields prior

to calling **PyxosPointUpdatePnv()**. Macros defined in the **Platform.h** file are provided to aid in this transformation. See the *Modifying the Platform.h File* section on page 115 for more information on this.

PyxosPilotUpdateUnhostedPointIo()

Updates the digital I/O values on an unhosted Point.

Syntax:

```
PyxosSts PyxosPilotUpdateUnhostedPointIo(PyxosTimeslot timeslot,
                                         Byte setMask,
                                         Byte clearMask);
```

Remarks: Use this function to update the digital outputs on an unhosted Point. Each digital output is represented by a bit position using one of the bitmasks described in Table 14. See the *Pyxos FT EVK User's Guide* for more information on the I/O pins listed in Table 14.

Table 14 PyxosPilotUpdateUnhostedPointIo() Bitmasks

Bitmask	Description
PYXOS_IO_VALUE_MASK_DIO_0	The I/O value for the DIO-0 pin.
PYXOS_IO_VALUE_MASK_DIO_1	The I/O value for the DIO-1 pin.
PYXOS_IO_VALUE_MASK_DIO_2	The I/O value for the DIO-2 pin.
PYXOS_IO_VALUE_MASK_DIO_3	The I/O value for the DIO-3 pin.

Typically, the Point interface file defines aliases for the I/O value bitmasks. To set a digital output, enter the corresponding bitmask as the *setMask* parameter. To clear a digital output, enter the corresponding bitmask as the *clearMask* parameter. Using these parameters, you can set one or more digital outputs and clear one or more other digital outputs with a single call to the function. Any bits that are 0 in both the *setMask* and *clearMask* parameters will receive the same value that the Pilot sent the last time the I/O was updated. Initially, the value of each digital output is 0.

After the update has been acknowledged, the **PyxosPilotUpdatePnvCompleted()** callback will be called with the *success* parameter set to True, and with the *pci* parameter set to **PYXOS_REGI_UNHOSTED_IO**. If an update times out before it is acknowledged, the **PyxosPilotUpdatePnvCompleted()** callback will be called with the *success* parameter set to False. You can specify the timeout value with the **Number of Frames Before Timeout** property on the Pilot Application Options when you configure the Pilot application with the Pyxos FT Interface Developer. This corresponds to the **PYXOS_MAX_UNACKD_FRAME_COUNT** macro in the **MyPyxosApplication.h** file.

PyxosPilotPollRegister()

Polls the current value of a Pyxos Chip register on a Point.

Syntax:

```
PyxosSts PyxosPilotPollRegister(PyxosTimeslot timeslot,
                                PyxosPci regIndex);
```

Remarks: You can reference the Pyxos Point containing the register to be polled by its timeslot (*timeslot* parameter), and you can reference the register to be polled by its Pyxos Chip index (*regIndex* parameter). To use this function, the timeslot must have already been allocated to the Pyxos Point with **PyxosPilotAllocateTimeslot()**, and the interface for the Pyxos Point must have already been set with **PyxosPilotSetPointInterface()**. Only registers defined by the **PYXOS_STANDARD_REGISTERS** macro may be polled, unless the Point is unhosted, in which case the **PYXOS_REGI_UNHOSTED_IO** register may be polled. See *Accessing Pyxos Registers* on page 162 for more information about the register definitions, including the **PYXOS_STANDARD_REGISTERS** macro.

This function marks the register as needing to be polled. Poll requests are sent at the end of each call to **PyxosPilotEventHandler()**, along with all other pending network updates. Updates and poll requests compete on a round-robin basis. However, if any poll has been requested and there are no updates in progress, the Pyxos API will schedule a poll request (leaving only one data slot for updates). If more than 4 Pyxos Chip values need to be polled, some will be deferred until the next poll cycle.

If the application calls this function and a poll for the requested register is outstanding, the value will be polled only once, the next time **PyxosPilotEventHandler()** is called.

After the poll request has been acknowledged, the **PyxosPilotPollRegisterCompleted()** callback will be called with the *success* parameter set to True. This indicates that the Pyxos Point containing the register has received the poll request. The value should be received some time later, via the **PyxosPilotPnvUpdateOccurred()** callback.

If a poll request is sent and times out before it is acknowledged, the **PyxosPilotPollRegisterCompleted()** callback will be called with the *success* parameter set to False. You can specify the timeout value with the **Number of Frames Before Timeout** property on the Pilot Application Options when you configure the Pilot application with the Pyxos FT Interface Developer. This corresponds to the **PYXOS_MAX_UNACKD_FRAME_COUNT** macro in the **MyPyxosApplication.h** file. A successful completion event only means that the Point has acknowledged the poll request, not that it has sent the requested data.

To poll the digital values on an unhosted Point, call **PyxosPilotPollRegister()** with the *pci* parameter set to **PYXOS_REGI_UNHOSTED_IO**. The Point will send the entire 4-byte I/O register, including the current input values. This data is represented as a bitmask, which can be obtained using the **PYXOS_GET_UNHOSTED_IO_DATA** macro. For example code demonstrating this, see the *Example Code For Updating and Monitoring Unhosted Point I/O* section on page 74.

NOTE: Polling the **PYXOS_REGI_UID** register will cause the Point to send its unique ID, which will cause the Pilot API to call the **PyxosPilotRegistrationRequest()** callback instead of the **PyxosPilotPnvUpdateOccurred()** callback. The Pilot should not poll the **PYXOS_REGI_UID** register, as this may interfere with the configuration of the Point.

PyxosPilotGetPnvValue()

Gets the cached value of an input PNV. The function returns a pointer to the cache which contains the last value set with **PyxosPilotUpdatePnv()**.

Syntax:

```
PyxosSts PyxosPilotGetPnvValue(PyxosTimeslot timeslot,
                               PyxosPci pci,
                               const Byte **ppPnvValue,
                               PyxosPnvSize *pSize);
```

Remarks: You can reference the Pyxos Point containing the PNV by its timeslot (*timeslot* parameter), and you can reference the PNV by its Pyxos Chip index (*pci* parameter). To use this function, the timeslot must have already been allocated to the Pyxos Point with **PyxosPilotAllocateTimeslot()**, and the interface for the Pyxos Point must have already been set with **PyxosPilotSetPointInterface()**.

This function returns a pointer to the value of the PNV that is cached by the Pilot API as the ***ppPnvValue* parameter. The application can retrieve the value from the cache, but it must not modify the value. The returned pointer may be invalidated by any subsequent calls into the Pyxos FT API. This function returns the size of the value, in bytes, using the **pSize* parameter.

If the value is an output value, this function may return **PyxosSts_ValueNotAvailable**. This indicates that either the Point has not sent any updates for this value, or the cached value is currently inconsistent. A cached value is considered to be inconsistent if the PNV is larger than 4 bytes and the Point has updated some of those bytes, but not others. If the value is inconsistent, the Point should send the remaining bytes within several frames.

The Pilot application must not poll the unique ID of a Point, as this may interfere with the registration process. If the Pilot application does poll the unique ID register, the Pilot API will call the **PyxosPilotRegistrationRequestReceived()** callback when the UID arrives, rather than the **PyxosPilotPnvUpdateOccurred()** callback.

PyxosPilotIsPnvUpdatePending()

Determines if there are any updates for a PNV that have been cached by the Pyxos API but have either not yet been sent to the Point or have not yet been acknowledged by the Point.

Syntax:

```
Bool PyxosPilotIsPnvUpdatePending(PyxosTimeslot timeslot,
                                  PyxosPci pci);
```

Remarks: The Pyxos Point containing the PNV to be checked must be referenced by its timeslot (*timeslot* parameter). The PNV must be referenced by its Pyxos Chip index (*pci* parameter) as defined in the Point's interface definition file. The function will return True if the PNV has any updates pending, and False otherwise. You cannot call this function on an output PNV.

Callbacks for Reading and Writing PNVs

This section describes the callbacks that are called in response to PNV updates and polls in more detail.

PyxosPilotUpdatePnvCompleted()

Provides notification that a PNV update request has either successfully completed, or timed out. This is called after **PyxosPilotUpdatePnv()** has been called.

Syntax:

```
void PyxosPilotUpdatePnvCompleted(PyxosTimeslot timeslot,
                                  PyxosPci pci,
                                  Bool success);
```

Remarks: This callback is called when a PNV update initiated by **PyxosPilotUpdatePnv()**has been acknowledged, or has timed out. You can use the *timeslot* parameter to identify the Pyxos Point affected by the update, and you can use the Pyxos Chip index (*pci* parameter) to identify the PNV that was updated. The *success* parameter will be set to True if the update has been acknowledged by the Pyxos Point, or False if the request has timed out.

The timeout is expressed as a function of the frame count since the update was sent on the network. You can specify this value with the **Number of Frames Before Timeout** property on the Pilot Application Options dialog when you configure the Pilot application with the Pyxos FT Interface Developer. This corresponds to the **PYXOS_MAX_UNACKD_FRAME_COUNT** macro in the **MyPyxosApplication.h** file.

PyxosPilotPollRegisterCompleted()

Provides notification that a register poll request has successfully completed or timed out. This is called after **PyxosPilotPollRegister()** has been called, and the poll initiated by the function has been acknowledged or timed-out.

Syntax:

```
void PyxosPilotPollRegisterCompleted (PyxosTimeslot timeslot,
                                      PyxosPci regIndex,
                                      Bool success);
```

Remarks: This is called when a poll request initiated by **PyxosPilotPollRegister()** has either been acknowledged or has timed out. You can use the *timeslot* parameter to identify the Pyxos Point containing the register that was being polled. You can use the Pyxos Chip index (*regIndex* parameter) to identify the register.

The *success* parameter will be True if the poll request has been acknowledged by the Pyxos Point, or False if the request has timed out.

The timeout is expressed as a function of the frame count since the poll request was sent on the network. You can specify this value with the **Number of Frames Before Timeout** property on the Pilot Application Options dialog when you configure the Pilot application with the Pyxos FT Interface Developer. This corresponds to the **PYXOS_MAX_UNACKD_FRAME_COUNT** macro in the **MyPyxosApplication.h** file. A successful completion event only means that the Point has acknowledged the poll request, not that it has sent the requested data.

PyxosPilotPnvUpdateOccurred()

Provides notification that a PNV value has been updated.

Syntax:

```
void PyxosPilotPnvUpdateOccurred(PyxosTimeslot timeslot,
                                 PyxosPci pci, const Byte
                                 *pPnvValue, Byte length);
```

Remarks: You can use the *timeslot* parameter to identify the Pyxos Point containing the PNV that has been updated. You can use the Pyxos Chip index (*pci* parameter) to identify the PNV. The *pPnvValue* parameter is a pointer to the updated PNV value.

The PNV updates that cause this callback to be triggered can be initiated by the Pyxos Point, or by the Pilot application when it calls **PyxosPilotPollRegister()**.

All PNVs are sent in big endian format. If your host processor is a little endian processor, your application may have to swap the bytes of multi-byte fields prior to calling **PyxosPointUpdatePnv()**. Macros defined in the **Platform.h** file are provided to aid in this transformation. See the *Modifying the Platform.h File* section on page 115 for more information.

When an unhosted Point sends an update containing the value of its digital inputs, the Pyxos API calls this callback with the *pci* parameter set to **PYXOS_REGI_UNHOSTED_IO**, and the data pointed to by the *pPnvValue* parameter includes the entire I/O register. To retrieve only the data bits the application can use, use the **PYXOS_GET_UNHOSTED_IO_MASK** macro when you send the update. The **PYXOS_GET_UNHOSTED_IO_MASK** macro takes *pPnvValue* as its single argument.

Example Code for Sending and Receiving PNVs

This section shows how the Pyxos Pilot sends and receives PNV updates. In this example, the Pilot supports two Points, an actuator and a sensor. The sensor has a single output PNV called ACME_SENSOR_SWITCH, and the actuator has a single input PNV called ACME_ACTUATOR_LIGHT. Both are of type **SNVT_switch**. Whenever the sensor updates the ACME_SENSOR_SWITCH, the Pilot sends the update to the actuators' ACME_ACTUATOR_LIGHT.

The **UpdatePointStatus()** and **ValidatePointConfiguration()** functions are used to detect and correct communication errors, and are described in the *Checking Point Configuration* section on page 86.

The **actuatorTimeslot** global variable represents the timeslot of the actuator Point. This example assumes that the actuatorTimeslot is a global variable set by the Pilot application when it set the actuator's Point interface.

Whenever a Point sends an update to the Pilot, the API calls the **PyxosPilotPnvUpdateOccurred()** callback.

```
void PyxosPilotPnvUpdateOccurred(PyxosTimeslot timeslot,
                                 PyxosPci pci,
                                 const Byte *pPnvValue,
                                 Byte length)
{
   UpdatePointStatus(timeslot, TRUE);
   if (pci == PYXOS_REGI_CONFIG) {
      ValidatePointConfiguration(timeslot, pPnvValue);
   } else {
      /* Process PNV updates as necessary */
      if (PyxosPilotGetPointInterface(timeslot) ==
           &ACME_SensorPointInterface) {
         switch(pci) {
            case ACME_SENSOR_SWITCH:
               /* Update the actuator's light. */
               PyxosPilotUpdatePnv(actuatorTimeslot,
                                   ACME_ACTUATOR_LIGHT,
                                   pPnvValue);

               break;
         }
      }
   }
}
```

Example Code For Updating and Monitoring Unhosted Point I/O

This section shows how the Pyxos Pilot updates and monitors I/O values on an unhosted Point. In this example, the Pilot supports two unhosted Points, an actuator and a sensor. The sensor supports two switches, called **ACME_SENSOR_SWITCH1_INPUT_MASK** and **ACME_SENSOR_SWITCH2_INPUT_MASK**. The actuator supports two LEDs, which are called **ACME_ACTUATOR_LED1_OUTPUT_MASK** and **ACME_ACTUATOR_LED2_OUTPUT_MASK**. Whenever the Pilot receives an update from the sensor Point it updates LED 1 to the value of switch 1, and LED 2 to the value of switch 2.

The **UpdatePointStatus()** and **ValidatePointConfiguration()** functions are used to detect and correct communication errors and are described in the *Checking Point Configuration* section on page 86.

The **sensorTimeslot** and **actuatorTimeslot** global variables represent the timeslots of the sensor and actuator Points respectively. This example assumes that these global variables are set by the Pilot application when it registers the Points.

Whenever the sensor sends an update, the API calls the **PyxosPilotPnvUpdateOccurred()** callback. The switch values are stored in the I/O register, and therefore the Pyxos Chip index (pci) is **PYXOS_REGI_UNHOSTED_IO**.

The **setOutput()** utility function is used to set an output on the specified Point.

```
PyxosTimeslot sensorTimeslot = PYXOS_NO_TIMESLOT;
PyxosTimeslot actuatorTimeslot = PYXOS_NO_TIMESLOT;

/* Set or clear the output indicated by the outputMask. */
void setOutput(PyxosTimeslot timeslot, Byte outputMask, Bool on)
{
   if (on) {
     /* Set output, don't clear anything */
     PyxosPilotUpdateUnhostedPointIo(timeslot, outputMask, 0);
   } else {
     /* Clear led, don't set anything */
     PyxosPilotUpdateUnhostedPointIo(timeslot, 0, outputMask);
   }
}

void PyxosPilotPnvUpdateOccurred(PyxosTimeslot timeslot,
                                 PyxosPci pci,
                                 const Byte *pPnvValue,
                                 Byte length)
{
   UpdatePointStatus(timeslot, TRUE);
   if (pci == PYXOS_REGI_CONFIG) {
     ValidatePointConfiguration(timeslot, pPnvValue);
   } else {
     if (timeslot == sensorTimeslot) {
       if (pci == PYXOS_REGI_UNHOSTED_IO) {
         /* Get the I/O values from the PNV */
         Byte ioValue = PYXOS_GET_UNHOSTED_IO_DATA(pPnvValue);

         /* Set or clear led 1 based on value of switch 1 */
         setOutput(
             actuatorTimeslot,
             ACME_ACTUATOR_LED1_OUTPUT_MASK,
             (ioValue & ACME_SENSOR_SWITCH1_INPUT_MASK) != 0);

         /* Set or clear led 2 based on value of switch 2 */
         setOutput(actuatorTimeslot,
                ACME_ACTUATOR_LED2_OUTPUT_MASK,
                (ioValue & ACME_SENSOR_SWITCH2_INPUT_MASK) != 0);
       }
     }
   }
}
```

Reading Network Statistics

The Pilot application connected to a Pyxos FT network will gather network statistics about a Pyxos Point as soon as it is added to the network. This section describes the functions you can use to read and clear these statistics. These

functions are included in both the Point API and the Pilot API. Because the Pilot is responsible for the health of the network, the Pilot application typically uses these functions. However, it is possible to use them from a Pyxos Point application, particularly to aid in debugging or for error isolation purposes in noisy environments.

PyxosReadNetworkStats()

Returns the network statistics maintained by the Pyxos FT Chip for a particular Pyxos Point.

Syntax:

```
PyxosSts PyxosReadNetworkStats(PyxosTimeslot timeslot,
                               Dword *pTotalFrames,
                               Word *pCrcCounter,
                               Word *pMissedSlotCounter);
```

Remarks: The *timeslot* parameter identifies the Point whose network statistics should be returned. The **pTotalFrames* value is updated with the number of frames started since the statistics have been cleared, the **pCrcCounter* value is updated with the number of CRC errors on the specified timeslot, and the **pMissedSlotCounter* value is updated with the number of missed slots for that timeslot. The Pyxos FT Chip logs a CRC error whenever it receives a data (read or write) packet with a CRC error. A missed slot error is logged if the Pyxos FT Chip cannot recognize a valid preamble. This primarily occurs when there is no Point configured in the timeslot. CRC errors and missed slot statistics peg at 0xFFFF. The total number of frames is stored as a 24-bit number that wraps around. You can clear these statistics with **PyxosClearNetworkStats()**. For more information about these statistics see the *Protocol Statistics* section of Chapter 7, *Pyxos FT Protocol.*

If the **PYXOS_REGISTER_BUFFER_SIZE** macro in the **MyPyxosApplication.h** file is set to a value less than 4, calling this function from a callback will fail and return the **PyxosSts_NotAllowed** error.

NOTE: To use this function, the **PYXOS_INCLUDE_NETWORK_STATS_FUNCTIONS** macro must be defined in the **MyPyxosApplication.h** file. This means that you must select the **Include Statistics Functions** check box when you configure the Pilot application with the Pyxos FT Interface Developer.

PyxosClearNetworkStats()

Clears the network statistics collected by the Pyxos FT Chip.

Syntax:

```
PyxosSts PyxosClearNetworkStats(void);
```

Remarks: If the **PYXOS_REGISTER_BUFFER_SIZE** macro in the **MyPyxosApplication.h** file is set to a value less than 4, calling this function from a callback will fail and return the **PyxosSts_NotAllowed** error.

To use this function, the **PYXOS_INCLUDE_NETWORK_STATS_FUNCTIONS** macro must be defined in the **MyPyxosApplication.h file**. This means that you

must select the **Include Statistics Functions** check box when you configure the Pilot application with the Pyxos FT Interface Developer.

Detecting, Reporting, and Correcting Communication Errors

The Pilot application is responsible for detecting, reporting, and (if possible) correcting communication errors. This section describes strategies that the Pilot can use to perform these tasks.

Communication errors may occur for a number of reasons, such as hardware problems, improper configuration, or application errors. Hardware problems cannot be fixed by the Pilot application, but can be reported. In some cases, the Pilot may be able to reset a Pyxos Point to work around an application error. The most common type of problem results from misconfiguration. The most common cause of misconfiguration is due to a Point or the Pilot being reset, for example as a result of a power cycle.

The next section describes the types of problems that will result in communication errors. Following that are sections describing simple strategies you can use to detect and recover from the most commons sorts of errors, and descriptions of the functions and callbacks you can use to do so. Example code is included after these descriptions to demonstrate how you could use those functions and callbacks.

Types of Errors

This section describes the types of communication errors that may be encountered in a Pyxos network.

Hardware Errors

Hardware problems in the wiring or the Points may cause a number of errors. Some of these may be hard errors, and others may be intermittent errors. There is little the Pilot can do other than report the problem in these cases. If a hardware error is suspected, the network statistics functions described previously in this chapter may provide useful information.

Unconfigured Points

If a Pyxos Point becomes unexpectedly unconfigured, it will no longer be able to communicate with the Pilot. The most likely reason for this is that the Pyxos Point was reset. If the Pyxos Point uses hardwired registration, it will automatically configure itself and send a Point Ready command on that timeslot, and the Pilot API will call the **PyxosPilotPointConfigured()** callback so that the Pilot application knows about it. If the Pyxos Point uses automatic registration, it will send the registration request to the Pilot automatically, but only if the Pilot is currently advertising at least one free timeslot. If the Pyxos Point uses manual registration, it will not inform the Pilot.

The Pilot must be able to detect when Points become unconfigured and attempt to reconfigure them, whether or not the Points resend their unique IDs, and must set the Points online after they have been successfully reconfigured.

Transaction ID Mismatches

The Pyxos FT Protocol uses transaction IDs to prevent retries from being treated as new updates. Transaction ID mismatches can occur if the Pilot, the Pyxos Point or both are unexpectedly reset, or if a Pilot transaction times out and is aborted. Communication between the Point and the Pilot cannot resume until the transaction mismatch is resolved. The Pyxos Pilot API can automatically resolve some mismatches, but other mismatches require the Point to be reset. See the *TID Synchronization* section on page 131 for more information concerning transaction ID synchronization.

Hanging Applications

If a Pyxos Point application stops calling **PyxosPointEventHandler()** for any reason, the Pilot will not be able to send application data to it. This could be due to a bug in the application, or some hardware problem that is preventing the application from running properly. Resetting and reconfiguring the Pyxos Point may fix the problem.

Invalid Configuration

It is possible that a static discharge could cause the Pyxos FT Chip configuration to become invalid. Typically the result of this is that the Pyxos Point will reset and become unconfigured. For an unhosted Point, the Pilot must detect that the Point is missing its timeslot, and reconfigure it. For a hosted Point, the host must detect that the Pyxos FT Chip has an invalid configuration and recover it.

The Pilot application must monitor the configuration register of its Points, and if a hosted Pyxos Point becomes unhosted, the Pilot must reset and reconfigure the Point. The problem could also be with the Pilot's configuration. The Pilot must periodically check its own configuration and fix it as necessary, as described in the next section.

Detecting and Correcting Errors

While there are several types of errors that may cause communication problems, there are a few simple things the Pilot can do to detect and, in many cases, correct these failures.

1. The Pilot application must periodically check its own configuration by calling the **PyxosPilotCheckConfiguration()** function. If this function returns an error, the Pilot must call the **PyxosPilotReInitPyxosInterface()** function to fix its configuration. If this fails (because the serial driver can no longer communicate with the Pyxos FT Chip, for example), it may be necessary for the Pilot application to reset the driver or to reset itself (and as a result, reset the Pyxos FT Chip). The recovery action required to re-establish communication with the Pyxos FT Chip may be dependent on the implementation of the Pyxos Serial Driver.

2. Periodically poll the configuration register of every registered Pyxos Point, and validate the Point's type (hosted or unhosted).

    a. If the poll request is acknowledged, then the Pyxos Point is configured and the Pilot can communicate with the Point. If it is not acknowledged, the Point may be unconfigured or there may be a transaction ID mismatch. The Pilot must reset the Point to clear a possible transaction ID problem, and then reconfigure the Pyxos Point. However, success does not necessarily mean that the Point is in a valid state. That can only be determined for sure if both the poll is acknowledged and the Point sends a value.

    b. If the Pyxos Point sends the configuration register or any other values in a timely manner, the Pyxos Point must be able to communicate with the Pilot. The Pyxos Point may not be able to send the configuration register right away, if it has a lot of other data to send, so the Pilot must take action only if it does not get any data from the Point for several frames. In that case, there may be a transaction ID mismatch. The Pilot must reset the Pyxos Point and reconfigure it.

    c. If the Pilot receives the configuration register, but the host bit is incorrect, the Pyxos Point is misconfigured. The Pilot must reset the Point and reconfigure it.

3. If the Pilot has persistent errors updating a Pyxos Point's input PNV, the Point may be misconfigured or the application may be hung. The Pilot can reset and reconfigure the Point to try to fix this condition.

4. If reconfiguring the device does not work after a short time, the Pilot should typically provide some error indication, but continue trying to reconfigure the device. This way if the device becomes reconnected later the Pilot will automatically reconfigure it.

In some instances, a Pyxos Point may be replaced or removed from the system due to failure. The Pilot may be able to deduce that a Point has been replaced when it has detected communication errors with a Point and discovers another Point of the same type. However, the Pilot will only find out about the replacement device if there are available timeslots. It may be necessary for the Pilot application to delete a Point, thus freeing its timeslot. However, it is generally advisable to do this only when requested based on operator input. Until told otherwise, the Pilot application should generally keep trying to reconfigure the Point. If automatic replacement is a requirement in your system, make sure that your maximum number of timeslots is at least one greater than the maximum number of points that will be installed in your system at any time, to ensure that replacement points can be discovered.

A simple and reliable method for reconfiguring Points that use the automatic or manual registration methods is to follow the steps listed below. The functions introduced in these steps are described in the next section, *Functions for Resetting and Reconfiguring Pyxos Points*.

1. Reset the Point by calling **PyxosPilotResetPoint()**. If the Point was not already configured this step will do nothing, but it will not cause any harm. If the Point was already configured it must be reset to reconfigure it.

2. In the **PyxosPilotResetPointCompleted()** callback, call **PyxosPilotReconfigurePoint()**to reconfigure the Point, whether the reset initiated in step 1 was successful or not.

3. If the reconfiguration is successful, the API will call the **PyxosPilotPointConfigured()** callback. The Pilot application must then set the Point online.

4. If the reconfiguration failed, the API will call the **PyxosPilotPointConfigurationFailed()** callback. This failure may occur for a number of reasons:

    a. The Point was not ready to be reconfigured yet.

    b. The Point is missing.

    c. The Point is already configured but due to a long-standing communication error the Point was never reset.

    The Pilot should continue attempting to reconfigure the Point. However, it may be necessary to reset the Point first, e.g. if the Point was disconnected the last time the Pilot attempted to reset it. A simple approach is to reset the Point in the **PyxosPilotPointConfigurationFailed()** callback, and then depend on the **PyxosPilotResetPointCompleted()** callback to reconfigure the Point.

Points using hardwired registration do not need to be reconfigured. However, the Pilot application must still monitor these Points to determine whether or not they can communicate with the Pilot. If not, the Pilot may need to reset the Points in order to resynchronize the transaction IDs. However, to the Pilot must not reset a hardwired Point too frequently. Resetting an automatic or manual Point prior to reconfiguring it is not a problem since the reset only works if the Point is already configured, and the Pilot is attempting to reconfigure it anyway. Hardwired Points, on the other hand, configure themselves, and if the Pilot keeps resetting the Point because it cannot communicate with the Point it will never come up.

The Pilot application can use the same strategy to re-establish communication with Points using the hardwired registration method as it does for Points using the manual or automatic registration methods. When a communication error is detected, the Pilot application should reset the Point. In the **PyxosPilotResetCompleted()** callback, the Pilot should call **PyxosPilotReconfigurePoint()**. When the Point uses hardwired registration, this function does not update the Point's timeslot (the Point does that), but it does wait for the Point to be ready. When the Point is ready the Pyxos API will call the **PyxosPilotPointConfigured()** callback. If the Point is not ready within the specified timeout, the Pyxos API will call the **PyxosPilotPointConfigurationFailed()** callback, and the application can attempt to reset the Point from that callback. The timeout parameter specified when calling **PyxosPilotReconfigurePoint()** is particularly important for hardwired Points, to ensure that the Pyxos API waits long enough to prevent the application from resetting the Point prematurely

The following sections describe the syntax of the functions and callbacks you will use to handle errors on the Pyxos FT network. Example code demonstrating how to use these functions and callbacks follows.

If a hardwired Point resets on its own, it will configure itself, and the Pilot API will call the **PyxosPilotPointConfigured()** callback. In this case, the Pilot does not need to reset the Point again or call **PyxosPilotConfigurePoint().** Instead it need only call **PyxosPilotSetPointOnline()** to notify the Point that it can start processing data. As a result, the implementation of the **PyxosPilotPointConfiugred()** callback does not have to determine why the Point has been reconfigured, but it is notified that the Point has been reconfigured and that the Point is not currently online.

Functions for Resetting and Reconfiguring Pyxos Points

This section describes the functions you will use when resetting or reconfiguring a Pyxos Point in more detail.

PyxosPilotResetPoint()

Resets a Pyxos Point. You may need to reset a Pyxos Point if it does not respond to PNV polls and updates.

Syntax:

```
PyxosSts PyxosPilotResetPoint(PyxosTimeslot timeslot);
```

Remarks: You can reference the Pyxos Point to be reset by its timeslot (*timeslot* parameter). This function cancels all pending polls and updates to the specified Pyxos Point, and sends a reset command to the Point.

If the Pyxos Point stops acknowledging updates or polls, the application may call this function to reset the Point. When the operation is complete, the Pyxos API calls the **PyxosPilotResetPointCompleted()** callback to indicate that the Pyxos Point has acknowledged the reset, or that the reset has timed out. The application will typically call **PyxosPilotReconfigurePoint()** from the **PyxosPilotResetPointCompleted()** callback. If the Point uses manual or automatic registration, this will reconfigure the Point. If it uses hardwired registration, this will verify that the Point has configured itself.

PyxosPilotReconfigurePoint()

Reconfigures a Pyxos Point and verifies that the configuration of the Point has completed. If the Point uses automatic or manual registration, this function reconfigures the Point by resending its timeslot and then waits for the Point to indicate that it is ready. If the Point uses hardwired registration, this function does not attempt to reconfigure the Point, but waits for the Point to reconfigure itself. This has no effect on the current values of the input PNVs cached by the Pilot for this Pyxos Point.

Syntax:

```
PyxosSts PyxosPilotReconfigurePoint(PyxosTimeslot timeslot,
                                    Word timeout);
```

Remarks: If the Pilot resets a Point it can call this function to reconfigure the Point. Upon completion, the API will either call the **PyxosPilotPointConfigured()** callback or the **PyxosPilotPointConfigurationFailed()** callback, depending on whether or not the Pyxos Point was successfully reconfigured. If the Point is hardwired, this function does not attempt to assign the timeslot to the Point, but does wait until the Point has been successfully configured before calling the **PyxosPilotPointConfigured()** callback.

The *timeout* parameter indicates the number of frames that the Pilot should wait before timing out the configuration process. The time it takes for this process to complete varies depending on the Point's processor and application, especially if the Point has just been reset. Be sure to specify a sufficient timeout. A value of 0 indicates that the Pilot should use the default time period you specified with the **Number of Frames Before Timeout** property on the Pilot Application Options dialog when you configured the Pilot with the Pyxos FT Interface Developer. This value corresponds to the **PYXOS_MAX_UNACKD_FRAME_COUNT** macro in the **MyPyxosApplication.h** file.

It is not possible to reconfigure a point that used automatic or manual registration, and has already been allocated a timeslot. Attempting to do so will fail, and will cause the Pilot API to call the **PyxosPilotReconfigurePointFailed()** callback. If a Point may already be configured, the Pilot application should call the **PyxosPilotResetPoint()** function, and wait for the **PyxosPilotResetPointCompleted()** callback to be called, before calling this function.

PyxosPilotReplacePoint()

Replaces one Point with another of the same type, so that the new Point uses the same timeslot and the same PNV values as the old Point.

Syntax:
```
PyxosSts PyxosPilotReplacePoint(PyxosTimeslot timeslot
                                PyxosUniqueId *pUniqueId,
                                Word timeout);
```

Remarks: Use the *timeslot* parameter to specify the Pyxos Point to replace. This function updates the unique ID of the old Pyxos Point with the unique ID of the specified Pyxos Point, and then reconfigures the Point and waits for the completion, as described in the section above. If the Point is successfully reconfigured, the Pyxos API calls the **PyxosPilotPointConfigured()** callback. If the configuration times out, it calls the **PyxosPilotPointConfigurationFailed()** callback.

Callbacks for Resetting and Reconfiguring Pyxos Points

This section describes some of the callbacks you will use when resetting or reconfiguring a Pyxos Point in more detail. The other callbacks that you will use when reconfiguring a Pyxos Point are the **PyxosPilotPointConfigured()** and

PyxosPilotPointConfigurationFailed() callbacks, both of which were described in the *Callbacks for Registering Points* section.

PyxosPilotResetPointCompleted()

Provides notification that a call to **PyxosPilotResetPoint()** has completed, meaning that a Pyxos Point has been successfully reset or a reset operation has timed out.

Syntax:

```
void PyxosPilotResetPointCompleted(PyxosTimeslot timeslot,
                                   Bool success)
```

Remarks: If the *success* parameter returns True, the Pyxos Point has been successfully reset. If it is False, it is likely that the Point has been removed or it is not currently configured. Typically, a Pilot application calls **PyxosPilotReconfigurePoint()** after a Point has been reset, even if the reset failed, since the most likely cause of failure is that the Point is unconfigured.

Functions for Checking the Pyxos FT Chip Configurations

This section describes the functions you can use to check the configuration of a Pyxos FT Chip. You should do this periodically to ensure that the Pyxos FT Chip has not been reset or otherwise misconfigured, especially if there is no network traffic

PyxosPilotCheckConfiguration()

Checks the configuration of a Pyxos FT Chip. This function must be called periodically, to ensure that the Pyxos FT Chip has not been reset or otherwise misconfigured, especially if there is no network traffic.

Syntax:

```
PyxosSts PyxosPilotCheckConfiguration(void);
```

Remarks: This function reads the configuration of the Pyxos FT Chip and verifies that it is in the expected state. If the configuration of the Pyxos FT Chip is valid, the function returns **PyxosSts_Good**. Otherwise, the function returns **PyxosSts_BadConfiguration**. The function may also return any errors returned by the Pyxos serial driver.

If this function returns anything other than **PyxosSts_Good**, you must call **PyxosPilotReInitPyxosInterface()** to correct the Pyxos FT Chip's configuration. If this is not successful, it may be necessary to reset the processor and the Pyxos FT Chip to re-establish communication with the Pyxos FT Chip.

PyxosPilotReInitPyxosInterface()

Re-initializes a Pyxos FT Chip. This function must be called anytime **PyxosPilotCheckConfiguration()** returns an error.

Syntax:

```
PyxosSts PyxosPilotReInitPyxosInterface(void);
```

Remarks: This function re-initializes the Pyxos FT Chip configuration, and reads it back to make sure that it is correct. If this function does not return **PyxosSts_Good**, it may be necessary to reset the Pyxos FT Chip, the processor or both. If so, this recovery action must be performed by the application.

Error Correction Example

This example illustrates how a Pilot application can detect and recover from communication problems with Pyxos Points. This example includes the following sections:

- *Checking Pilot Configuration*: This section illustrates how to detect and correct problems with the Pilot's configuration.
- *Point Status Declarations*: This section defines data structures and literals used to maintain status information about each Point.
- *Checking Point Configuration*. This section illustrates how to detect communication errors and recover from them using the Point status information.
- *Reconfiguring a Point*. This section illustrates how to reliably reconfigure a Point.

This example includes calls to a few application-specific functions whose implementations are not shown. For example, the **UpdateUserInterface()** function is meant to update the user interface when a Point is connected or disconnected. The use of these functions will be called out.

The Pyxos FT API defines a number of other callbacks that are not included in the following section. These callbacks are described previously in the chapter. Some of these callbacks play a role in error detection and correction as well.

Checking Pilot Configuration

The following function is called on each iteration of the main control loop to check and (if necessary) correct the Pilot's configuration. This function relies on an application defined function (not shown) called **TimeToCheckPilotConfiguration()** that returns True whenever it is time to check the Pilot's configuration. This could be implemented using a timer or simply based on the number of times it has been called.

```
void CheckPilotConfiguration(void)
{
   /* Periodically check the Pilot configuration.  This should
      not depend on the frame count, since no write frames will
      occur if there is a problem with the Pilot configuration.
```

```
      If the host has a timer, it could use that to determine the
      period.  Otherwise it could be based on how many iterations
      of the main control loop have passed.
    */
   if (TimeToCheckPilotConfiguration()) {
      /* Check to see if the Pilot configuration is good */
      if (PyxosPilotCheckConfiguration() != PyxosSts_Good) {
         /* Its not good, try to fix it */
         if (PyxosPilotReInitPyxosInterface() != PyxosSts_Good) {
            /* It may be necessary to reset the Pilot; this step
               may depend on the particular driver
             */

            /* To do:  post an error, and then do whatever is
               necessary to regain communication with the Pyxos
               FT Chip.  This may require resetting the driver or
               the host itself.
             */
         }
      }
   }
}
```

Point Status Declarations

The following data structures are used in this example to keep track of the status of each Point.

```
/* The maximum number of frames without receiving an update
   before deciding that the Point is not properly configured
 */
#define MAX_FRAMES_WITHOUT_AN_UPDATE 100

/* The frequency to check on the health of the Point, in frames;
   this must be much less than MAX_FRAMES_WITHOUT_AN_UPDATE
 */
#define NUM_FRAMES_BETWEEN_CHECKUP \
                     (MAX_FRAMES_WITHOUT_AN_UPDATE/2)

typedef struct
{
  /* Set to TRUE when the Point is connected and correctly
     configured, FALSE when it is not.
   */
  Bool pointConnected;

  /* Frame count the last time the Point was checked.  When this
     is NUM_FRAMES_BETWEEN_CHECKUP less than the current write
     frame, check up on the Point.
   */
  Dword fcPointChecked;

  /* Frame count the last time a value was received.  If this is
     ever MAX_FRAMES_WITHOUT_AN_UPDATE less than the current
     write frame, assume the Point has a problem.
   */
```

```
  Dword fcValueReceived;
} PointStatus;


/* Array of status information for each Point*/
PointStatus pointStatus[PYXOS_MAX_TIMESLOTS];
```

Checking Point Configuration

The following function is used to maintain Point status information using the **pointStatus** array defined in the previous section. The application calls this function with the *connected* parameter set to False whenever it determines that the Point is disconnected, and calls it with the *connected* parameter set to True whenever the Pilot receives a value from the Point (whether it was previously connected or not) to ensure that the frame count values are properly updated.

This function relies on an application defined function (not shown) called **UpdateUserInterface()**, which is responsible for updating some kind of user interface (LED, liquid crystal display, etc) indicating the connection status of each Point.

```
/* Update the Point status information. Update the user
  interface to denote that the Point is connected or
  disconnected, and maintain the pointStatus array.  This should
  be called each time we confirm that the Point is connected and
  each time we determine that it is disconnected.
 */
void UpdatePointStatus(PyxosTimeslot timeslot, Bool connected)
{
   /* Update the user interface as appropriate */
   UpdateUserInterface(timeslot, connected);
   pointStatus[timeslot].pointConnected = connected;

   if (connected) {
      /* Update the frame count of the most recent value
        received.
       */
      pointStatus[timeslot].fcValueReceived =
        pyxosPilotWriteFrameCount;
   }
}
```

The following function is called on each iteration of the main control loop in order to check on the connectivity of each of the Points.

```
/* This function is called on each iteration of the main control
   loop to check the status of each Point.  It periodically polls
   the configuration register.  If, after polling the
   configuration register, the Point has not responded for a
   number of frames, the Point is reset and will be reconfigured
   in the PyxosPilotResetPointCompleted() callback.
 */
void CheckPoints(void)
{
   PyxosTimeslot timeslot;
```

```
    for (timeslot = 0; timeslot < PYXOS_MAX_TIMESLOTS; timeslot++)
    {
       if (pointStatus[timeslot].pointConnected) {
           /* The Point has been configured and has recently been
               communicating with the Pilot
            */
          if (pyxosPilotWriteFrameCount -
              pointStatus[timeslot].fcValueReceived >
              MAX_FRAMES_WITHOUT_AN_UPDATE) {
             /* No data has been received for a long time.  The
                Point is probably misconfigured.
              */

             /* Update config status, display user info, etc. */
             UpdatePointStatus(timeslot, FALSE);

             /* Reset and then reconfigure the Point */
             PyxosPilotResetPoint(timeslot);
          } else if (pyxosPilotWriteFrameCount -
                    pointStatus[timeslot].fcPointChecked >
                    NUM_FRAMES_BETWEEN_CHECKUP) {
             /* It has been a while since we checked up on the
                Point.  Do so now.
              */
             pointStatus[timeslot].fcPointChecked =
                    pyxosPilotWriteFrameCount;

             PyxosPilotPollRegister(timeslot, PYXOS_REGI_CONFIG);
          }
       }
    }
}
```

The **ValidatePointConfiguration()** function is used to check the configuration of a Point whenever the Pilot receives the Point's configuration register. It is called in the **PyxosPilotPnvUpdateOccurred()** callback. This function uses a utility function called **HostedPoint()** (not shown) that takes a Point interface Pointer and returns True if the Point is hosted, and False otherwise.

```
/* This function is used to validate a Point's configuration
   register.  If the register is invalid, the Point is reset.
 */
void ValidatePointConfiguration(PyxosTimeslot timeslot,
                                const Byte *pConfigReg)
{
   Bool badConfig;
   PyxosConfigRegister config;

   /* We know that the device is configured, and in the correct
      timeslot, otherwise we wouldn't have gotten this update.
      However, we should check the host type to make sure it is
      correct.  The host type could get corrupted due to an ESD
      hit.
    */
   memcpy(&config, pConfigReg, sizeof(config));
```

```
   if (HostedPoint(PyxosPilotGetPointInterface(timeslot))) {
      badConfig = PYXOS_CFG_GET_HOST_TYPE(config) !=
                  PYXOS_CFG_HOST_POINT;
   } else {
      badConfig = PYXOS_CFG_GET_HOST_TYPE(config) !=
                  PYXOS_CFG_UNHOSTED_POINT;
   }

   if (badConfig) {
      /* Report error */
      UpdatePointStatus(timeslot, FALSE);
      PyxosPilotResetPoint(timeslot);
   }
 }
```

The **PyxosPilotPnvUpdateOccurred()** callback is called by the Pyxos API whenever a PNV update is received by the Pilot. In addition to any application processing of the PNV, this function should mark the Point as being connected. In addition, if the PNV is the Point's configuration register, the function should verify that the configuration is correct.

```
void PyxosPilotPnvUpdateOccurred(PyxosTimeslot timeslot,
                                 PyxosPci pci,
                                 const Byte *pPnvValue,
                                 Byte length)
{
   UpdatePointStatus(timeslot, TRUE);
   if (pci == PYXOS_REGI_CONFIG) {
      ValidatePointConfiguration(timeslot, pPnvValue);
   } else {
      /* Process PNV updates as necessary */
   }
}
```

The **PyxosPilotUpdatePnvCompleted()** callback is called by the Pyxos API when a Pyxos FT network is successfully updated, or when a PNV update times out. If the *success* parameter is False, the status of the Point should be set to **Disconnected** and Point should be reset. If the **status** parameter is set to *success*, it does not necessarily mean that the Point is in good shape. It only indicates that the Pilot can send updates to the Point and receive acknowledgement from the Point. This could be the case even if the Point's send TID does not match the Pilot's receive TID. For that reason the Point status is updated only on failure.

```
void PyxosPilotUpdatePnvCompleted(PyxosTimeslot timeslot,
                                  PyxosPci pci,
                                  Bool success)
{
   if (!success) {
      /* Can no longer talk to the Point.  Try resetting to re-
         establish communication.
       */
      UpdatePointStatus(timeslot, FALSE);
      PyxosPilotResetPoint(timeslot);
```

```
   }
}
```

The **PyxosPilotPollRegisterCompleted()** callback is called by the API when a poll request has been acknowledged or has timed out. If the *success* parameter is False, the status of the Point should be set to **Disconnected**, and Point should be reset. The status of success does not necessarily mean that the Point is in good shape. It only indicates that the Pilot can send updates to the Point and receive acknowledgement from the Point. This could be the case even if the Point's send TID does not match the Pilot's receive TID. For that reason the Point status is updated only on failure.

```
void PyxosPilotPollRegisterCompleted(PyxosTimeslot
                                     timeslot,PyxosPci pci,
                                     Bool success)
{
   if (!success) {
      /* Can no longer talk to the Point.  Try resetting to re-
         establish communication.
       */
      UpdatePointStatus(timeslot, FALSE);
      PyxosPilotResetPoint(timeslot);
   }
}
```

Reconfiguring a Point

The following code illustrates how to properly reconfigure a Pyxos Point after it has been reset, after a configuration error has been detected, or after a previous configuration attempt has failed. The callbacks use **UpdatePointStatus()** to manage the **pointStatus** array.

The **PyxosPilotResetPointCompleted()** callback is called by the API when a reset has completed or timed out. It should attempt to reconfigure the Point, and make sure that the status of the Point is set to "unconnected".

```
/* The configuration timeout depends on how long it takes the
   Point to reset.  This is very specific to the Point
   application and host processor.  This parameter is expressed
   in terms of frames, and so it depends on the value of
   PYXOS_NUM_TIMESLOTS.  In this example, allow 1 second for the
   Point to reset and be ready to be reconfigured, and assume 32
   timeslots, resulting in a frame every 25 Msec (25Mse*40 = 1
   second.
 */
 #define CONFIGURATION_TIMEOUT 40

 /* PyxosPilotResetPoint was called, and has either succeeded or
    timed out.
  */
 void PyxosPilotResetPointCompleted(PyxosTimeslot timeslot,
                                    Bool success)
 {
   /* If the reset succeeded then the Point is no longer
      configured.  If it failed it is probably because it is no
```

```
      longer configured.  Mark it as unconnected, and then try to
      reconfigure it.
    */
  UpdatePointStatus(timeslot, FALSE);

  PyxosPilotReconfigurePoint(timeslot, CONFIGURATION_TIMEOUT);
}
```

The **PyxosPilotPointConfigured()** callback is called by the Pyxos API if the configuration completes successfully, and the **PyxosPilotPointConfigurationFailed()** callback is called if the configuration process times out. These callbacks are also used after the initial timeslot allocation. See the *Example Code for Automatic and Manual Registration* section on page 62 for examples of these callbacks.

Reconfiguring the Network After a Reset

When the Pilot application resets, all information about the Pyxos Points stored by the Pyxos Pilot API is lost. Depending on whether the Pyxos Points were reset or not, they may or may not be configured when the Pilot reset is complete. In either case, the Pilot must re-allocate timeslots and set Point interfaces in order to restore the network to the same status it was in before the reset. This section describes what the Pilot application must do to reconfigure the network after it resets.

A Pilot application should almost always provide non-volatile storage to keep track of the Points that it has discovered. At a minimum, this storage should include the unique ID and timeslot of each Pyxos Point that is currently defined in the system. If the Pilot cannot automatically determine the interface to use from the Point, the Pilot should also record which interface each Point uses. There are two main reasons why the Pilot should store this information:

1. If any user intervention was required to define the network, this information should be captured in the non-volatile data. Otherwise user intervention will be required each time the Pilot resets.

2. Even if all of the Points on the network use automatic registration and the Pilot can register them without any user intervention, it is still a good idea to maintain the unique ID of each Point. When the Pilot starts, it will reset any configured points in the network and any Points using the automatic registration method will re-register themselves with the Pilot. However, the Pilot does not need to wait for the Points to send registration requests if it allocates the timeslots on startup after reading the information from non-volatile data.

It may not be necessary for the Pilot application to maintain non-volatile storage if the network is composed entirely of hardwired Points, or if the network has no manual Points and there is no possibility that the Pilot will reset without all the automatic Points resetting as well.

Reset Example

In this example the Pilot stores only the timeslot and unique ID of each Point. If any user intervention was required to distinguish the function of one Point

versus another, that information should be stored in the non-volatile data as well.

The data for each Point is stored in an array, indexed by timeslot, using the following definition:

```
/* NonVolatilePointInfoEntry is used to store persistent
   information about a specific Point.
 */
typedef struct
{
/* The Point's unique ID.  0 if none. */
PyxosUniqueId uniqueId;
} NonVolatilePointInfoEntry;
```

The **RestorePointsFromNvData()** function is called by the main function, after calling **PyxosPilotInit()** (but before calling the event handler). This function reads the non-volatile data using the **ReadNonVolatileData()** utility function (not shown), and then re-allocates the timeslots based on that information. After the timeslots have been allocated, the **PyxosPilotPointConfigured()** callback will continue the process. The Pilot should not set the interface until **PyxosPilotPointConfigured()** has been called, as doing so will cancel reconfiguring the Point.

```
static const PyxosUniqueId nullUniqueId = {0,0,0,0,0,0,0,0};
/*
Restore Points from non-volatile data.
*/
void RestorePointsFromNvData(void)
{
   PyxosTimeslot timeslot;
   NonVolatilePointInfoEntry nvData[PYXOS_MAX_TIMESLOTS];

   /* This function reads the unique IDs of each Point out of
      non-volatile data into the nvData array, indexed by
      timeslot.
    */
    ReadNonVolatileData(nvData);

    /* Scan the nvData array and re-allocate the timeslots of
      each Point that appears in the array.
     */
    for (timeslot = 0;
        timeslot < PYXOS_NUM_TIMESLOTS;
        timeslot++) {
      if (memcmp(nvData[timeslot].uniqueId, nullUniqueId,
                 sizeof(PyxosUniqueId)) != 0) {
         /* Re-allocate the timeslot to the Point. */
         PyxosPilotAllocateTimeslot(&nvData[timeslot].uniqueId,
                                    timeslot,
                                    CONFIGURATION_TIMEOUT,
                                    FALSE);
      }
   }
}
```

The **UpdateNvData()** function is called each time a Point is allocated or deleted. If more information, such as the program interface, is stored in the non-volatile data, this function may need to be called in other places as well. The **WriteNonVolatileData()** function (not shown) is used to write the array of NonVolatilePointInfoEntry out to non-volatile memory.

```
/* This function updates the non-volatile data with the current
   timeslot assignments.
 */
void UpdateNvData(void)
{
   PyxosTimeslot timeslot;

   /* An array of unique IDs, indexed by timeslot. */
   NonVolatilePointInfoEntry nvData[PYXOS_MAX_TIMESLOTS];

   memset(nvData, 0, sizeof(nvData));

   /* Find the unique IDs of every Point and fill in the nvData
      array.
    */
   for (timeslot = 0; timeslot < PYXOS_NUM_TIMESLOTS; timeslot++)
   {
      const PyxosUniqueId *pUniqueId =
         PyxosPilotGetUniqueId(timeslot);

      if (pUniqueId != NULL) {
         memcpy(nvData[timeslot].uniqueId, pUniqueId,
         sizeof(PyxosUniqueId));
      }
   }

   /* Write the nvData array out to non-volatile data. */
   WriteNonVolatileData(nvData);
}
```

Analyzing Pyxos FT Network Communication

When debugging a Pyxos application, it may be useful to implement a protocol analyzer that logs the packets sent on the Pyxos FT network. You can use the **PyxosPilotProtocolAnalyzerCallback()** to aid in this development. This information may also be extremely valuable when diagnosing problems in the field.

If you select the **Select the Enable Support for Protocol Analyzer Callbacks** check box when you configure the Pilot with the Pilot Application Options dialog, this function will be called every time the Pilot API reads or writes a packet. You can log these packets in whatever way works with the resources available to your host processor.

PyxosPilotProtocolAnalyzerCallback()

Provides notification that a the Pilot API has read or written a packet. If you select the **Select the Enable Support for Protocol Analyzer Callbacks** check box

when you configure the Pilot with the Pilot Application Options dialog, this function will be called every time the Pilot API reads or writes a packet.

Syntax:

```
PYXOS_API_DECL void
   PyxosPilotProtocolAnalyzerCallback(Bool incoming,
                                      Dword writeFrameCount,
                                      PyxosTimeslot timeslot,
                                      const PyxosPacket
                                      *pPacket);
```

Remarks: You can use this function to create a protocol analyzer to monitor the Pyxos FT network. It is called every time the Pilot reads or writes a packet. This callback does not necessarily get called for every frame. For example, the Pilot application may write data to a timeslot and the Pyxos FT Chip may continue writing that data for several frames. Similarly, once the data has been acknowledged, the chip will send idles once the Pilot has no more data to send. If a Pyxos Point has no data to send to the Pilot, the Point's chip will send idles, but the Pilot API will not read them, and will not call this function.

The **incoming** parameter is True when the Pilot has read the timeslot, and False if the Pilot has written the timeslot. The *writeFrameCount* parameter is equal to the **pyxosPilotWriteFrameCount** global value, which is incremented each time the Pilot processes a write frame. The timeslot is a value between 0 and **PYXOS_NUM_TIMESLOTS** - 1, representing the timeslot of data. The final parameter, *pPacket*, points to an internal buffer containing the 10 bytes of the Pyxos packet data. The application must not modify this data at any time, or use this pointer once the callback returns. No Pyxos FT API functions may be called during this callback. The format is shown below.

NOTE: When operating in on-demand TDM mode (meaning that you selected the **Use On-Demand TDM Mode** check box when configuring the Pilot with the Pilot Application Options dialog), the Pilot forces frames onto the network when the Pilot Application calls **PyxosPilotEventHandler()** and the network is currently idle. It does this by sending the next outstanding updates, if there are any. If there are none, it forces a frame to be sent by writing an idle pattern (both PCIs set to 0xff) to timeslot 0. As a result, **PyxosPilotProtocolAnalyzerCallback()** may be called for timeslot 0 more often than for any other timeslot. See Chapter 7 for more information on on-demand TDM mode.

```
/* Each timeslot contains two entries consisting of a PyxosPci
and 4 bytes of data. These entries are represented by a
PyxosDataItem (PDI)
 */
   #define PYXOS_NUM_PDIS 2
   typedef struct _PyxosDataItem
   {
       PyxosPci index;
       Byte     data[PYXOS_PCV_SIZE];
   } PyxosDataItem;

   /* The PyxosPacket structure represents a single packet. */
   typedef struct _PyxosPacket
   {
       PyxosDataItem pdi[PYXOS_NUM_PDIS];
```

```
} PyxosPacket;
```

System Diagnostics

You can design system diagnostic capability into your Pyxos Pilot application to simplify field debugging by a maintenance technician. The information that you will need to gather, and how that information should be reported to a technician, depends largely on your application needs and platform capabilities. Some typical diagnostics that you should consider reporting are:

1. **Timeslot allocation report:** Which Points are allocated to each timeslot. Include which interface each Point uses, and any pertinent user configuration.
2. **Status information:**
    a. For each Point, indicate whether or not the Pilot is currently able to communicate with it, and if not, what action the Pilot is taking to recover the Point.
    b. Provide the ability to collect the missed slot and CRC error statistics gathered by calling **PyxosReadNetworkStats()**.
    c. The number of errors reported by callback functions.
3. **Tracing:**
    a. Provide the output of from the **PyxosPilotProtocolAnalyzerCallback()** callback.
    b. Provide tracing from other callback functions, especially when a failure has occurred.

5

Including the Pyxos FT API

This chapter describes how to include the Pyxos Pilot or Point API in your application.

Introduction

This chapter describes how to include the Pyxos FT API in your application. Both the Pilot API and the Point API are delivered as portable C source code, rather than as a library. There are two reasons for this:

1. You can port the Pyxos FT API to other platforms.
2. You can customize the Pyxos FT API with compile-time options that you choose with the Pyxos FT Interface Developer.

To include the Pyxos FT API in your application, follow these steps:

1. Customize the [*Pyxos FT EVK*]**\Pyxos FT API\Platform.h** file to include a section for each of the host processors you intend to support. See Chapter 6 for more information on this.
2. Implement a Pyxos serial driver for your host platform. See Chapter 6 for more information on this.
3. Use the Pyxos FT Interface Developer to create the required include files for your Pyxos Point or Pilot application, as described in Chapter 2. This tool generates includes files that are required by the Pyxos FT API.
4. Include the Pyxos FT API files as described in the remainder of this chapter.

Common Components

The Pyxos FT Point and Pilot APIs consist of the following components. In the following list, [*Pyxos FT EVK*] is the directory in which you installed the Pyxos FT EVK software, usually **C:\Program Files\Echelon\Pyxos FT EVK**:

1. The Pyxos Point interface include file, which is generated by the Pyxos FT Interface Developer for your Pyxos Point application. One of these files is created for each Pyxos Point. This interface file is used by the Point application, the Pyxos FT API, and the Pilot application.
2. The **MyPyxosApplication.h** file generated by the Pyxos FT Interface Developer for your Pyxos Point or Pyxos Pilot application. This file is not generally shared with other applications, as it customizes the implementation of the Pyxos FT API for a particular type of Pyxos Point or Pilot. This file includes the Pyxos Point interface include file (or all of them on a Pilot). While the content of this file is customized for each type of Point and Pilot, the name is fixed, because the file is included by the Pyxos FT API. The Pyxos FT API uses the definitions in this file, as well as the definitions of the Pyxos Point Interface for compile time customization of the Point API.
3. The C source files in the [*Pyxos FT EVK*]**\Pyxos FT API** folder.
4. The external API header files that the API includes, and your application may need to reference, in the [*Pyxos FT EVK*]**\Pyxos FT API\include** folder. Your application must include the **Pyxos.h** file, which automatically includes all of the other required include files, including

your **MyPyxosApplication.h** file. The Pyxos FT API external include files are:

- **Platform.h**: Platform-specific type definitions. You must customize this file to support your host processor. See Chapter 6 for more information on this.
- **Pyxos.h**: The main Pyxos FT API include file for both the Pyxos Point and Pilot APIs. This file includes your **MyPyxosApplication.h** file, and indirectly includes your Point interface file or files.
- **PyxosRegister.h**: Pyxos FT Chip register definitions.
- **PyxosShared.h**: Miscellaneous definitions that are shared by both the Pyxos Pilot and Point APIs.
- **PyxosPoint.h**: External definitions and functions used by the Point API. Used by Pyxos Point applications only. The **Pyxos.h** file does not include this for Pilot applications.
- **PyxosPilot.h**: External definitions and functions used by the Pilot API. Used by Pyxos Pilot applications only. The **Pyxos.h** file does not include this for Point applications.

5. The internal API header files contained in the [*Pyxos FT EVK*]**/Pyxos FT API/include/internal** folder. The include files in this folder are used by the Pyxos FT API, but are not meant for use by the Pyxos application.

6. Shared Pyxos Serial Driver source code, in the [*Pyxos FT EVK*]**\Pyxos FT API\Serial API** folder. This consists of a single file, **psUtilImpl.c**.

7. Shared Pyxos Serial Driver include files, in the [*Pyxos FT EVK*]**\Pyxos FT API\Serial API\include** folder. These include files are used by the Pyxos Point and Pilot APIs.

8. The Customized Pyxos Serial Driver Code. You must implement the Pyxos serial driver, as described in Chapter 6. This implementation may be shared by many Points and Pilots, but is affected both by the host processor and the pins used to connect to the Pyxos FT Chip.

9. You will also have to define your host's type in your project settings to pick up the correct definitions from the **platform.h** file. See Chapter 6 for more information on this.

Including the Point API in your Application

The project you use to create the Pyxos Point application must include the following source files, where [*Pyxos FT EVK*] is the directory in which you installed the Pyxos FT EVK software, usually **C:\Program Files\Echelon\Pyxos FT EVK**:

- [*Pyxos FT EVK*]**\Pyxos FT API\PyxosPoint.c**
- [*Pyxos FT EVK*]**\Pyxos FT API\PyxosBits.c**
- [*Pyxos FT EVK*]**\Pyxos FT API\PyxosUtil.c**
- [*Pyxos FT EVK*]**\Serial API\psUtilImpl.c**
- Your implementation of the Pyxos Serial API

- Your Point application source files.

The project must include the following folders in the include path:

- [*Pyxos FT EVK*]**\Pyxos FT API\include**
- [*Pyxos FT EVK*]**\Pyxos FT API\Serial API\include**
- Path of the folders containing your **MyPyxosApplication.h** file and your Point interface include file.
- Path for any include files that your custom serial driver might need.
- Path for your application's include files.

Any source files that access either the Point API or the Point's interface definitions must include the **Pyxos.h** file, which automatically includes the **MyPyxosApplication.h** file, the Point's interface include file and all of the external API functions supported by the Point API.

Including the Pilot API in your Application

The project you use to create the Pyxos Pilot application must include the following source files, where [*Pyxos FT EVK*] is the directory in which you installed the Pyxos FT EVK software, usually **C:\Program Files\Echelon\Pyxos FT EVK**:

- [*Pyxos FT EVK*]**\Pyxos FT API\PyxosPilot.c**
- [*Pyxos FT EVK*]**\Pyxos FT API\PyxosPilotProcessInputs.c**
- [*Pyxos FT EVK*]**\Pyxos FT API\pyxosPilotProcessOutputs.c**
- [*Pyxos FT EVK*]**\Pyxos FT API\PyxosBits.c**
- [*Pyxos FT EVK*]**\Pyxos FT API\PyxosUtil.c.**
- [*Pyxos FT EVK*]**\Pyxos FT API\Serial API\psUtilImpl.c**
- Your implementation of the Pyxos Serial API
- Your Point application source files.

The project must include the following folders in the include path

- [*Pyxos FT EVK*]**\Pyxos FT API\Include**
- [*Pyxos FT EVK*]**\Pyxos FT API\Serial API\Include**
- Path of the folders containing your **MyPyxosApplication.h** file and all of your Point interface include files.
- Path for any include files that your custom serial driver might need.
- Path for your application's include files.

Any source files that access either the Pilot API or any of the Point's interface definitions must include the **Pyxos.h** file, which automatically includes the **MyPyxosApplication.h** file, all of the Point's interface include files and all of the external API functions supported by the Pilot API.

ANSI C

The Pyxos FT API is compatible with strict ANSI C compilers. However, you may need to modify some of the type definitions used to implement bitfields in the **platform.h** file if you intend to use a strict ANSI C compiler. The **BitField**

and **SignedBitField** types are typically defined as an 8-bit quantity, while ANSI C requires that bitfields be defined as **int**. The **BitFieldWord** type, used to implement bitfields that are between 9 and 16 bits (or span the 9 and 16 bit boundary) is typically defined as a 16-bit value. In order to use a strict ANSI C compiler you will need to define all three of these types using **int**. However, doing so will have two side effects:

1. Increased memory consumption
2. If you are using the ShortStack API and have included any LONWORKS definitions using these fields, you will probably need to redefine these structures to eliminate the bit fields and replace them with the appropriately sized data items to ensure that the structures maintain the proper alignment (matching those specified by the ANSI/CEA-709.1 protocol). You will then need to convert any references to the bit fields to use bitmasks.

For more information about the customizing the **platform.h** file, see Chapter 6.

6

Porting the Pyxos FT API

This chapter describes how to port the Pyxos FT API to a new processor or compiler. This includes hardware considerations and details on the Pyxos Serial API (psAPI), and on modifications you can make to the **platform.h** file to suit your microprocessor.

SPI Overview

When you use the Pyxos FT Chip in hosted mode, your host processor communicates with the Pyxos FT Chip using a serial peripheral interface (SPI) bus. The SPI bus is a synchronous serial data link standard. Devices communicate in master/slave mode where the master device (your host) initiates each data frame. Multiple slave devices are allowed with individual slave select (chip select) lines. A typical SPI slave port consists of a clock input, a select line, data input, and data output. The Pyxos FT Chip provides these interfaces, as well as an interrupt line to improve microcontroller response times, and support for two- and three-wire modes of operation.

Many microcontrollers have an embedded SPI port that can be configured to work with the Pyxos FT Chip. If the microcontroller does not include an SPI port, the SPI can be bit-banged from software.

The SPI protocol simply defines a physical interface—there are no universal SPI standards for data framing and command structure. However, many serial memories have adopted a standard for messaging over an SPI port, and many software drivers have been developed that use this standard. The Pyxos FT Chip adopts this standard with only minor variations—existing serial memory drivers can be easily adapted to work with the Pyxos FT Chip.

SPI Slave Mode Port Connections

The Pyxos FT Chip SPI interface consists of the standard 4-pin SPI slave mode interface (**CS~/DIO2**, **SCLK/DI**, **MOSI/DIO0**, and **MISO/DIO1**) plus an interrupt signal (**INT~/DIO3**). These pins are summarized in Table 15 below. A host microcontroller can interface to the Pyxos FT Chip with as few as two pins (using **SCLK** and tying **MOSI** and **MISO** together). Higher performance microcontrollers can use all five pins to take advantage of the interrupt and other features.

Table 15 Pyxos FT Chip SPI Slave Mode Interface Pins

Name	Pin	SPI Function
CS~/DIO2	1	Active-low chip select for selecting the Pyxos FT Chip on the SPI bus and framing transfers
SCLK/DI	3	SPI bus transfer clock (max 1.25MHz)
MOSI/DIO0	4	Master-out, slave-in serial data
MISO/DIO1	5	Master-in, slave-out serial data
INT~/DIO3	2	Active-low interrupt output to signal Pyxos FT network activity

SPI Modes, Transfer Framing, and Half-Duplex Operation

There are four standard modes of SPI clock operation, depending upon whether the output data (both **MOSI** and **MISO**) are changed on the asserted or released edge of the clock, and whether **SCLK** idles high or low. The assert clock edge is the edge going from idle to active; the release edge of the clock is the edge going from active to idle. The Pyxos FT Chip supports two of these modes: mode 0 and mode 3, as described in Table 16.

In addition, data transfers are framed either by the chip select (**CS~**), or by timeouts. If **CS~** framing is used, **CS~** is asserted (low) before the data transfer begins, and is released (high) after the last byte of the transfer completes. If timeout framing is used, the time between bits and bytes within a transfer must be less than the minimum timeout value and **CS~** must remain high for the entire duration of the transfer. Timeouts are defined by holding **SCLK** high for longer than the maximum timeout value (this method of defining data transfers cannot be used in SPI mode 0). If **SCLK** is expected to idle for significant periods (greater than the timeout value) within a transfer, then **CS~** framing must be used along with SPI Mode 0.

Table 16 Supported SPI Transfer Modes and Transfer Framing Options

SPI Mode	SCLK Idle	Change Output on SCLK Edge	Sample Input on SCLK Edge	Transfer Framing
0[1]	Low	Release (high-to-low)	Assert (low-to-high)	CS~
1	Low	Assert (low-to-high)	Release (high-to-low)	NA
2	High	Release (low-to-high)	Assert (high-to-low)	NA
3[2]	High	Assert (high-to-low)	Release (low-to-high)	CS~
				timeout

Notes:
1. **SCLK** is allowed to idle for greater than timeout within a transfer in this mode.
2. **SCLK** period must be less than timeout for all bits with a transfer in this mode.

All transfers to and from the Pyxos FT Chip are half-duplex: data is sent only in one direction at a time. The **MISO** pin is driven only during the portion of a transfer where the Pyxos FT Chip is actually sending data. For example, during a read operation, the first two bytes set up the read, so the Pyxos FT Chip will not drive **MISO**; after that, the Pyxos FT Chip will drive **MISO** until the end of the transfer. If the host microcontroller also obeys this convention (i.e., only drives the **MOSI** pin when sending data), then the **MISO** and **MOSI** pins can be electrically tied together to one pin on the microcontroller.

Microcontroller Connections

The Pyxos FT Chip SPI slave mode interface provides several options for connecting to a host microcontroller, depending upon performance, flexibility,

and available hardware resources. There are three connection schemes: four-wire connection, three-wire connection, and two-wire connection.

The four-wire connection scheme shown in Figure 16 provides the most flexible solution. It allows other SPI peripherals to be attached to the SPI bus. However, you must determine whether there is sufficient bandwidth available to service all of the devices on the SPI bus. It is the fastest connection scheme since timeout framed transfers require that the timeout period (100µs) expire between transfers. However, **CS~** framed transfers only require that **CS~** be raised for 400ns between transfers. It is straightforward to use this solution; if the microcontroller contains a hardware SPI port or there is an available software SPI master, then this can be used as is. This design is most appropriate for Pilots or a Point that has multiple SPI devices.

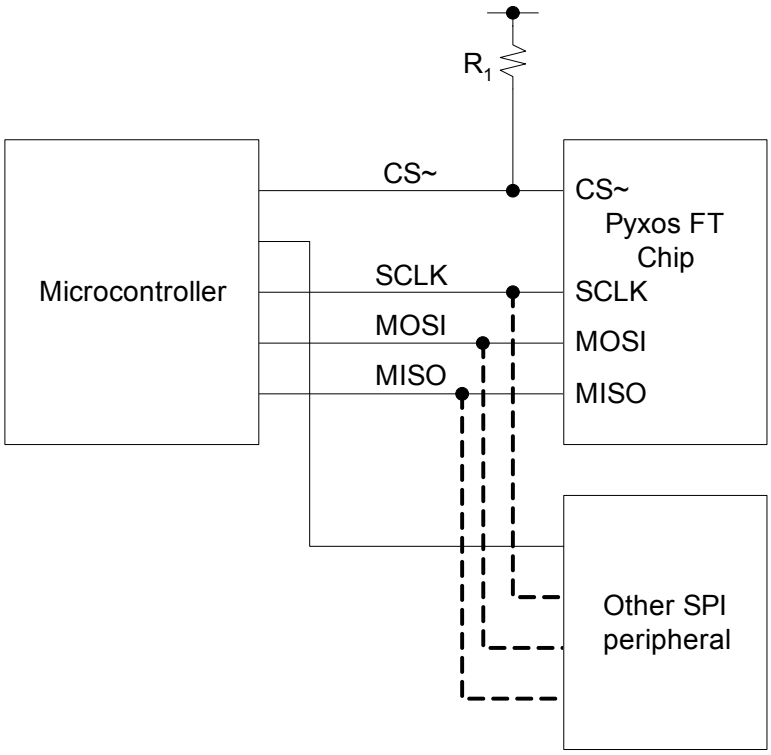

Figure 16 Four-wire Connection Scheme

The three-wire connection scheme shown in Figure 17 is used for Pyxos Points when the Pyxos FT Chip is the only SPI device that the microcontroller communicates with, at least on that SPI port. This design is not appropriate for Pilots since the inter-transfer gap must be at least 100µs. However, if you are developing a Pyxos Point and the microcontroller includes a hardware SPI port or has an available software SPI port, then this is a good choice.

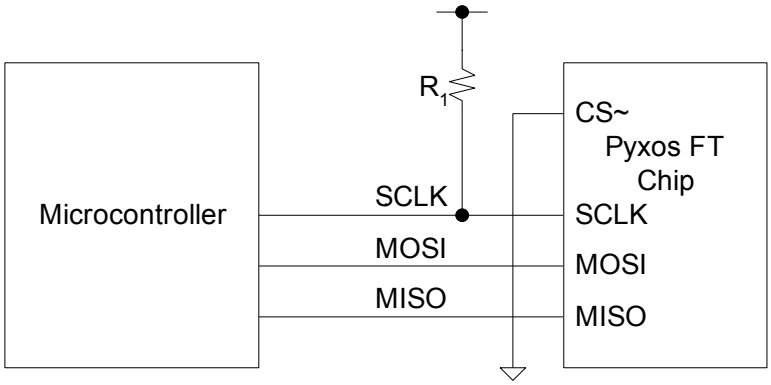

Figure 17 Three-wire Connection Scheme

The two-connection scheme shown in Figure 18 is used when the host microcontroller has extremely limited IO capability. This connection likely requires that the SPI port be bit-banged on the microcontroller. Hardware based SPI ports do not typically support sharing the **MOSI** and **MISO** signals. This is most appropriate for low-cost Points.

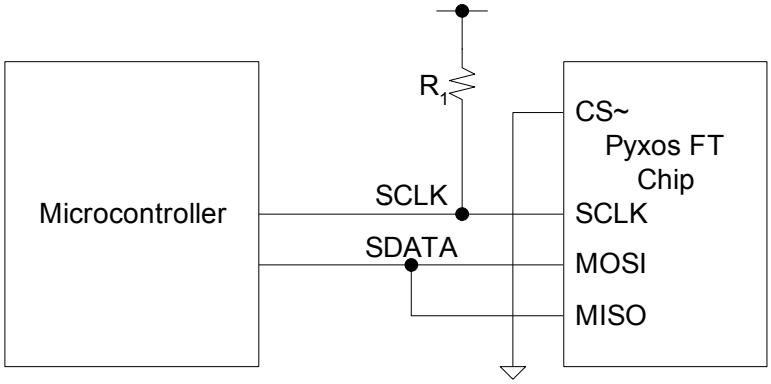

Figure 18 Two-wire Connection Scheme

Table 17 SPI Connection Options

Connection Scheme	Host Pins	Transfer Framing	Inter-transfer Time	Supported by Standard SPI	Multi SPI Device Support	Typical Usage
Four-wire	4	CS~	400ns	Yes	Yes	Pilot or Point with other SPI devices
Three-wire	3	Timeout	100µs	Yes	No	Point
Two-wire	2	Timeout	100µs	No	No	Low-cost Point

Your application's performance requirements and your selected host microcontroller will determine which communications schemes are available and sufficient. Some applications may be able to tolerate the performance overhead of a bit-banged SPI port, and others may need to run the SPI bus at the maximum allowable clock rate.

On the Pilot, the Pyxos FT Chip needs to transfer all of the timeslot data for every frame. However, on a Point, the Pyxos FT Chip can send and receive at most two four-byte values per frame. On the Pilot, as the number of timeslots decreases, the amount of data that needs to be exchanged between the microcontroller and the Pyxos FT Chip decreases, but so does the frame time. As a result, the SPI bandwidth requirement on a Pilot is relatively constant. However, on a Point, fewer timeslots on the network require greater SPI bandwidth.

Table 18 indicates the maximum number of bytes that need to be transferred between the host microcontroller and the Pyxos FT Chip on both a Pilot and a Point. These bandwidth values are only required if the application expects to service all I/O with minimal latency. If the application can tolerate greater latencies, the bandwidth requirements can be scaled down. Table 18 does not include overhead; e.g. reading each byte from the Pyxos FT Chip with separate SPI transfers will increase the bandwidth requirements by a factor of three, while reading all timeslots in one transfer incurs very little overhead. There may also be overhead factors in the software that need to be accounted for. Finally, these are maximum sustained-average rates; peak rates may be substantially higher.

Table 18 Sustained SPI bandwidth requirements

Time-slots	Frame Time (ms)[1]	Pilot		Point	
		Max Data Bytes Per Frame[2,3]	Average SPI Bandwidth (kbps)[5]	Max Data Bytes Per Frame[2,4]	Average SPI Bandwidth (kbps)[5]
2	1.8	40	175	16	70
4	3.4	80	190	16	38
6	4.9	120	195	16	26
8	6.5	160	198	16	20
10	8.0	200	200	16	16
12	9.5	240	201	16	13
14	11.1	280	202	16	12
16	12.6	320	203	16	10
18	14.2	360	203	16	9.0
20	15.7	400	204	16	8.1
22	17.3	440	204	16	7.4
24	18.8	480	204	16	6.8
26	20.3	520	205	16	6.3
28	21.9	560	205	16	5.9
30	23.4	600	205	16	5.5
32	25.0	640	205	16	5.1

Notes:

1. Approximate value.
2. Does not include SPI transfer overhead bytes.
3. A timeslot requires 10 bytes (two indices + eight bytes of data), and there are both a read timeslot and write timeslot in each frame.
4. A Point receives and sends at most two indices in each direction per frame.
5. Sustained average rate; peak rate may be significantly greater.

Serial Driver Design

The serial driver design is dependent on the host used, and whether it will be used on a Pilot or Point. There are three different approaches that can be used for the implementation of the serial driver:

- **MANUAL**: The serial driver drives specific I/O pins of the host manually to communicate over the SPI bus with the Pyxos FT Chip.
- **USART**: The serial driver uses an existing universal synchronous asynchronous receiver/transmitter (USART) directly to communicate over the SPI bus with the Pyxos FT Chip.
- **DMA**: The serial driver uses a direct memory access (DMA) engine provided by the host to communicate over the SPI bus with the Pyxos FT Chip.

The Pyxos FT EVK uses the peripheral DMA controller (PDC) of the ARM7 chip to implement the serial driver.

SPI Protocol Operation Codes

The SPI interface uses a simple protocol modeled after SPI EEPROM memories for reading and writing to memory locations within the Pyxos FT Chip. Existing SPI EEPROM drivers can easily be adapted to working with the Pyxos FT Chip SPI interface.

The Pyxos FT Chip supports a total of four SPI operations, as described in Table 19. Figures 19 through 22, which follow Table 19, illustrate each of these operations with examples.

All data and address bytes are transferred most-significant bit first.

Table 19 Pyxos FT Chip SPI Op Codes

Op Code[1,2]	Mnemonic	Operation
000$A_9A_8$011	READ	Read data from memory
000$A_9A_8$010	WRITE	Write data to memory
000XX101	RDSR	Read status register
000XX001	WRSR	Write status register

Notes:

1. 'X' represents a don't care bit; these may be either 0 or 1.
2. A_9 and A_8 are the most significant address bits. The rest of the address ($A_7 – A_0$) follows in the next byte.

The read and write operations (Figures 19 and 20) support multi-byte transfers. The initial address is taken from the first two bytes (A_9 to A_0). The address is then incremented for successive bytes (incrementing address 0x3FF will wrap around to 0x000). Thus, a large block of data can be read or written in one

transfer with only two bytes of overhead. The transfer is terminated by either raising **CS~** (for **CS~** framing) or allowing **SCLK** to timeout (for timeout framing). A read operation can be terminated in the middle of a byte. Write operations must write all 8 bits of a byte to modify the byte on the Pyxos FT Chip.

Figure 19 shows a single byte read and Figure 20 shows a single byte write. Both assume SPI Mode 0 and **CS~** framing.

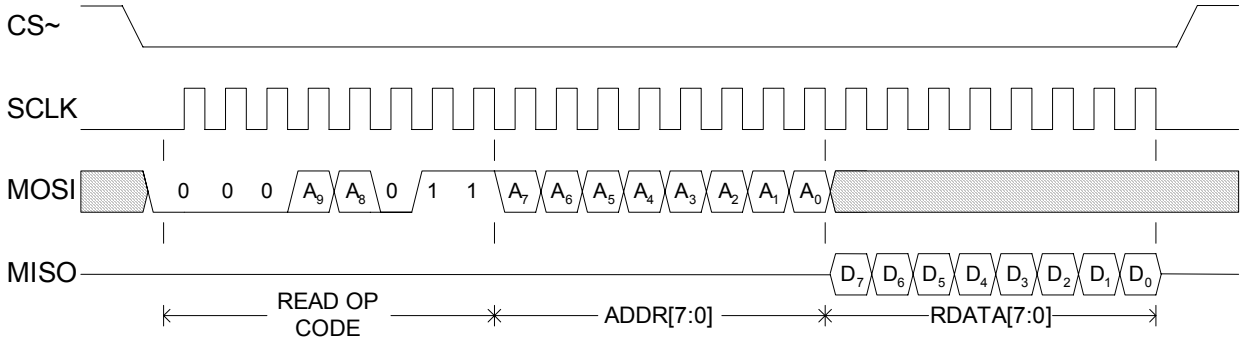

Figure 19 Example SPI Read Operations Using SPI Mode 0 and CS~ Framing

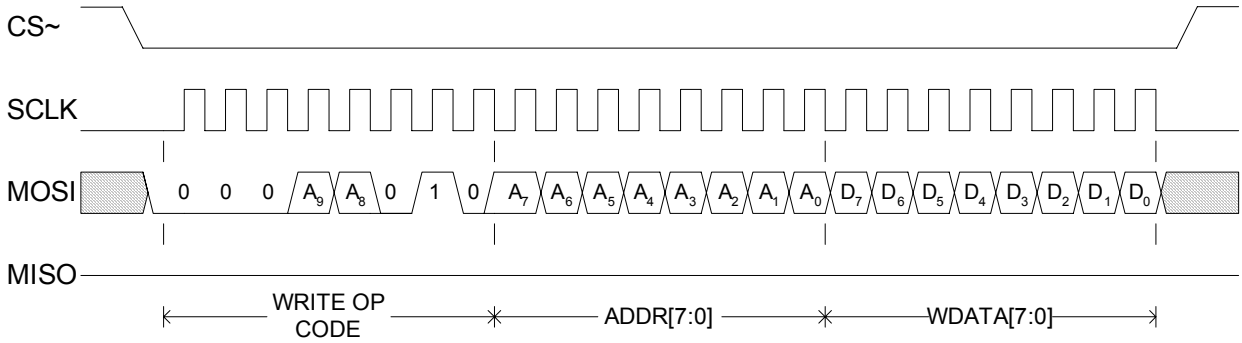

Figure 20 Example SPI Write Operation Using SPI Mode 0 and CS~ Framing

The RDSR and WRSR operations (Figures 21 and 22) provide quick access to the interrupt register (this register is also available through the standard READ and WRITE operations using the 10-bit address of the ISR). These operations reduce the transfer length to access the ISR by 8 bits. All four bytes of the ISR can be accessed, or the transfer can be terminated early. If a write is terminated early, only those bytes that are completely transferred will be written. Attempting to read or write beyond the four ISR bytes will have no affect.

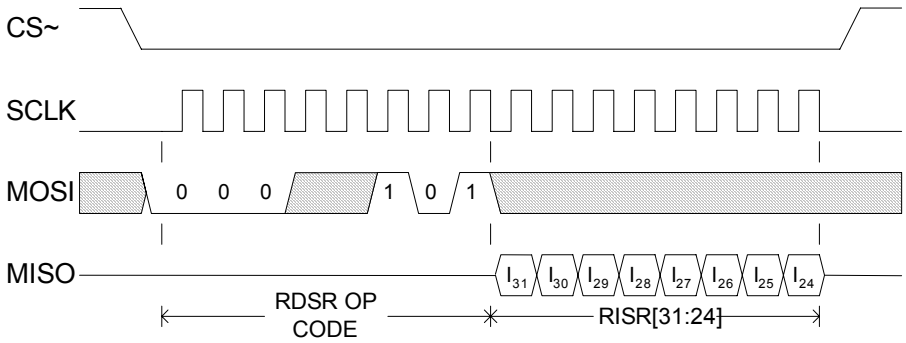

Figure 21 Example SPI Read Status Register Operation Using SPI Mode 0 and CS~ Framing

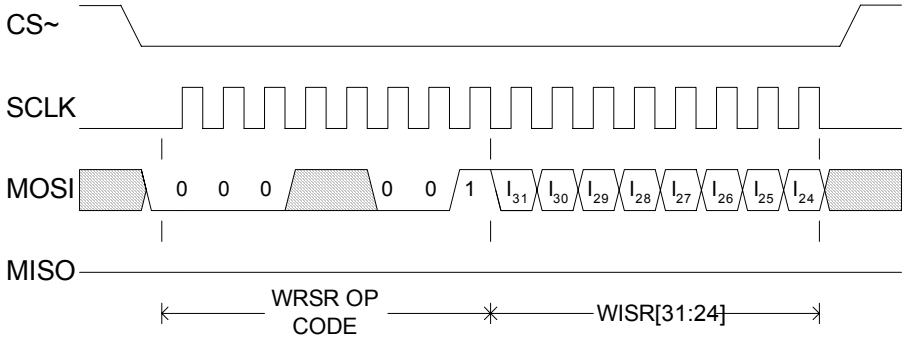

Figure 22 Example SPI Write Status Register Operation Using SPI Mode 0 and CS~ Framing

Detailed SPI Timing

Figures 23 through 25 provide detailed timing diagrams for several different methods of interfacing to the Pyxos FT Chip SPI port. Table 20 lists the values for the timing parameters.

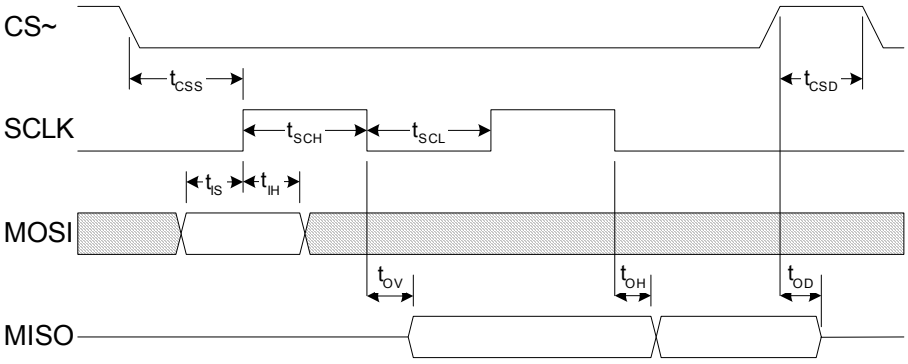

Figure 23 Detailed SPI Port Timing for SPI Mode 0 and CS~ Framing

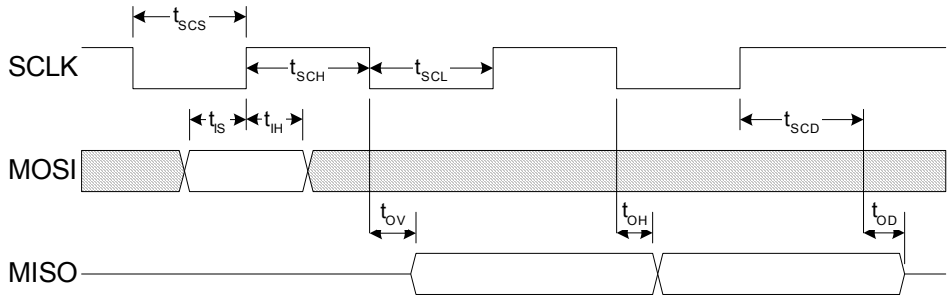

Figure 24 Detailed SPI Port Timing for SPI Mode 3 and Timeout Framing with **CS~** Held Active (Low)

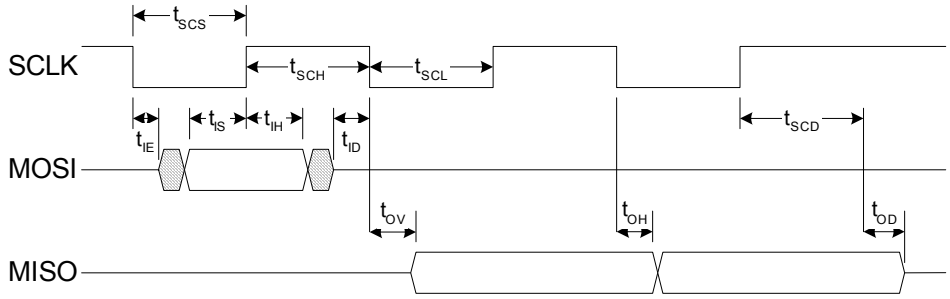

Figure 25 Detailed Timing for Two-wire (**MISO** and **MOSI** Tied Together at Microcontroller) SPI Port Interface with **CS~** Held Active (Low)

Table 20 SPI Interface Detailed Timing Parameters

Parameter	Description	Min	Max	Units
f_{SCLK}	Maximum SCLK frequency		1.25	MHz
t_{CSS}	CS~ setup to first rising edge of SCLK[1]	400		ns
t_{SCS}	SCLK setup to first rising edge of SCLK[2]	400		ns
t_{SCH}	SCLK high width	0.4	99	us
t_{SCL}	SCLK low width	0.4		us
t_{IS}	MOSI setup to rising edge of SCLK	12		ns
t_{IH}	MOSI hold after rising edge of SCLK	1		ns
t_{OV}	SCLK falling edge to MISO valid		12	ns
t_{OH}	MISO hold time after falling edge of SCLK	1		ns
t_{OD}	MISO disable time after end of transfer	400		ns
t_{CSD}	Time required to hold CS~ high between subsequent transfers[1]	400		ns
t_{SCD}	Time required to hold SCLK high to terminate a transfer[2]	103		us
t_{IE}	Time after first falling edge of SCLK before MOSI enabled[3]	0		ns
t_{ID}	Time on last MOSI bit before SCLK falling edge to MOSI disabled[3]	0		ns

Notes:

1. For **CS~** framed transfers.
2. For timeout framed transfers.
3. For shared **MOSI/MISO** transfers.

Pyxos Serial API

This section describes the Pyxos Serial API, and how to port it to a different microprocessor. The Pyxos Serial API interfaces are defined in the **psApi.h** file. The examples that ship with the Pyxos FT EVK contain an implementation of the Pyxos Serial API for an Atmel® ARM® AT91SAM7S64 microprocessor.

Table 21 lists the functions that are implemented by the Pyxos Serial API. These functions are described in more detail later in the chapter.

Table 21 Pyxos Serial API

Function	Description
psInit()	Initializes the Pyxos Serial Driver.
PsWrite()	Writes data to the Pyxos FT Chip at a specified address.
psRead()	Reads data from the Pyxos FT Chip at a specified address.
psIsInterruptSet()	Checks to see if the interrupt line of the Pyxos FT Chip is set or not.

Pyxos Serial API Functions

The following sections describe the functions of the Pyxos Serial API in detail.

psInit()

Initializes the Pyxos serial driver and put the Pyxos FT Chip in idle state.

Syntax:

```
void psInit(void);
```

Remarks: This function must be called once by a Pilot or Point application before any other functions of this API are called.

PS_PBUFFER

PS_BUFFER is a pointer to the data buffer used by **psRead()** and **psWrite()**.

Syntax:

unsigned char* const PS_PBUFFER;

Remarks: The size of this buffer is defined by the **PS_BUFFER_LENGTH** macro. This buffer will be provided and initialized by the driver.

The Point API defines the **PS_BUFFER_LENGTH** macro in the **PyxosPoint.h** file, based on the number of PNVs supported by the Point. The Pilot API defines the **PS_BUFFER_LENGTH** macro in the **PyxosPilot.h** file.

psWrite()

Writes data from the PS_PBUFFER location to the Pyxos FT Chip over the SPI bus.

Syntax:

```
int psWrite(const unsigned int addr, const unsigned int
length);
```

Remarks: Will block until this transaction has succeeded or failed. The *addr* parameter contains the SPI chip address to write to. See the Pyxos FT Chip data sheet for more information on this parameter.

The *length* parameter is the length of the data values in memory in bytes. This parameter does not contain the OPCODE and address byte, which are handled by this driver only. The parameter must be greater than 0, and must be less than or equal to the value of the **PS_BUFFER_LENGTH** macro.

The function returns 0 when all the values have been written to the chip successfully. If the function returns a non-zero value, then the operation failed.

psRead()

Reads data from the Pyxos FT Chip over the SPI bus and stores it at the PS_PBUFFER location in memory.

Syntax:

```
int psRead(const unsigned int addr,
           const unsigned int length);
```

Remarks: Will block until this transaction has succeeded or failed. The *addr* parameter contains the SPI chip address to read from. See the Pyxos FT Chip data sheet for more information on this parameter.

The *length* parameter is the number of bytes of data values to read from the chip. This parameter does not contain the OPCODE and address byte, which are handled by this driver only. The parameter must be greater than 0, and must be less than or equal to the value of the **PS_BUFFER_LENGTH** macro.

The function returns 0 when all the data has been successfully read from the chip. If the function returns a non-zero value, then the operation failed.

psIsInterruptSet()

Checks if the interrupt line of the Pyxos FT Chip is currently set or not.

Syntax:

```
int psIsInterruptSet(void);
```

Remarks: This function returns 1 if the chip is currently signaling an interrupt on the **IR** line. Otherwise, it returns 0.

Modifying the Platform.h File

The **platform.h** file defines the development environment, compiler, and target platform for the Pyxos application. It includes declarations for the built-in Neuron C types which are the basis for Pyxos types. The **platform.h** file is not project-specific. It is compiler and platform specific, and you can edit it to ensure maximum cross-platform and cross-compiler portability.

The **platform.h** file includes comments and porting instructions. It supports multiple targets and platforms in a single file using compiler directives. For each compiler and platform, the file contains a section defining all compiler-dependent and platform-dependent types and preferences. Each such section starts with an **#ifdef** *XXX* clause, where *XXX* indicates the compiler in use. Use the symbol used by your compiler, enforce the presence of the symbol by means of compiler arguments, or change the default implementation of the **platform.h** file so that it meets your target compiler and platform without conditional compilation. The compilation of **platform.h** will fail with an error if you fail to meet these requirements.

The master copy of the platform.h file is kept in the [*Pyxos FT EVK*]**Pyxos FT API\include** folder (**c:\Program Files\Echelon\Pyxos FT EVK\Pyxos FT API\include** by default). Edit this to add compiler definitions for your target compilers and platforms that persist for all projects.

In addition to type definitions, the **Platform.h** file includes the following macros that can be used by little-endian processors to convert data to and from Pyxos FT network (big-endian) format to the processor's native format:

- **NET_SWAB_WORD(***aWord***)**. Swaps the bytes in the 16-bit *aWord* value.
- **NET_SWAB_LONG(***aLong***)**. Swaps the bytes in the 32-bit *aLong* value.
- **NET_SWAP_WORD(***aWord***)**. Returns the byte swapped value of the 16-bit *aWord* value.
- **NET_SWAP_LONG(***aLong***)**. Returns the byte swapped value of the 32-bit *aLong* value.

These macros have no effect on big-endian hosts, and so may be used whether the host is a big-endian or little-endian processor.

Using Types

The Pyxos FT Interface Developer produces type definitions for the PNVs defined by the Point interfaces when it creates the Resources files. The Pyxos FT API also defines several data structures. For maximum portability, all types defined by the Pyxos FT Interface Developer and by the Pyxos FT API are based on a small set of nc* types. These types are host-side equivalents to the built-in Neuron C types, which are used as the basis for both Pyxos FT and LONWORKS network types.

For example, the **platform.h** file contains a type definition for a Neuron C signed integer equivalent type called **ncsInt**. This type must be the equivalent of a

Neuron C signed integer. On most target platforms, the **ncsInt** type will be defined as a signed char type.

In the following example, one or more PNVs on a Point use the **SNVT_switch** type from the standard resource files. The Pyxos FT Interface Developer creates the following entry in the Point's **Resources.h** file:

```
typedef struct
{
    ncuShort value;           ncsShort state;
} SNVT_switch;
```

Type definitions for structures assume a padding of 0 (zero) bytes, and a packing of 1 byte. Consult your compiler's documentation, and specify the appropriate compiler-options to enforce this. You can add appropriate compiler directives (#pragmas), if needed, to the compiler-specific section of the **platform.h** file.

Bitfield Members

Bitfield members are an exception to the above rules, since they are not based on the nc* type equivalents. Bitfields use three types: **Bitfield**, **SignedBitfield**, and **BitFieldWord**. These types are also defined in the **platform.h** file.

The **Bitfield** and **SignedBitfield** types support unsigned and signed bitfields packed in an 8-bit byte. The **BitfieldWord** type is used when the number of bits is more than 8 but less than 17. Bitfields are generated to work with compilers with both big-endian bitfield ordering and with little-endian bitfield ordering. The Pyxos FT Interface Developer inserts anonymous bitfields to achieve the correct alignment and padding. You must verify that bitfields developed with the Pyxos FT Interface Developer, and aggregates (structures, unions), meet the requirements for your target compiler to produce bitfield and aggregate data structures that match Neuron C's layout.

Enumerations

Enumerations are not produced by the Pyxos FT Interface Developer. The ANSI C language defines an enumeration (enum) type to be equivalent to a signed int type. The ANSI C language does not have a standard mechanism to enforce a particular size for an enumeration. Neuron C always uses 8-bit enumerations (because a Neuron C signed int type is an 8-bit scalar), so the **Resources.h** files generated by the Pyxos FT Interface Developer represent enumerations with the **ncsInt** type defined in the **platform.h** file.

7

Pyxos FT Protocol

This chapter describes the Pyxos FT Protocol, and how to directly access the Pyxos FT Chip to use the protocol.

Introduction

This chapter describes the Pyxos FT Protocol and how to directly access the Pyxos FT Chip to use the protocol. The simplest way to develop a Pyxos Point or Pyxos Pilot application is to use the Pyxos APIs provided with the Pyxos software. When using these APIs, you do not generally need to be concerned with the underlying Pyxos FT Protocol.

However, this chapter will be useful to you under any of the following circumstances:

- You want to modify or extend the Pyxos Pilot or Point API to better suit the needs of your application.
- You want to understand the protocol to aid debugging while developing your Pyxos application.
- You want to program directly to the Pyxos FT Chip, rather than use the Pyxos FT API. This may be necessary when implementing a Pyxos Point using a host microcontroller with limited hardware resources.

Pyxos FT Protocol Overview

The Pyxos FT protocol uses *time-division multiplexing* (TDM) to manage the communications between the Pilot and the Points. TDM is a type of digital multiplexing in which two or more simultaneous bit streams are encoded as sub-channels into a single bit stream by interleaving bits from the different bit streams. The combined bit stream is decoded at the receiving end. For a Pyxos FT network, the different sub-channels are the bit streams from the different Points on the Pyxos FT network, each communicating with the Pilot. Each of the sub-channels has a fixed bit rate, providing deterministic response for each of the sub-channels. The Pyxos FT Chips on the Pilot and the Points manage the interleaving of the channels.

The Pyxos FT protocol divides the time domain into several recurrent *timeslots* of fixed length, one for each sub-channel. A Pyxos *frame* consists of one window of time containing all the timeslots for all the sub-channels. Every frame includes 8 bytes of data sent by the Pilot to each Point, and 8 bytes of data from each Point to the Pilot.

The Pilot assigns a unique timeslot to each Point when the Point joins the network. Timeslot assignment is not necessarily sequential, but is randomized across the Points in the network. However, once a Point is assigned a timeslot, it uses that same timeslot until the Point leaves the network.

A Point sends data to the Pilot and receives data from the Pilot only during its assigned timeslot. The Pyxos FT Chip physically writes to a "write" timeslot and reads from a "read" timeslot, but a Pilot application sees both the write and read timeslots as a single timeslot assigned to the Point. Each timeslot carries 16 bytes of data—8 bytes in the read timeslot and 8 bytes in the write timeslot. The Pilot sends data to a Pyxos Point by writing to the Pilot Pyxos FT Chip memory. And, a Pyxos Point sends data to the Pilot by writing to the Point Pyxos FT Chip memory. The Pilot can also request the current value of a location in the Point Pyxos FT Chip memory by sending a poll request.

A Pyxos FT network can support up to 32 Points, and the number of timeslots is configurable, from 2 to 32. The Pilot can reserve timeslots for network expansion or it can assign as many timeslots as the current number of Points in the network. There must be at least as many timeslots as Points on the network.

Figure 26 shows a frame divided into four timeslots; the figure shows the physical write and read timeslots. Two of the timeslots are allocated to Points on the network, and the remaining timeslots are unused and are available for future Points that might join the network.

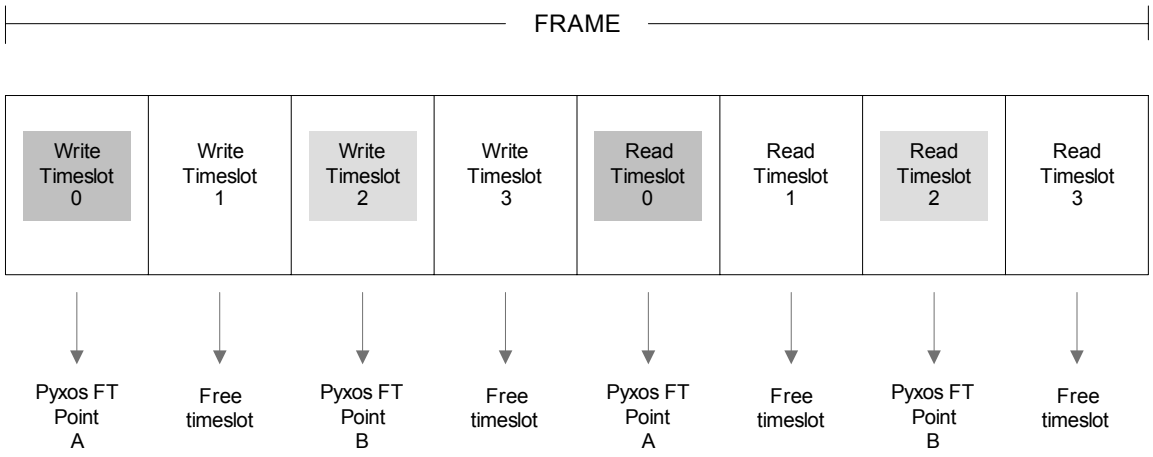

Figure 26 Timeslots in Each Network Frame

The number of timeslots within each frame directly determines overall network latency and response time. For a network with two timeslots (two write plus two read timeslots), that is, two Points, response time is less than 2 ms; for a network with a full 32 timeslots (32 write plus 32 read timeslots for 32 Points), response time is about 25 ms. It does not matter if a timeslot is allocated to a device or is a free timeslot—the network response time is the same.

The Pyxos FT Protocol ensures reliable delivery of data. The Pyxos FT Chip automatically acknowledges transactions, and new data cannot be sent until the previous data has been successfully delivered. Each packet has its own 18-bit CRC for error detection so that only valid data is delivered and acknowledged.

When a Pyxos Point is added to a network, it must be assigned to a timeslot. The Pyxos FT Protocol provides built-in mechanisms to facilitate registration of new Pyxos Points into a network. There are several registration methods supported including automatic registration, manual registration, and hardwired registration.

Memory and Registers

From a host microcontroller, the Pyxos FT Chip appears to be a serial memory with an SPI port. A host application controls a Pyxos FT Chip by reading and writing to memory on the chip. The host can read any location at any time. Writing to certain locations will invoke side effects, such as sending the value to another Pyxos FT Chip or configuring the Pyxos FT Chip.

The memory space is divided into two areas: control registers and data memory. On a Pilot, the data memory is used for the *frame memory*. The Pilot uses the frame memory to buffer communications with Points.

The data memory is reserved for *Pyxos Chip value* (PCV) memory on a Point. PCV memory contains a collection of PCVs addressed by *Pyxos Chip index* (PCI). The Pilot and Pyxos Points exchange data by reading and writing to a PCI—the value is the PCV. When a Pyxos Point writes to a PCI, the PCI and PCV are propagated to the Pilot. A Pilot can address a Pyxos Point in a timeslot, and write a PCV to a PCI on that Point over the network.

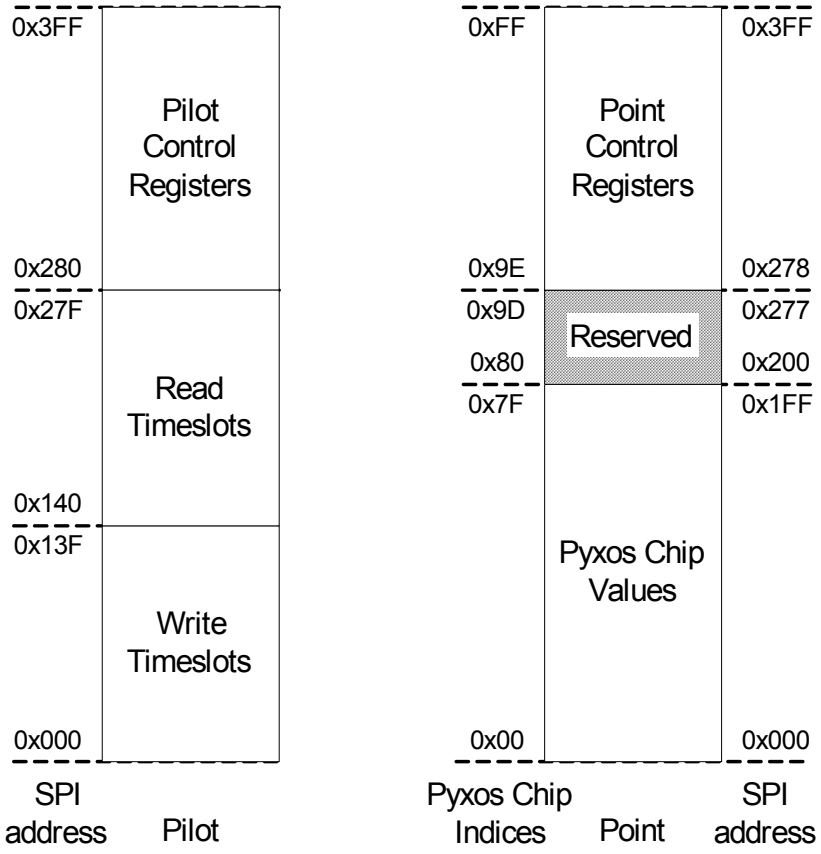

Figure 27 Pyxos FT Chip Memory and Registers

When accessed over the SPI interface, the memory space is byte addressable. Over the network, the memory space is addressed by PCI. A PCI addresses a PCV, which is a four-byte value. Bytes are packed big-endian into a PCI location (the most-significant byte is at the lowest address). Bits are numbered in the conventional manner (little-endian—the least-significant bit is at position 0).

To translate from a PCI to an SPI address, multiply the PCI by four. To access a particular byte within a PCI location, add the appropriate offset. To translate from an SPI address to a PCI, divide by four. Figure 28 shows an example of this mapping for PCI 0xF0.

PCI	0xF0			
bit index	31			0

SPI address	0x3C0		0x3C1		0x3C2		0x3C3	
bit index	7	0	7	0	7	0	7	0

Figure 28 Register Addressing From Network and From SPI Interface

There is an asymmetry in how the Pilot and the Pyxos Points view and address PCVs. From the Pilot's perspective, the Pyxos Points' PCV memories appear to be a large memory space addressed by timeslot and PCI. The Pyxos Points, however, can only see their own PCV memory space. For a Pilot to send a message, it must specify both the timeslot and the PCI, but a Pyxos Point only needs to provide the PCI.

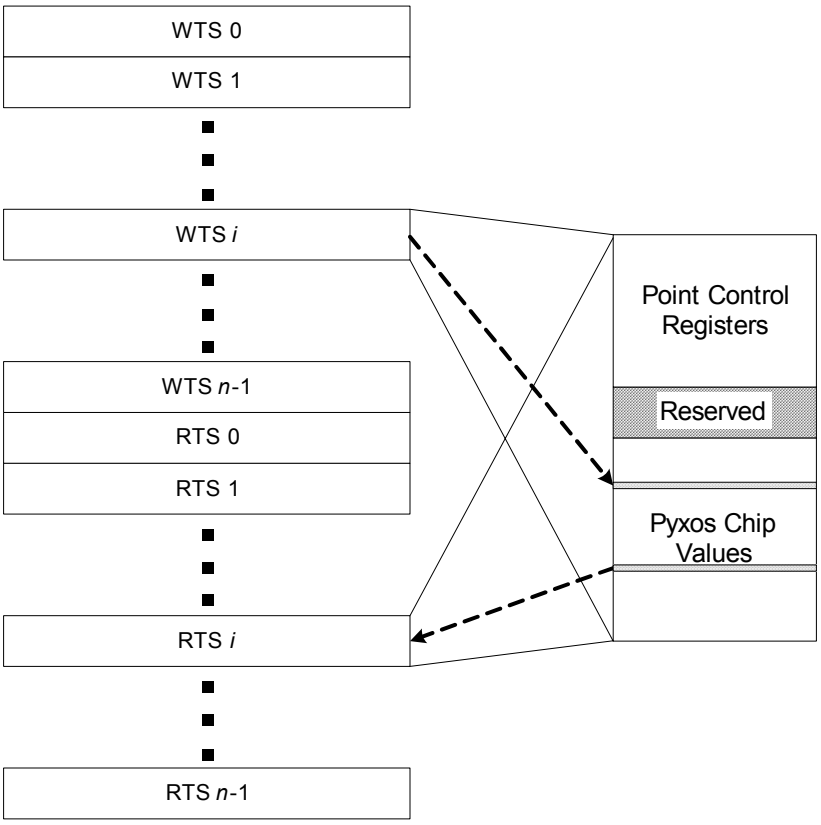

Figure 29 Pilot View of PCV Memory

The Pilot's frame memory is an intermediate buffer where the data is sent to or received from a Pyxos Point. The Pilot application maintains a local cache of some subset of the PCVs. When it updates a PCV, it must also send the update to the appropriate Pyxos Point by looking up the timeslot and writing the PCV to the PCI at that timeslot. When a Pyxos Point updates a PCV, the PCI and PCV

are automatically propagated to the Pilot's frame memory. The Pilot application must then accept the data and merge it into its copy of the PCVs.

Control Registers

Registers at PCI 0x9E and above (SPI addresses 0x278 and above) are used to control the operation of the Pyxos FT Chip. These registers configure the Pyxos FT Chip and network, provide network status, and control data transfers. Some registers behave differently on a Pyxos Pilot than on Pyxos Point. These registers are summarized in Table 22. Many of the control registers are discussed in detail in later sections.

Table 22 Pyxos Control Register Map

Index Addresses	SPI Address	Register Function	
		On Pilot	On Point
0xFF	0x3FC – 0x3FF	IDLE – Can be freely written and read.	
0xFE	0x3F8 – 0x3FB	Not used.	`UID2` – Three least significant bytes of unique ID. See *Configuration and Registration*.
0xFD	0x3F4 – 0x3F7	Not used.	`UID1` – Three most significant bytes of unique ID. See *Configuration and Registration*.
0xFC	0x3F0 – 0x3F3	CONFIG – Controls device type, network configuration, and slot assignment. See *Configuration and Registration*.	
0xFB	0x3EC – 0x3EF	SOFCNT – Frame counter. See *Protocol Statistics*.	
0xFA	0x3E8 – 0x3EB	RSTCNTL – Writing 0xde 0xad 0xbe 0xef to this register resets the Pyxos FT Chip. When updated over the network, an ACK is sent before performing the reset.	
0xF9	0x3E4 – 0x3E7	Not used.	PID2 – Four most significant bytes of program ID. See *Configuration and Registration*.

Index Addresses	SPI Address	Register Function	
		On Pilot	On Point
0xF8	0x3E0 – 0x3E3	Not used.	PID1—Four least significant bytes of program ID. See *Configuration and Registration*.
0xF7	0x3DC – 0x3DF	ISR—Interrupt status register. See *Interrupts*.	
0xF6	0x3D8 – 0x3DB	IENA—Interrupt enable. Controls which state changes results in asserting the **INT~** pin. See *Interrupts*.	
0xF5	0x3D4 – 0x3D7	Not used.	POLL—Pyxos FT Chip automatically sends requested PCIs. See *Transactions*.
0xF0 – 0xF4	0x3C0 – 0x3D3	Reserved.	
0xEF	0x3BC – 0x3BF	Not used.	DIO FCT 1—Configure and control **DIO** pins. See *DIO*.
0xE0 – 0xEE	0x380 – 0x3BB	Reserved.	
0xC0 – 0xDF	0x300 – 0x37F	CRC/MS—Counts CRC errors and missed slots for each timeslot. See *Protocol Statistics*.	
0xB0 – 0xBF	0x2C0 – 0x2FF	Reserved.	
0xA8 – 0xAF	0x2A0 – 0xBF	Pilot RCVD flags—Indicates received timeslots. See *Transactions*.	Point SENT flags—Indicates PCIs to send. See *Transactions*.
0xA0 – 0xA7	0x280 – 0x29F	Pilot SENT flags—Indicates timeslots to send. See *Transactions*.	Point RCVD flags—Indicates received PCIs. See *Transactions*.
0x9F	0x27C– 0x27F	Not used.	POINT_READY—Sent by a Point application after it has been configured to indicate that it is ready to accept updates. See the *Configuration and Registration* section on page 141 for more information.
0x9E	0x278– 0x27B	Not used	SET_POINT_ONLINE—Sent by the Pilot application when it is ready for the Point to start sending data. See the *Configuration and Registration* section on page 141 for more information.

Frame Memory

On a Pilot, the memory from SPI address 0x000 through 0x27F is used to buffer send and receive data from the Pyxos Points. The memory is mapped to timeslots in 10-byte increments. The 10 bytes are two *Pyxos Data items* (PDI) sent to the Pyxos Point at the addressed timeslot. Each PDI is a PCI (one byte) and a corresponding PCV (four bytes). This document often uses the following notation for a PDI: { PCI, PCV }, where PCI is a one-byte value, and PCV is a four-byte value. SPI addresses in the range 0x000 through 0x13F buffer the write timeslots; addresses 0x140 through 0x27F buffer the read timeslots. Figure 30 shows the layout of a timeslot buffer in the Pilot Pyxos FT Chip, and Table 23 shows the mapping of timeslots to SPI address.

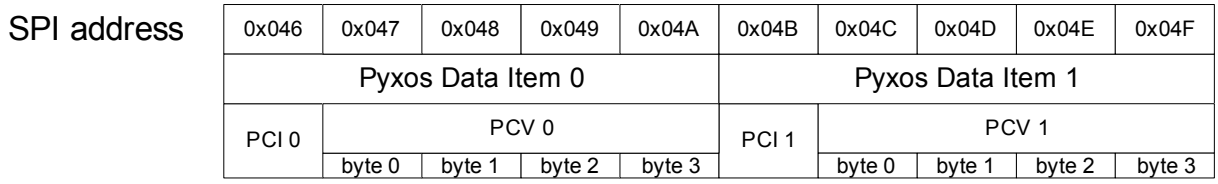

Figure 30 Example Timeslot Layout For Write Timeslot 7

Table 23 Timeslot SPI Address Assignments

Time-slot	Write Timeslot SPI Addresses	Read Timeslot SPI Addresses	Time-slot	Write Timeslot SPI Addresses	Read Timeslot SPI Addresses
0	0x000 – 0x009	0x140 – 0x149	16	0x0A0 – 0x0A9	0x1E0 – 0x1E9
1	0x00A – 0x013	0x14A – 0x153	17	0x0AA – 0x0B3	0x1EA – 0x1F3
2	0x014 – 0x01D	0x154 – 0x15D	18	0x0B4 – 0x0BD	0x1F4 – 0x1FD
3	0x01E – 0x027	0x15E – 0x167	19	0x0BE – 0x0C7	0x1FE – 0x207
4	0x028 – 0x031	0x168 – 0x171	20	0x0C8 – 0x0D1	0x208 – 0x211
5	0x032 – 0x03B	0x172 – 0x17B	21	0x0D2 – 0x0DB	0x212 – 0x21B
6	0x03C – 0x045	0x17C – 0x185	22	0x0DC – 0x0E5	0x21C – 0x225
7	0x046 – 0x04F	0x186 – 0x18F	23	0x0E6 – 0x0EF	0x226 – 0x22F
8	0x050 – 0x059	0x190 – 0x199	24	0x0F0 – 0x0F9	0x230 – 0x239
9	0x05A – 0x063	0x19A – 0x1A3	25	0x0FA – 0x103	0x23A – 0x243
10	0x064 – 0x06D	0x1A4 – 0x1AD	26	0x104 – 0x10D	0x244 – 0x24D
11	0x06E – 0x077	0x1AE – 0x1B7	27	0x10E – 0x117	0x24E – 0x257
12	0x078 – 0x081	0x1B8 – 0x1C1	28	0x118 – 0x121	0x258 – 0x261

13	0x082 – 0x08B	0x1C2 – 0x1CB	29	0x122 – 0x12B	0x262 – 0x26B
14	0x08C – 0x095	0x1CC – 0x1D5	30	0x12C – 0x135	0x26C – 0x275
15	0x096 – 0x09F	0x1D6 – 0x1DF	31	0x136 – 0x13F	0x276 – 0x27F

PCV Memory

On a Pyxos Point, the memory from PCI 0x00 through 0x7F (SPI addresses 0x00 through 0x1FF) is used to store PCVs. When a Point microcontroller writes to a PCI in this address space, the PCI and PCV are sent to the Pilot through the appropriate read timeslot. When the Pilot sends a new PCI and PCV pair to the Point in a write timeslot, a bit is set (in the Point's RCVD bits) to indicate that new data has arrived. The Point microcontroller can then access the new data at the indicated PCI.

The PCV memory is supports unidirectional transactions only—PCVs can be sent from or received in a given PCI, but attempting to use a PCI bi-directionally will result in corrupt data. There is no mechanism, for example, to prevent an application from writing to a PCI at the same time as the network.

Physical Layer

A cycle of communication where the Pilot exchanges a packet in each direction with each Point is referred to as a *frame*. Each frame starts with a *start-of-frame* (SOF) packet, followed by the *write timeslots* (WTSlots), and then the *read timeslots* (RTSlots). The WTSlots is divided into a sequence of individual packets, one per timeslot. Similarly, the RTSlots is divided into packets, one per timeslot.

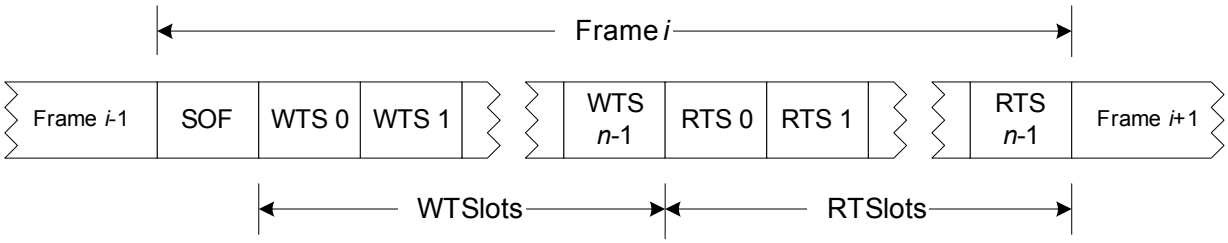

Figure 31 Pyxos FT Protocol Frame

A frame can have up to 32 timeslots, numbered from 0 to *n*-1, where *n* is the number of timeslots in the frame. The timeslot identifiers assigned to each Point are used to index into the WTSlots and RTSlots. The Point then accepts the packet in the WTSlots at its index and sends a packet out in the RTSlots at its index. For example, a Point with timeslot identifier 3 will read WTS 3 and write to RTS 3.

There are three types of packets in the Pyxos FT Protocol:

- SOF packet
- Write packet (in the WTSlots)
- Read packet (in the RTSlots)

The SOF and write packets are sent from the Pilot in one automatic transmission. The read packets are sent from each Point, one at a time. The SOF and read packets start with a preamble to allow for physical layer synchronization. All packets contain an 18-bit CRC.

The SOF has two primary purposes. It signals the start of a frame, and it indicates the length of the frame. The Points use this information to determine where to find their write packet, and when to send their read packet. The SOF packet includes a pattern that marks the SOF packet as distinct from other data. A field within the SOF encodes the number of timeslots in the frame.

Each data packet (WTS or RTS) contains a pair of Pyxos data items (PDI) and a 3-bit flags field. Each PDI consists of a one-byte Pyxos Chip index (PCI) and a four-byte Pyxos Chip value (PCV). The PCI uniquely identifies the data item within the Point. The flags field is discussed in the following section.

The Pyxos FT network runs at 312.5 kHz—or 3.2µs per bit. The length of each of the packet types in bits and time is shown in Table 24. There is a 2-bit gap between the WTS and the RTS.

Table 24 Packet Sizes

Packet Type	Bits	Time (µs)
SOF	87	278.4
WTS	111	355.2
Gap	2	6.4
RTS	130	416.0

The frame time depends upon the number of timeslots in the frame. The number of timeslots must be an even number between 2 and 32, inclusive. Table 25 shows the frame time and frame rate for all valid frame lengths.

Table 25 Frame Times

Timeslots	WTSlots time (ms)	RTSlots time (ms)	Frame time[1] (ms)	Frame rate (fps)
2	0.71	0.83	1.8	547
4	1.42	1.66	3.4	297
6	2.13	2.50	4.9	204
8	2.84	3.33	6.5	155
10	3.55	4.16	8.0	125
12	4.26	4.99	9.5	105
14	4.97	5.82	11.1	90.2

Timeslots	WTSlots time (ms)	RTSlots time (ms)	Frame time[1] (ms)	Frame rate (fps)
16	5.68	6.66	12.6	79.2
18	6.39	7.49	14.2	70.6
20	7.10	8.32	15.7	63.7
22	7.81	9.15	17.3	58.0
24	8.52	9.98	18.8	53.2
26	9.24	10.8	20.3	49.2
28	9.95	11.6	21.9	45.7
30	10.7	12.5	23.4	42.7
32	11.4	13.3	25.0	40.1

Notes:

1. Includes SOF (278.4µs) and gap (6.4µs).

Transactions

A transaction in the Pyxos FT Protocol is a successful, acknowledged transfer of data between the Pilot and a Pyxos Point. In any given frame, there are as many as 64 open transactions—one transaction from the Pilot to each Point, and one transaction from each Point to the Pilot. In addition, the Points can queue up future transactions for each PCI. The Pyxos FT Chip maintains several data structures and buffers to manage the transactions and keep track of the queues.

There are two basic transaction types—a *write transaction* from the Pilot to a Point, and a *read transaction* from a Point to the Pilot. There are several special forms of these transactions that are used during initialization (TID synchronization) or when there is no data to send (Idle transactions). The Pyxos FT Protocol also provides several higher-level services (block transfers, polling, and registration).

This section examines the on-chip data structures, transaction processing, the higher level services (except registration, which is discussed in a separate section), and how the programmer uses these resources effectively.

Packet Flags—ACK, ACKD TID, and TID

All data packets contain a flags field that is used for handshaking between the Pilot and the Points. There are three flags: TID, ACK, and ACKD TID.

The TID flag is a simple one-bit transaction identifier. The TID toggles for each unique packet sent in each timeslot. This flag prevents the same data from being delivered and accepted multiple times. The Pyxos FT Chip maintains the state of

the TID for all outgoing and incoming transactions. On outgoing transactions, it toggles the state of the TID for each new transaction it sends. On incoming transactions, it adopts the state of the TID from the transaction when it accepts the transaction; it will not accept a transaction if the incoming TID matches the state of the last accepted TID (except in special circumstances—more is said about this later).

The ACK flag is used to acknowledge the receipt of a previous packet—the ACK flag on a write packet acknowledges the previous read packet from the corresponding timeslot, and similarly on a read packet. This flag ensure that the transaction completes before going to the next transaction.

The ACKD TID identifies which transaction is being acknowledged. This ensures that an acknowledgement does not get confused with the wrong transaction.

SENT and RCVD Flags

The Pyxos FT Chip keeps track of data items to send, and new data items available from the network. It prevents new data from being accepted until the previous item has been consumed. It also prevents an attached host processor from attempting to send new data until the previous data has been consumed. This state is maintained in the SENT and RCVD flags described in Table 22. On the Pilot, the SENT and RCVD flags track the status of transactions for each timeslot. On a Point, the SENT and RCVD flags track the status of transactions for each PCI.

An application can read these flags to monitor the state of a transaction, but cannot directly write to them. They are only set when a transaction is initiated. However, an application can clear one of these flags; doing so will cancel an ongoing transaction. For more information on canceling a transaction, see the *Canceling a Transaction* section.

On the Pilot, there are a total of 32 SENT flags (S0 to S31) and 32 RCVD flags (R0 to R31) —one for each timeslot buffer. If the network is configured for less than 32 timeslots, then only the lower SENT and RCVD flags are used. The flags are mapped into the control register space with S0 to S7 and R0 to R7 at the most-significant bytes of their respective registers. S0 and R0 are at the least-significant bit position of those bytes, as shown in Figure 32. This allows the Pilot to address the base of the register and only shift out the flags for the number of timeslots in the network. For example, if there are only 8 timeslots in the network, then only the first byte of SENT flags and RCVD flags need to be monitored.

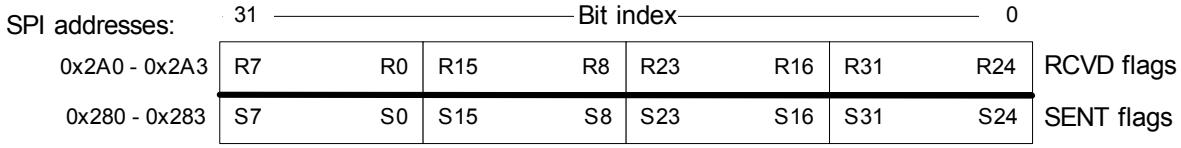

Figure 32 Pilot SENT and RCVD flags

On a Point, there are 256 SENT flags (S0 to S255) and 256 RCVD flags (R0 to R255) – one for each PCI. Some of these correspond to PCVs that are not available (PCIs 0x80 through 0x9F), or to PCIs that the Point application cannot directly send or receive (the registers from 0xA0 through 0xFF). However, the registers can be sent or updated in response to Pilot requests (for example a Poll

request—see the *Polling* section). A Point can have one outstanding read or write for each PCV in its memory (a scheduling algorithm then determines the order that the PCVs are sent). The location of these flags in the Point Pyxos FT Chip is shown in Figure 33.

The PCI SENT and RCVD flags ensure that data is transferred properly between the Pilot and Point applications. However, they only support unidirectional transactions. Attempting to use a single PCI to both send and receive data from a Point application will result in corrupt data. There is no mechanism, for example, to prevent an application from writing to a PCI at the same time as the network.

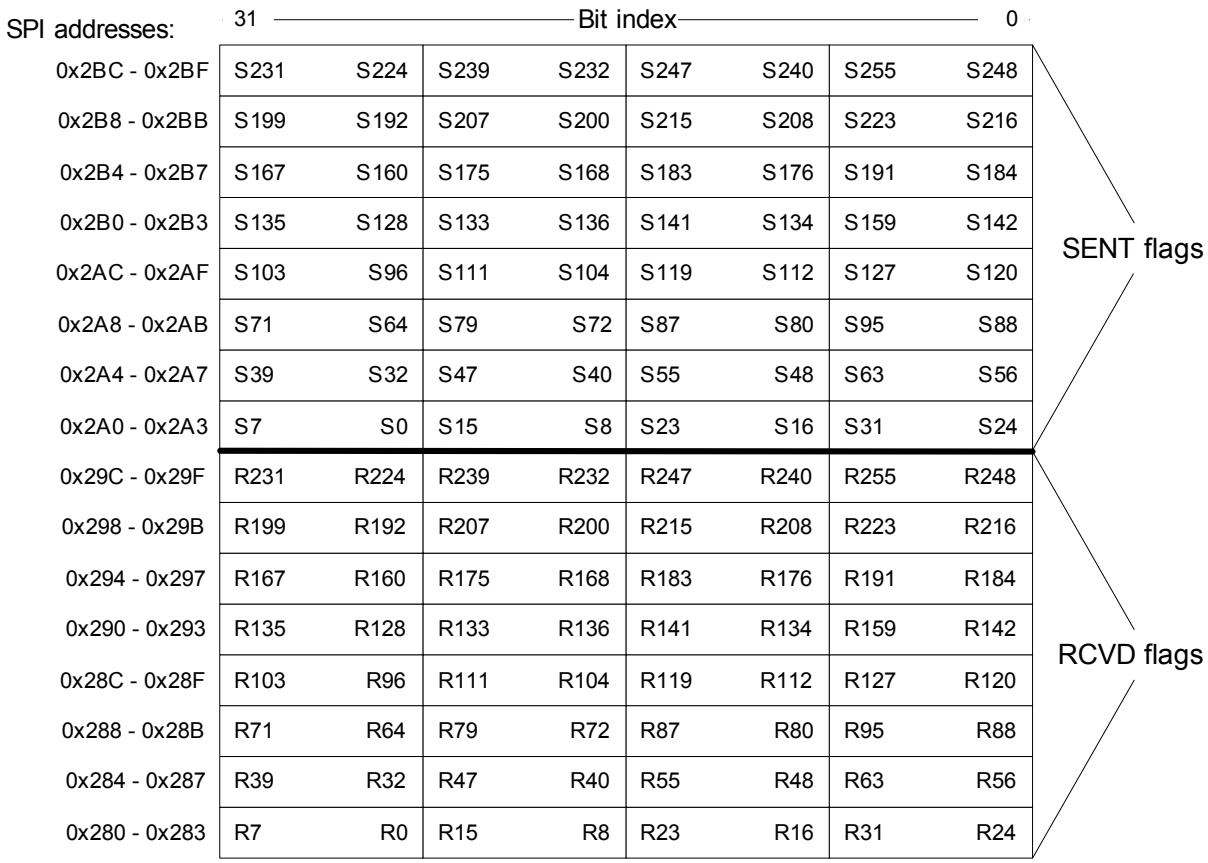

Figure 33 Point SENT and RCVD flags

Read and Write Hotspots

Transactions begin when a host (Pilot or Point) application writes to specific locations on the Pyxos FT Chip over the SPI port. The transactions complete when the application reads specific locations. These locations are *read* and *write hotspots*; there are such hotspots for both the Pilot (*timeslot hotspots*) and the Points (*PCV hotspots*).

Write hotspots are the last byte of a timeslot buffer (on a Pilot) or a PCV (on a Point). When an application writes to one of these locations, the Pyxos FT Chip will set the corresponding SENT flag. The write hotspots prevent the Pyxos FT

Chip from beginning to transmit the data item before the application has completed the data transfer.

On the Pilot, there is an additional requirement that the SPI transfer that writes to the timeslot write hotspot must also include at least one byte that is not 0xFF. If this condition is not met, the SENT flag will not be set. This allows the Pilot to transfer all of the write timeslots in one SPI transfer, using {0xFF, 0xFFFFFFFF} for any Pyxos Data Item that has no update.

Read hotspots are the last two bytes of a timeslot buffer or a PCV. When an application reads from either of these locations, the Pyxos FT Chip will clear the corresponding RCVD flag. The read hotspots prevent the Pyxos FT Chip from overwriting the buffer with new data from the network until the application finishes reading the data item. When an application reads from data memory over the SPI port, the Pyxos FT Chip caches the byte one beyond the currently addressed byte. This ensures that reading the next to last byte and the last byte of a timeslot buffer or PCV in one SPI transfer will return coherent data.

In general, an application should read or write an entire data item in one SPI transfer. This will prevent data items from being sent prematurely or being overwritten before being completely read.

When an application has less data than necessary to fill a PCV or timeslot buffer, it should right-align the data. For example, a two-byte quantity should be stored in the two least-significant bytes of a PCV. Or, if the Pilot application has only a single PDI to send to a timeslot buffer, it should use the right-most PDI (bytes 5 through 9); an unused PDI should always be {0xFF, 0xFFFFFFFF}. This ensures that the application will always read or write from the hotspots, and that it may be able to optimize SPI accesses in some cases to read or write only the relevant bytes.

Canceling a Transaction

An application can monitor the SENT and RCVD flags, but it cannot directly write to them. However, it can clear one of the flags by writing a 1 to the bit in the Pyxos FT Chip register that contains the flag. The application shares access to these flags with the Pyxos FT Chip protocol engine. Writing ones to clear a bit allows the application to selectively clear flags without affecting the state of other flags in the register. When the application clears a flag, it cannot determine whether the clearing was a result of its action or because the protocol engine completed its task.

Clearing a RCVD flag will not cancel a transaction, as the data has already been delivered by the time a RCVD flag is set. In addition, clearing a RCVD flag on the Pilot will result in the Pilot not returning an ACK—the sending Point will not be able to terminate the transaction and will require a reset. It is much safer to simply read and discard a data item if the application determines that it is no longer interested in it, than to clear a RCVD flag.

Clearing a SENT flag on the Pilot will attempt to cancel the transaction. The application cannot be certain whether it prevented delivery, but it will free the timeslot buffer to be used for another transaction. When the Pilot successfully cancels a write timeslot transaction, it will toggle its outgoing TID; however, the Point will not toggle its incoming TID. To re-establish communication, the TIDs must be re-synchronized (see the *TID Synchronization* section below).

The Point application cannot cancel an ongoing read transaction. Once the Pyxos FT Chip has begun the transaction, it will continue to attempt to complete it. However, clearing a SENT flag on a Point will attempt to cancel a queued transaction for the corresponding PCI. The application cannot be certain whether it actually cancelled the delivery, or the Pyxos FT Chip simply began the transaction. Clearing a Point SENT flag will not disrupt any ongoing or future transactions.

Idle Transactions

When the Pyxos FT Chip has no data to send, it sends an *idle timeslot*. An idle timeslot consists of two idle PDIs; the *idle PDI* is { 0xFF, 0xFFFFFFFF }. Sending and receiving an idle timeslot is referred to as the *Idle Transaction*.

The Idle Transaction has several useful properties. The Idle Transaction is always accepted by the destination node, but is never acknowledged. Since the Idle Transaction is always accepted, it is a special case of a TID synchronizing transaction (see *TID Synchronization*). The Idle Transaction will never set or clear a SENT or RCVD flag; it does not require any servicing from the application to ensure its use, either to send or receive.

In on-demand Pilot mode (see *Configuration and Registration* below), the Pilot application can explicitly send an Idle Transaction if it has no data to send. This allows the network to run for a single cycle, allowing the Points to send any data they have.

A PDI with a 0xFF PCI is not an idle PDI, unless the PCV is also 0xFFFFFFFF. If a Pilot attempts to send a timeslot where both PCIs are 0xFF, but one of them is not the idle PDI, then the SENT bit will get set (because one of the bytes is not 0xFF). However, the Point will still treat the timeslots as idle, and will not acknowledge the transfer. The Pilot application must cancel the transaction to clear such a transaction.

TID Synchronization

Both a sending and receiving node must agree on the state of the TID before they can successfully exchange data. After reset, the TIDs are set to an arbitrary value. A *TID synchronizing transaction* is required to ensure that the TIDs agree. Such transaction are required at other times as well, such as after the Pilot cancels a transaction.

A TID synchronizing transaction is any transaction where both PCIs are at least 0xF0. The receiving node never checks the state of the TID flag before accepting such transactions. A Point always accepts a TID synchronizing transaction—the RCVD flags for these PCIs never get set (they correspond to control registers; writing to them takes effect immediately without notifying the application). On the Pilot, they are accepted as long as the Pilot application has ensured that the RCVD flag is clear by reading the timeslot buffer.

When a Pyxos FT Chip accepts a TID synchronizing, it adopts the TID flag as the state of the last incoming TID. After this, the sending and receiving nodes agree on the state of the TID, and all subsequent transactions can proceed normally.

After assigning a Point to a timeslot, the Pilot and Point send TID synchronizing transactions to each other to establish reliable communication before performing any other transactions. If the Point is an unhosted Point, the Pilot will need to

Poll a register in the 0xF0 – 0xFF range to ensure a TID synchronizing transaction in the read timeslot.

Although the Idle Transaction is a TID synchronizing transaction, an application may attempt to send data before an idle transaction has occurred. This is especially true if the Pilot is using on-demand Pilot mode. The best approach is for both Pilot and Point applications to explicitly send a TID synchronizing transaction before any other transactions.

The Pilot can use a poll (PCI 0xF5) of the UID of the Point as a TID synchronizing transaction. This will ensure that the write timeslot TID is synchronized, but it does not ensure that the read timeslot TID will be synchronized. This will work for an unhosted Point, however, the application on a hosted Point may attempt to queue up a transaction before the UID poll occurs.

For hosted Points, the application should send its UID as the first transaction after receiving a slot assignment. It can determine when it has received its slot assignment by monitoring the CFG bit in the CONFIG register—see the *Configuration and Registration* section for more information on this. And because the round-robin scheduler may select other PCIs first, the Point application should not attempt to send any other PCIs before it sees the SENT flags of the UID registers clear.

Because the Pyxos FT Chip accepts TID synchronizing transactions regardless of the state of the TID, it is possible for the Pyxos FT Chip to see and accept such a transaction multiple times. For example, if the acknowledgement is lost on the first attempt due to noise, the sending node will continue to send the transaction until it sees the acknowledgement. In general, this is harmless. However, when the Pilot sends a Poll request, it may see multiple responses. The Pilot should be able to tolerate multiple responses to a Poll request.

Write (Pilot-to-Point) Transactions

A write transaction starts with the Pilot application's request to send data and completes when the Point accepts the data. The steps involved in this transaction include:

1. *Initiate*: the Pilot application writes to a write timeslot buffer over the SPI bus.
2. *Send*: the Pilot Pyxos FT Chip sends a write timeslot buffer to a Point.
3. *Accept*: the Point Pyxos FT Chip accepts data, returns an acknowledgement, and notifies the Point application of new data.
4. *Read*: the Point application reads new data.
5. *Terminate*: the Pilot Pyxos FT Chip receives acknowledgement and clears the transaction.

These steps are illustrated in the figure below for a simple case—the Pilot sends a single PDI (PCI*k*, PCV*k*) to the Point in timeslot *j*. PCI*k* is in the range of 0x00 to 0x9F and the ISR updates are not shown. These steps are described in more detail in the following sections.

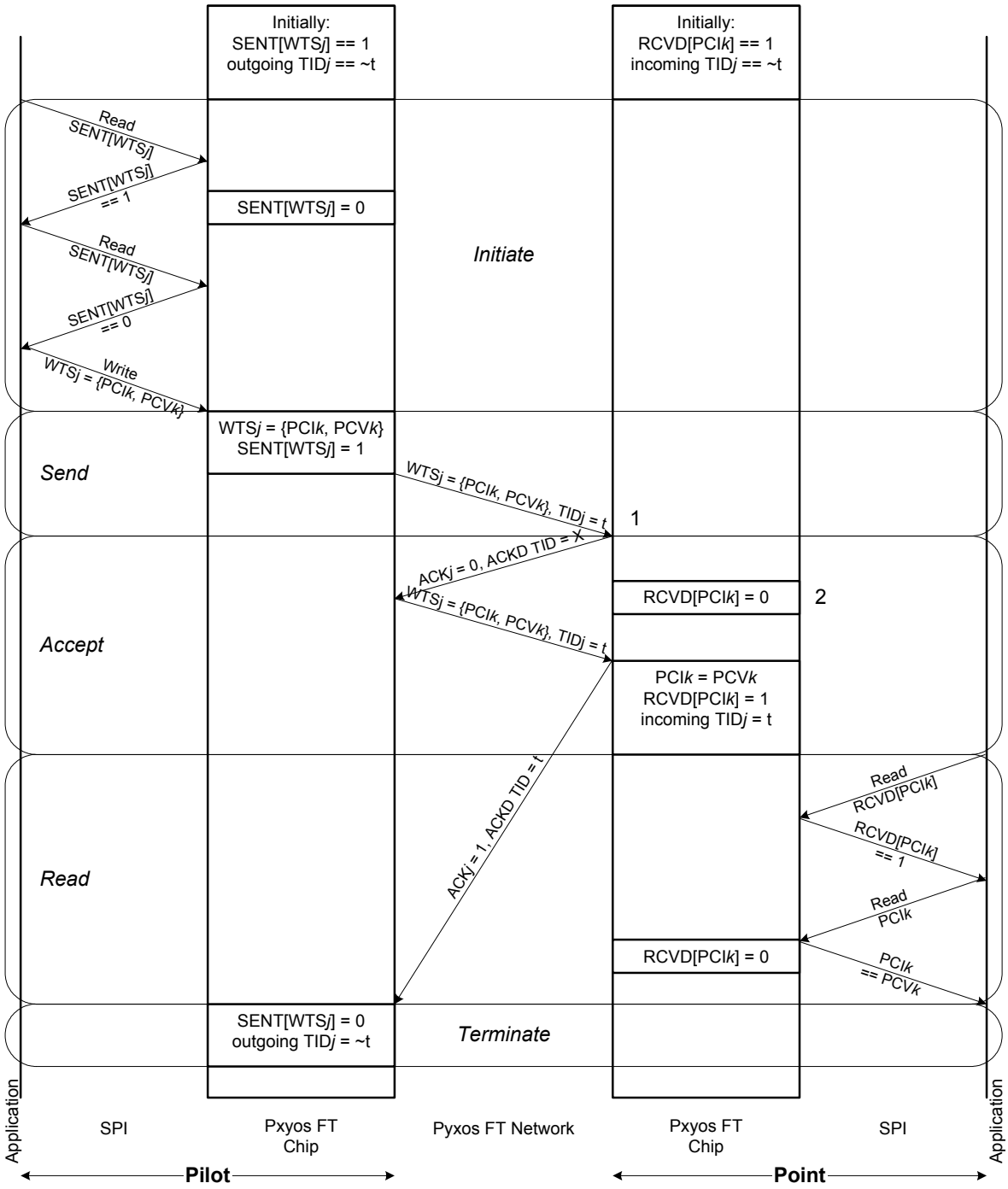

Figure 34 Example Write Transaction

Notes:

1. Packet rejected (ACKj=0) because RCVD[PCIk]==1.
2. RCVD[PCIk] cleared by previous transaction.

Initiate

When the Pilot writes to a write timeslot buffer over the SPI bus, the corresponding SENT flag is set. If the SENT flag is already set, then the Pyxos

FT Chip will ignore any attempt to write the timeslot buffer, and the write data is discarded. The Pilot application must ensure that the previous transaction has cleared by checking the SENT flag before writing to the timeslot buffer.

Not all SPI writes to a write timeslot buffer will set a SENT flag. The SPI write must include a write timeslot hotspot and must contain at least one byte that is not 0xFF. See the *Read and Write Hotspots* section for more details on this.

Send

Once the SENT flag has been set, the Pilot Pyxos FT Chip will send the data in the timeslot buffer in the next frame. It continues to send the data with the TID flag until it receives (on the corresponding read timeslot) an ACK where the ACKD TID matches its TID.

If the Pilot never receives an acknowledgement, it will continue to send the timeslot buffer. However, if the addressed Point cannot respond—for example, if it has been disconnected from the network—then this condition may persist indefinitely. The Pilot application can monitor for this and establish a transaction timeout. If it does not receive an acknowledgement from the Point within the timeout period, it can recover by canceling the transaction, sending a reset command, and re-establishing communication with the Point. This is explained in more detail in the *Reset Handling and Error Recovery* section below.

Accept

A Point Pyxos FT Chip examines every write packet in its timeslot. It will accept the packet if the PCIs and TID of the incoming packet meet one of two conditions:

- The PCIs are both 0xF0 or greater.
- The RCVD flags for the PCIs are both clear and the TID is different from the last accepted TID.

When the Point Pyxos FT Chip accepts a write packet, it performs the following actions:

1. For any PCI less than 0xFF, copy the PCVs to the appropriate location as indicated by the PCIs.
2. Set the RCVD flag for any PCI in the packet that is less than 0xA0.
3. If any PCI RCVD flag gets set, set the ISR RCVD flag.
4. Update the local copy of the last received TID.
5. If either PCI is not 0xFF, set the ACK and ACKD TID flags for the corresponding read timeslot.
6. If the PCI is a POLL command, the appropriate SEND bits are immediately set.

If both PCIs are at least 0xF0, the packet is always unconditionally accepted. See the *TID Synchronization* section for more information. If both PCIs are 0xFF, the packet is treated as the idle packet. See the *Idle Transactions* section for more information.

PCIs greater than 0xA0 are in the register space. The Point Pyxos FT Chip handles these PCIs directly, without Point application intervention. So, the RCVD flag does not need to be set, and no interrupt will be sent to the Point application. Two registers (0x9F and 0x9E) are defined below 0xA0. These registers are not handled directly by the Pyxos FT Chip, but rather are used as

control registers accessed by the Point application. Therefore, the RCVD flag and an interrupt are set for registers in this range.

Writing to the CONFIG register, however, can disrupt communications. It is possible to prevent the Point from responding properly to the transaction when writing to this register. For example, if the Pilot attempts to de-configure a Point by clearing its CFG bit (see the *Configuration and Registration* section), the Point will not return an ACK. The Pilot will then need to cancel the transaction.

A better method of de-configuring a Point is to send it a soft reset (writing the reset pattern to the RSTCNTL register). The Point Pyxos FT Chip will always acknowledge this transaction before performing the reset.

Read

The Point application is notified of new data in one of two ways: either through monitoring the ISR or monitoring specific RCVD flags.

The ISR RCVD flag is set when new data has arrived (see above), so the application can monitor this single flag to wait for new data. In fact, the interrupt output of the chip can be configured (using the IENA register) to monitor for changes to the ISR RCVD flag. Once the application learns that the ISR RCVD flag has been set, it must then determine which PCI RCVD flag has been set by reading the RCVD flags.

If, however, the Point application is only concerned with a few PCIs, it may be more convenient to watch the RCVD flags for those PCIs. This avoids the extra step of searching for which RCVD flag was set.

Once the application determines that it has new data at a given PCI, it can read that data at any time. Since the RCVD flag is set, the Pyxos FT Chip will not accept new data until the Point application clears the RCVD flag, which it does by reading the data. See the *Read and Write Hotspots* section for more information on clearing the RCVD flag by reading the data.

Terminate

As soon as the Pilot Pyxos FT Chip recognizes an ACK for the TID that it is sending, it terminates the transaction. When it terminates a transaction, it performs the following steps:

1. Clear the SENT flag for the timeslot.
2. Set the ISR SENT flag.
3. Toggle the local copy of the TID.

Since the SENT flag has been cleared, it will no longer attempt to send the data from the timeslot buffer. The Pilot application can monitor the ISR SENT flag or the timeslot SENT flag to determine when the transaction has completed. It can then initiate a new transaction. If no new transaction is available, the Pilot Pyxos FT Chip will send the idle packet.

Read (Point-to-Pilot) Transactions

Read transactions differ from write transactions because of the way PCVs are queued and scheduled on a Point, and because of the way the Pilot acknowledges receipt of a read timeslot.

A Point Pyxos FT Chip can queue a read transaction for each PCV (up to 128). It contains a built-in round-robin scheduler to determine which PCVs to send next. When it selects one or two PCVs to send, those PCVs are copied to a transmit buffer, where they remain until the read transaction completes. This buffer allows the Point application to queue a second update to the PCV as soon as the first update has been copied to the buffer.

Read transaction acknowledgements require Pilot application intervention – the ACK is not sent until the application has seen the data and read it.

These differences lead to a slightly different flow for read transactions:

1. *Enqueue*: The Point application writes to a PCV over the SPI bus.
2. *Send*: The Point Pyxos FT Chip selects up to two PCVs to send, copies them to the transmit buffer, and sends the read timeslot to the Pilot.
3. *Accept*: The Pilot Pyxos FT Chip accepts the data and notifies the Pilot application, but does not acknowledge receipt.
4. *Read*: The Pilot application reads and processes the data.
5. *Acknowledge*: The Pilot Pyxos FT Chip acknowledges the read transaction.
6. *Terminate*: The Point Pyxos FT Chip receives acknowledgement and clears the transaction.

These steps are illustrated in Figure 35 for a simple case—the Point sends a single PDI (PCI*k*, PCV*k*) to the Pilot in timeslot *j*. PCI*k* is in the range of 0x00 to 0x9F and the ISR updates are not shown. These steps are described in more detail in the sections following Figure 35.

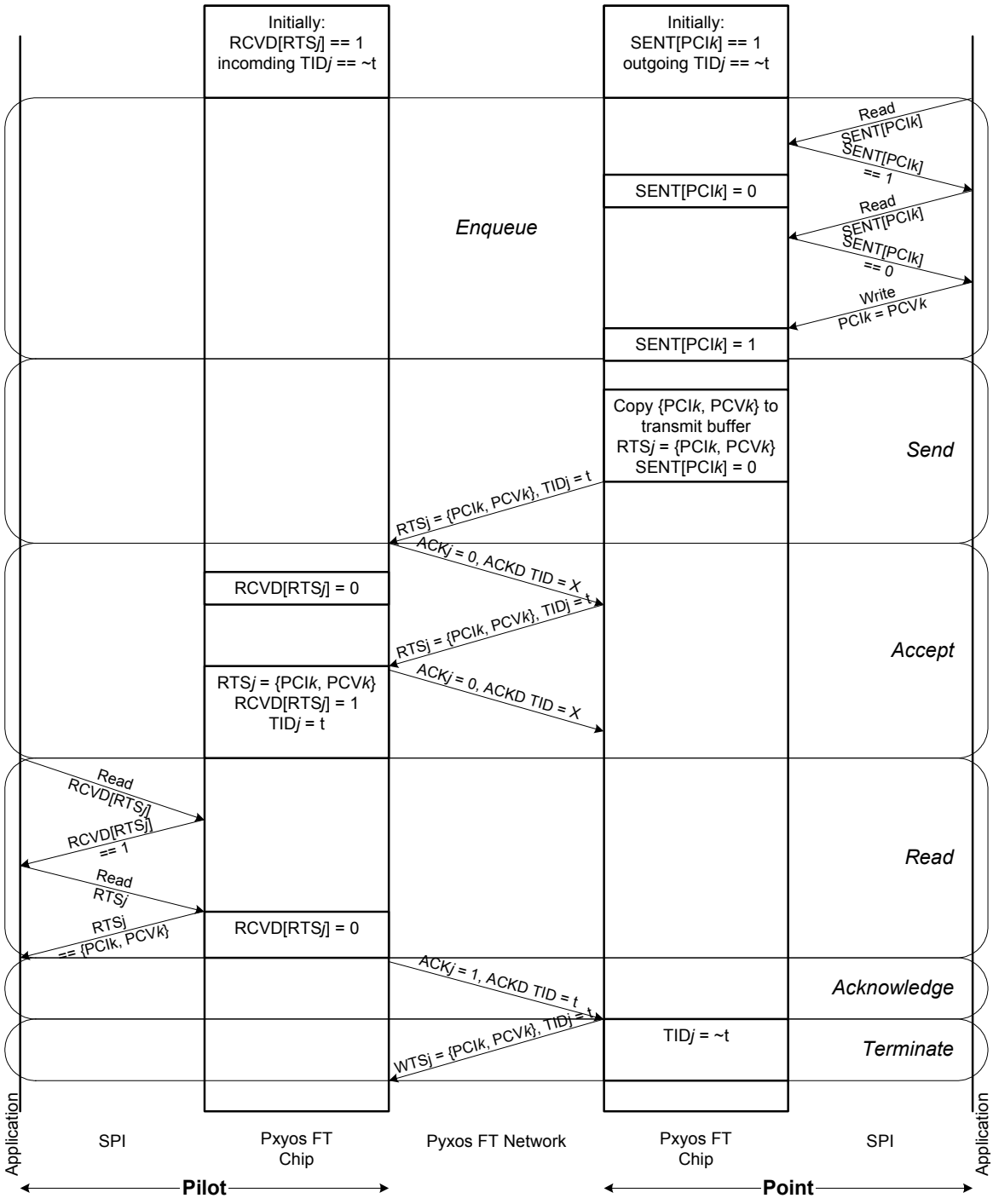

Figure 35 Example Read Transaction

Enqueue

When a Point application writes to PCV memory, the PCV is marked for sending to the Pilot in a read timeslot. The Point Pyxos FT Chip marks the PCV by setting the corresponding SENT flag. If the Point application writes to a control register (PCI 0xA0 or greater) other than one of the UID registers (0xFD and 0xFE) or POLL register 0xF5, then no SENT flag is set. In general, writing to a control register does not cause the register to be sent to the Pilot. The exception

for the UID registers is used as part of the registration process (see the *Configuration and Registration* section later). The POLL register exception allows the Point applications to send poll requests. The Pilot can still read the Point control registers using the Polling mechanism (see the *Polling* section later).

The Point Pyxos FT Chip will only set the SENT flag if the SPI write is to the PCV write hotspot. See the *Read and Write Hotspots* section for more information on this.

If the SENT flag for the PCI is already set when the Point application attempts to write to the PCI, the Pyxos FT Chip will ignore the data. The application must ensure that the SENT flag is clear before attempting to write to a PCI.

Send

The Point Pyxos FT Chip uses a round-robin scheduler to select PCIs to send on the next read timeslot. Once it selects the PCIs, it copies those values to its transmit buffer, clearing the SENT bits for those PCIs. Those values remain in the transmit buffer until the Point Pyxos FT Chip receives an acknowledgement.

The round-robin scheduler begins at the PCI after the last PCI sent, and increments through the entire PCV memory space, including the register space, wrapping around to PCI 0x00 after 0xFF. It stops either when it finds two PCIs with SENT flags set or when it reaches its starting Point.

If it reaches the starting Point without finding two PCIs, it rounds off the starting Point to the next mod 8 PCI. For example, if the last PCV sent was at PCI 0x1A, and there are no other PCIs to send, when it runs through SENT flags and reaches 0x1A again, it will then set the next start Point to be PCI 0x20.

The scheduler is invoked after the Point Pyxos FT Chip has processed its incoming write timeslot (if the transmit buffer is empty). This allows the Pyxos FT Chip to service a Poll request in the same frame in which it was made. It also allows the Point application time to process the write timeslot before the read timeslot.

Once the scheduler has found PCIs to send, the Pyxos FT Chip collects those PCIs and moves them to the transmit buffer. If only one PCI is found, the other Pyxos Data Item is filled with PCI 0xFF. If no PCIs are found to send, then both PDIs are filled with PCI 0xFF.

When the PCIs are copied to the transmit buffer, the SENT flags are cleared, and the ISR SENT flag is set. The Point application can monitor these flags to determine when it is allowed to queue another update.

Accept

The Pilot Pyxos FT Chip accepts the read timeslot if the following conditions are met:

- The PCIs are both 0xF0 or greater, or the TID is different from the last received TID.
- The RCVD flag for the timeslot is clear.

When it accepts the read timeslot packet, the Pilot Pyxos FT Chip performs the following:

1. If either PCI is not 0xFF, set the RCVD flag for the read timeslot and set the ISR RCVD flag.
2. Update the local copy of the last received TID.

Any packet that has both PCIs at 0xF0 or greater is accepted even if the TID does not indicate a new packet. This is used for TID synchronization for the read timeslots. See the *TID Synchronization* section for more details.

If both PCIs are 0xFF, the packet is treated as the idle packet. The Pilot Pyxos FT Chip will accept this packet as long as the timeslot buffer is not full (the RCVD flag is clear), but it will not set the RCVD flag. Consequently, this packet will never be acknowledged. See the *Idle Transactions* section for more details.

Read

The Pilot application can determine when new data has arrived in a timeslot by monitoring the RCVD flags, or the ISR RCVD flag. Typically, the Pilot application will check the RCVD flags at one or more of the interrupts (MORT or EORT), and use the flags as bitmasks to read in all of the new timeslot buffers. See the *Interrupts* section below for more information on this.

It is important for the Pilot to be synchronized to the network so that it clears the RCVD flags in the window between an RTS and its corresponding WTS. This is done using the MORT and EORT interrupts. Otherwise, it is possible for a TID synchronizing transaction to be sent from a Point continuously. If the Pilot clears the RCVD flag after the WTS for the timeslot, and before the next RTS, the ACK will not go out; but the Pyxos FT Chip will re-accept the transaction when it sees the RTS, because it ignores the TID, again clearing the ACK. This will continue until the Pilot application clears the RCVD flag after the RTS and before the WTS.

Acknowledge

When the Pilot application reads a read timeslot over the SPI bus, the Pilot Pyxos FT Chip performs the following:

1. If the RCVD flag for the timeslots is set, send an acknowledgement with the appropriate ACKD TID on the next write timeslot.
2. Clear the RCVD flag for the timeslot.
3. Update the local copy of the last received TID.

These actions are only performed when one of the read timeslot hotspots are read. See the *Read and Write Hotspots* section below.

Terminate

When the Point Pyxos FT Chip receives an ACK for the TID that it is sending, it terminates the transaction. It stops sending the contents of the transaction buffer, and it restarts the round-robin scheduler. If new PCIs are found for sending, it starts a new transaction. Otherwise, it sends the Idle Transaction.

The SEND flag was already cleared at the beginning of the transaction (at the Send step), when the PCVs were copied into the transmit buffer. There is local state in the transmit buffer to track the status of the transaction. However, this state is not available to the Point application, and so the Point application cannot determine when a transaction successfully completes. If the transaction does not

complete, the Pilot will determine that it has lost communication with the Point and attempt to recover. See the *Reset Handling and Error Recovery* section below for more information.

Block Transfers

A Pyxos Chip value is a four-byte quantity. If a larger quantity is required, block transfers can be used to ensure delivery of coherent values larger than four bytes.

A *block* is simply a contiguous group of PCIs. The lowest numbered PCI is considered to contain the least significant bytes of the block.

To send a block from Pilot to a Point, or vice-versa, the following rules must be observed:

- The block must be sent in PCI order, starting with the lowest numbered PCI.
- The entire block must always be transferred; sending only a subset of the block is not allowed.
- For Point-to-Pilot transfers, the Point application must write the entire block to the Pyxos FT Chip in one SPI transfer.
- Updates of a given block must be treated atomically—a subsequent block transfer cannot start until all PCIs of the previous transfer have been sent (all the RCVD flags for the PCIs in the block must be clear).

The Point application will see the PCVs arrive in order from the lowest PCI to the highest.

The Pilot application will generally see the PCVs arrive in increasing PCI order. However, the first PCI may not be the lowest PCI. The first PCI to arrive may be in the middle of the block, and when the highest numbered PCI of the block arrives, the next PCI will be the lowest PCI of the block. In very unusual cases some PCVs may be skipped and sent out of order. The Pilot application must be prepared to put the block back into proper order.

Polling

The Pyxos FT Protocol provides a special mechanism that allows the Pilot to request an update to a given PCI (a *Poll Request*). The Point Pyxos FT Chip automatically recognizes this and sends the requested updates with no application intervention required. A Point can also request an update from the Pilot, but the Pilot application must recognize the request and perform the update.

The Pilot application sends a Poll Request to a Point by using a POLL command, as described in Table 22. A POLL command consists of the PCI 0xF5 and up to four Poll Requests; each Poll Request includes a PCI to be polled (see Figure 36). If less than four PCIs are to be polled, the remainder of the PCV should be filled with 0xFF, which is interpreted as a null Poll Request.

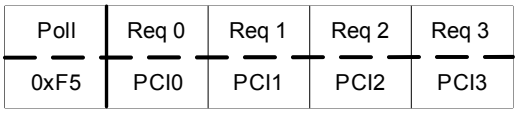

Figure 36 POLL Command Structure

The POLL command sets the SENT flag for the PCIs in the command. Any PCI except 0xFF will set the SENT flag and cause the value to be returned, even values in the register space. The Pilot can use this to query the configuration, UID, or PID of a Point.

The order in which the values are returned is indeterminate. It depends upon the state of the Point's round-robin scheduler, and on what other values have already been scheduled. You cannot perform two POLL commands in a single timeslot. However, multiple POLL commands can be sent in successive frames.

The Pilot application will always get at least one response to a Poll Request, but it may get more than one. If the Point application sends a PCI at the same time that the Pilot is sending a Poll Request for the PCI, the Point may send the PCI once or twice, depending upon the timing of the requests, and how quickly the value is sent.

Polling PCIs that a Point application may asynchronously update should be avoided. If the Pilot polls a PCI that the Point application plans to update, a race condition could exist that will prevent the Point from successfully updating the PCV. This occurs if the Point application reads the POINT_SENT flags and determines that the value can be sent, but the Pyxos FT Chip processes a poll request for that same PCI, before it has a chance to write the PCV to the Pyxos FT Chip, causing the SENT flag to be set. When the Point application tries to write the PCV to the Pyxos FT Chip, the write will fail silently because the SENT flag is set. To avoid this situation, the Pilot application should restrict polling to the register space, and only poll those registers that the Pyxos FT Chip does not send on its own. The Pilot application should receive all values that the Point has written after going online (see *Point Configuration and Registration* below), so there should be no need to poll any data values.

If the Pilot application sends multiple Poll Requests of the same PCI in multiple frames, no more than one response is guaranteed. If all of the Poll Requests reach the Point Pyxos FT Chip before the first can be serviced, then only one response will be made. However, if the requests are sent only after receiving a response, then each request will receive a response.

A Point can also poll the Pilot for an update to a PCI. If the Point writes to the Poll register (0xF5), a POLL command will be sent to the Pilot. However, the Pilot Pyxos FT Chip does not contain an image of the Point's PCV memory (it only has timeslot buffers). Instead, the Pilot application maintains this image, so the Pilot application must respond to the request. So, the Pilot POLL command is defined by convention only to be the same as a Point POLL command. The Pilot application must then interpret and service the request.

Configuration and Registration

Before a Pyxos FT Chip can be used for communications, it must be configured and, when used in a Point, registered. To configure the Pyxos FT Chip, the `CONFIG` register (SPI addresses 0x3F0 to 0x3F3) must be set correctly.

Configuration consists of declaring the Pyxos FT Chip to be either a Pilot or a Point, and then providing additional configuration information that depends on whether it is a Pilot or a Point.

Registration is the process that allocates a timeslot to a Point. The timeslot must be unique among all of the Points on the network. The application must

determine how best to map Points to timeslots. However, the Pyxos FT Protocol provides several built-in mechanisms to facilitate timeslot assignment.

- Automatic discovery—If the application does not have any constraints on the mapping, the Pyxos FT Protocol can randomly assign Points to any available timeslot.
- Manual registration—If the application requires the user to identify devices, the Pilot can assign timeslots as a user identifies devices.
- Hardwired registration—If the application can assign timeslots based on unique information within each Point such as an identifier provided by a wiring harness or other Point physical input, each Point can assign itself a timeslot.

An application can use any combination of these methods. The Pilot can maintain a free timeslot map, which it uses for the automatic discovery method. If there are any hardwired Points in the network, it can eliminate their timeslots from the free timeslot map. It can also reserve some timeslots for any Points registered with the manual registration method; the installer indicates which Point is being registered, and thus the appropriate timeslot.

You cannot reconfigure a Pyxos FT Chip from Point mode to Pilot mode, or vice-versa. To change modes, you must first deconfigure the Pyxos FT Chip and then set the CFG bit to 0.

UID and PID

The Pyxos FT Chip has two values that are used to identify devices during registration: the UID and the PID.

The unique identifier (UID) is a 48-bit value that is created during manufacture of the Pyxos FT Chip and is unique among all Pyxos FT Chips. The value is stored in control registers UID1 (PCI 0xFD) and UID2 (PCI 0xFE): the three most significant bytes of the UID are stored in UID1, and the three least significant in UID2. The fourth byte of UID1 is 0x2E, and the fourth byte of UID2 is 0x74.

The Pilot application can use the UID to uniquely identify all Points in the network. It is usually only used, however, as part of the registration process. Once the Pilot assigns a timeslot to the Point, the Pilot can reference the Point by its timeslot index.

The program identifier (PID) is a 64-bit value that is assigned by the Point host application. It is stored in two control registers: the most significant four bytes are in PID1 (PCI 0xF8) and the least significant four bytes are in PID2 (PCI 0xF9). When the Pyxos FT Chip is reset, the PID is reset to 0x0000000000000000. The host application then writes the appropriate value to the PID. Since unhosted Points do not have a processor to set their PID, the PID remains at the default value: 0x0000000000000000.

The value stored in the PID is application dependent, and is normally used to indicate a class of devices that have a similar interface (input and output variables, as well as what IO is controlled, and how the data is interpreted). The Pilot reads a Point's PID to determine how to use the Point.

Timeslot Map

The Pilot application must maintain a mapping of timeslot allocations to Points. This map will also indicate which timeslots are unclaimed, and which may be reserved for other uses, for example for hardwired Points.

The timeslot map can then be used during registration. The Pilot uses it to advertise free timeslots and to determine which timeslot to allocate to newly registered devices. If the Pilot discovers a Point that it has seen previously, the Pilot can put the Point back into its old timeslot. And, after a reset, the Pilot can short-circuit the registration process by putting the Point into its assigned timeslot directly.

If the timeslot map is kept in non-volatile storage, the Pilot application can recover the state of the system after resets and power cycles.

Pilot Configuration

The Pilot application determines how large the network is (number of timeslots) and whether frames are sent continuously or only on demand. Table 26 shows the bits that the Pilot application uses in the `CONFIG` register to configure network size and how frames are sent.

Table 26 Pilot CONFIG Register Fields

Field	Bits	Reset Value	Value	Effect/Notes
MD	31	0	0	Point mode (see below)
			1	Pilot mode
CF	30	0	0	On-demand TMD mode (frames sent only on command)
			1	Continuous TDM mode (frames sent continuously)
CFG	28	0	0	Do not send frames
			1	Send frames
TSCNT	27:24	0	TSCNT	Number of timeslots = 2 * (TSCNT + 1)
Reserved	23:0	0	0	Only write 0's to reserved fields

It is possible to change from on-demand mode to continuous mode, and vice-versa, while the Pyxos FT Chip is configured. The change will occur at the end of the next frame.

It is also possible to change the number of timeslots on the fly. However, if the Pilot reduces the number of available timeslots, any Points with a timeslot number greater than the number of available timeslots will get lost and require

re-registration. The frame time also changes when the number of timeslots changes.

Continuous and On-demand TDM Mode

When the Pilot is in continuous TDM mode, the Pyxos FT Chip continuously sends frames. Each frame is sent with only a small gap from the preceding frame, even if there is no new data to send. This mode is recommended for most networks. It ensures that data is delivered as quickly as possible.

When the Pilot is in on-demand TDM mode, the Pyxos FT Chip will only send a single frame when the Pilot application commands it to. The Pilot has no way of determining when a Point has new data for it, so it must periodically issue a frame to collect new read data from the Points. This mode is useful for applications that need to conserve power.

Only one frame will be sent on command, even if the write data is not accepted by the Point. The Pilot application can monitor the SENT flags and issue another frame if the data is not accepted.

The Pilot application commands the Pilot Pyxos FT Chip to issue a frame by writing data to a write timeslot buffer in frame memory (over the SPI bus). The frame will be sent, even if the write was simply the IDLE command (PCI 0xFF and PCV 0xFFFFFFFF). This is a useful command to send when the Pilot application has no data to send, but needs to issue a frame to determine whether a Point has new data available.

The frame will also be sent even if the SPI write is rejected due to the SENT flag still being set for that timeslot. The Pilot application must monitor the SENT bits and issue new frames as needed to ensure the data gets delivered.

In manual Pilot mode, a Point may occasionally interpret noise on an otherwise quiet line as the beginning of a packet. When this occurs, it will eventually attempt to recognize an SOF packet, and reject the noise when the SOF is found to be invalid. However, if the Pilot starts a frame while the Point is tracking the noise, the read frame may be lost. The Pilot may need to send a frame several times to ensure that all packets are delivered. The Pilot can monitor the SENT flags to determine when all packets have been delivered.

Point Configuration and Registration

A Point's configuration determines whether the device is hosted or unhosted, how to perform registration, and the timeslot identifier. Table 27 below shows the bits used in the `CONFIG` register for the Point.

Table 27 CONFIG Register Fields – Point

Field	Bits	Reset value	Value	Effect
MD	31	0	0	Point mode
			1	Pilot mode (see above)
HST[1]	30	HST	0	Hosted—if INST/MD pin low during reset
			1	Unhosted—if INST/MD pin high during reset
AD	29	0	0	Manual or hardwired registration
			1	Automatic discovery
CFG	28	0	0	Ignore timeslot packets
			1	Accept and send packets at timeslot TSID
TSCNT[1]	27:24	0	TSCNT	Number of timeslots = 2 * (TSCNT + 1). Updated each SOF packet. Not valid until first SOF.
Reserved	23:21	0	0	Only write 0's to reserved fields
PRE	20	0	0	Reject registration attempt from Pilot
			1	Accept registration attempt from Pilot
Reserved	19:5	0	0	Only write 0's to reserved fields
TSID	4:0	0	TSID	Timeslot index into WTSlots and RTSlots
Notes: 1. Read-only.				

The `TSID` field is the timeslot identifier; it tells the Pyxos FT Chip which timeslot to use for sending and receiving packets. The process of assigning a value to the `TSID` is referred to as *registration*. The PRE bit is used to prevent the Point Pyxos FT Chip from accepting registration commands from the Pilot until the Point application is ready. Once the `CFG` bit is set, the Point Pyxos FT Chip will begin to participate in the network. This should not be done until the `TSID` field is set appropriately during registration. Assigning values to these fields in the proper sequence for each registration scheme is discussed below.

The Pilot must ensure that no two Points on a single network are assigned the same timeslot. Otherwise, neither Point will be able to communicate, as they will both continuously collide with each other.

An application must never change the `TSID` field while the Pyxos FT Chip is configured. Doing so could cause the Pyxos FT Chip to interfere with a Point on another timeslot. If the timeslot needs to be changed, the Pyxos FT Chip must be de-configured first (by setting the CFG bit to 0).

A Point application must refrain from sending (i.e. writing to a PCI) until the Point has been configured. Part of the automatic and manual registration processes put the Pyxos FT Chip into a pseudo-configured state while looking for free timeslots. The Pyxos FT Chip may actually send values that it has queued up while in this state, but the Pilot will not yet recognize the sender, so the data will get lost.

For all registration schemes, the Point application must initialize its PID before allowing the Pyxos FT Chip to begin the registration. Once the Pilot recognizes a registration request from a Point, it will poll the PID. Writing the PID before starting the registration ensures that the correct PID is provided when the Pilot polls for it.

Once a Point is configured and registered, the first thing it must do is announce itself by sending its UID on its assigned timeslot. The Point application must ensure that this transaction begins before beginning any other transactions (it can monitor the SENT flag to be certain). Besides notifying the Pilot of its presence, the UID transaction is a TID synchronizing transaction; this assures that the Point and Pilot TIDs are synchronized on the read timeslot.

After verifying that the Pilot has received the UID, the Point application must write to its POINT_READY register (SPI addresses 0x27C to 0x27F), and wait for the Pilot to acknowledge receipt of the update. Table 28 shows the bits used in the POINT_READY register:

Table 28 Point POINT_READY Register Fields

Field	Bits	Reset value	Value	Effect
READY	31	0	0	Initial value. It is illegal to set this bit to 0
			1	Point is ready
Reserved	30:0	0	0	Only write zeros to reserved fields

It is the act of writing to the POINT_READY register and the receipt by the Pilot that indicates that the Point is ready, not the actual value written. The Pilot application may ignore the value of this register. The only legal value for the Point to write to the POINT_READY register is READY = 1, and reserved bits set to 0. In future releases, the reserved bits may include additional semantics. The Pilot must not update this value

After the Point application has verified that the POINT_READY register has been received by the Pilot, the Point application must wait until the Pilot application updates the SET_POINT_ONLINE register. It can do this by monitoring the POINT_RCVD flag for the POINT_ONLINE register.

If the Point is hosted, the Pilot application must wait for the POINT_READY command before setting the POINT_ONLINE register. Unhosted Points do not support the POINT_READY or SET_POINT_ONLINE registers. The Pilot application can determine whether the Point is hosted or not either by reading

the program ID (all zeros indicating that the Point is unhosted) or by reading the Point configuration register. The Pyxos Pilot API uses the program ID, since this program ID is generally needed by the Pilot application. The Pilot must not poll any values below PCI 0xF0 before it receives an update from the Point to ensure that the Point's send TID is synchronized with the Pilot's receive TID.

Table 29 shows the bits used in the SET_POINT_ONLINE register:

Table 29 POINT_READY Register Fields – Point

Field	Bits	Reset value	Value	Effect
ONLINE	31	0	0	Initial value. It is illegal to set this bit to 0
			1	Point is online.
Reserved	30:0	0	0	Only write zeros to reserved fields

The act of sending the SET_POINT_ONLINE register that indicates that the Point is online, not that the value actually sent. The Point application may ignore the value of this register. The only legal value for the Pilot application to write to this register is ONLINE = 1, and reserved bits set to 0. In future releases the reserved values may include additional semantics. Point applications must not update this value.

After the Point application has received the SET_POINT_ONLINE register, it is free to update and process incoming PCV values.

Automatic Discovery

Registration using automatic discovery is performed in three steps:

1. Free timeslot advertisements—The Pilot application advertises timeslots as available for use.
2. Registration request—An unconfigured Point Pyxos FT Chip recognizes a free timeslot advertisement, and requests a timeslot.
3. Timeslot assignment—The Pilot application recognizes the registration request and notifies the Point of its timeslot assignment.

Free Slot Advertisements

The Pilot application must keep track of which timeslots are currently allocated, and which are available to be assigned (see the *Timeslot Map* section for more information on this). The Pilot application must then decide when to advertise free timeslots. This can be done all the time, or constrained to occur only during specific periods, depending upon the application requirements.

A free timeslot advertisement consists of the UID1 and UID2 PCIs with all zeros for the data placed in the free write timeslot. That is, {0xFE, 0x00000000}, {0xFD, 0x00000000} in a write timeslot signals that the timeslot is available; both indices must appear in the same transaction.

Points do not acknowledge a free timeslot advertisement when they send a registration request. This transaction should not get acknowledged, and should continue until the Pilot cancels it. If the Pilot application finds that the

transaction has been acknowledged, it must assume that a Point has mistakenly assumed the timeslot. In this case, the Pilot application resets the Point in the timeslot and re-attempts the free timeslot advertisement.

Registration Request

If a Point Pyxos FT Chip is unconfigured and set for automatic registration (`MD` is 0; `HST` is 0; `AD` is 1; `CFG` is 0; and `PRE` is 1), then the Pyxos FT Chip monitors the timeslots looking for free timeslot advertisements. If it finds one or more, it randomly selects one of the free timeslots and attempts to send a registration request on that free timeslot. A registration request consists of UID1 and UID2 of the Pyxos FT Chip.

Once the Point Pyxos FT Chip sends the registration request, it waits for a timeslot assignment. The assignment may or may not be to the same timeslot that it used to make the registration request.

If more than one Point Pyxos FT Chips attempts to send a registration request on the same free timeslot, the Pilot will not receive a valid registration request. In this case, the registration request will not be acknowledged by the Pilot, and the Point Pyxos FT Chips will stop sending their registration request. They then back off for a random number of frames, and re-start the process.

Timeslot Assignment

Once the Pilot receives the registration request, it selects a timeslot to assign to the Point. The timeslot may be different than the timeslot that the Point used to make the registration request. Before making the actual assignment, the Pilot must cancel the free slot advertisement for the timeslot so that no other Points attempt to use the timeslot.

To make the timeslot assignment, the Pilot writes the UID of the Point into the assigned timeslot. The Pyxos FT Chip then recognizes its UID, writes the `TSID` field with the appropriate value, goes configured (sets `CFG` to 1), and then acknowledges the timeslot assignment transaction. If the Pilot does not receive an acknowledgement within a reasonable timeout period, it must assume that something has disrupted the Pyxos FT Chip. It must then attempt to reset the Point with the **PyxosPilotResetPoint()** function, and re-start the registration process.

In some cases, the Pilot application may need to assign timeslots based on the PID. In these cases, the Pilot must first temporarily assign the Point to a timeslot, and then read the PID. After reading the PID, it can move the Point to another timeslot. To move a Point to a different timeslot, the Pilot must first reset the Point, and then make the timeslot assignment to the permanent timeslot.

Manual Registration

Manual registration is used when there may be more than one device in the system with the same PID, but the Pilot application needs to be able to treat them differently. For example, the system may include two similar sensor inputs that are placed in physically different locations. The system will need to associate a physical Point with a location.

Unhosted Points all have a PID of 0x0000000000000000, so they will all appear identical to the Pilot. Therefore, an unhosted Point must use the manual registration scheme if the Pilot needs to be able to treat them differently. .

In these cases, the devices do not have enough information to identify themselves, so some user interaction with the installer is required. This type of registration can range from a simple button push at the Point in a particular order, to a sophisticated graphical interface that allows the installer to identify the device. All of these systems can be built on top of the manual registration scheme.

Manual registration is similar to automatic discovery. However, at a minimum, the registration request step requires user intervention to identify the device to be registered. User interaction can also be provided at the other steps to increase the flexibility of the registration process. The same basic steps are performed:

1. Free timeslot advertisements—The Pilot application advertises timeslots as available for use.
2. Registration request—When the installer directs the Point to begin looking for a free timeslot, the unconfigured Point Pyxos FT Chip recognizes a free timeslot advertisement, and requests a timeslot.
3. Timeslot assignment—The Pilot application recognizes the registration request and notifies the Point of its timeslot assignment.

Free Slot Advertisements

This is the same as for automatic discovery. However, the Pilot application may provide a mode that is entered through some external user interaction (e.g. pushing a button on the Pilot) that then advertise only free timeslots for a selected device. The installer can then indicate which device is being registered.

Registration Request

When using the manual registration method, the Point Pyxos FT Chip will only make a registration request when one of the two actions below is performed:

- A transition is detected on the INST/MODE pin.
- The Point application (hosted Points only) writes to the UID registers over the SPI bus.

The Point will only send one registration request when it detects one of these events, and finds a free timeslot. If the Pilot does not receive the registration request, this step will need to be repeated.

The installer must ensure that only one device is being registered at a time. If multiple devices attempt to send in a registration request at the same time, the Pilot may not be able to distinguish which device it needs to register.

Timeslot Assignment

This step is similar to what is done for automatic discovery. However, the Pilot application may again provide an installer confirmation step at this point. This confirmation may be simply the Pilot application sending a special command to the Point that causes the Point to wink an LED, or it may provide a more sophisticated confirmation mechanism.

Hardwired Registration

Hardwired registration does not rely on the Pilot or any network transactions. However, the Pilot application must reserve timeslots for hardwired Points.

This registration scheme is used when the Point application gets assigned a timeslot through some other mechanism than from the Pilot. For example, the Point's wiring harness could be keyed so that each position has a pre-assigned timeslot. In this example, the Point microcontroller reads the timeslot from the wiring harness. Another example is to manufacture each Point in a system with a different application that has a hard-coded timeslot. In this case, each Point's application contains all the information necessary for it to do timeslot assignment.

Before writing to the CONFIG register, the Point application must initialize the PID. Once the Point is configured, it will announce its presence to the Pilot, and the Pilot will poll the PID to determine what type of Point has occupied the timeslot.

Once the Point application determines the desired timeslot, it must inform the Pyxos FT Chip. The Point application writes to the CONFIG register to make the timeslot assignment. The assignment must be performed in two steps—first write the TSID, and then go configured.

In addition to writing the TSID, the first step also writes zero to the AD bit (manual register), the CFG bit (unconfigured), and the PRE bit.

The second step then sets the CFG bit to 1; the AD and PRE bits should remain at 0, and the TSID should not change.

After writing the configuration register, the Point application completes the registration process as described above, by performing the following steps:

1. Write the UID
2. Wait for acknowledgement
3. Write the POINT_READY
4. Wait for acknowledgement
5. Wait for SET_POINT_ONLINE to be sent by the Pilot

Protocol Statistics

The Pyxos FT Chip keeps track of the number of frames sent from the Pilot or received on the Point, as well as the number of CRC errors and missed slots that occur.

The frame counter is kept in the SOFCNT register (PCI 0xFB) on both the Pilot and the Point. The SOFCNT register is a 24-bit counter, using bits 23 to 0. The upper byte is reserved. The Pilot increments the SOFCNT register for each SOF packet it sends, and the Points each count the number of valid SOF packets they see. The counter rolls back to 0 after reaching 0xFFFFFF. You can write any valid value to this register and treat it as a counter that counts at the configured frame rate.

The CRC errors and missed slot counters are in the registers from PCI 0xC0 through 0xDF. There is a pair of counters (one for the CRC errors and one for the missed slot counters) for each timeslot, with timeslot 0 at PCI 0xC0 (SPI

addresses 0x300 to 0x303) and timeslot 31 at PCI 0xDF (SPI address 0x37C to 0x37F). The register layout is shown in Figure 7.11.

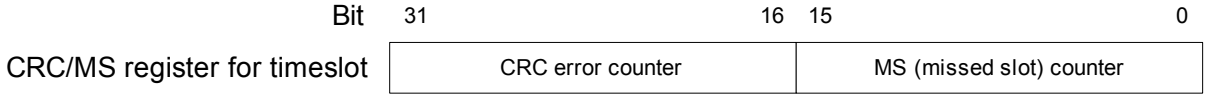

Figure 37 CRC/MS Register Layout

These counters stop counting when they reach 0xFFFF. You cannot directly write these counters. However, you can write a one to bit 29 of the ISR to clear all CRC and MS counters. The clear occurs at the beginning of the next frame after issuing the reset (at the SOF).

The Pilot accumulates error statistics for the read timeslots. The Points accumulate errors for all write timeslots, not just the timeslot for which it is configured.

The Pyxos FT Chip logs a CRC error whenever it receives a data (read or write) packet with a CRC error. There are several possible causes of CRC errors:

- Significant noise on the network.
- The network extent is beyond the specified limits.
- There is some network fault, such as a short or mis-wire.

A missed slot (MS) error is logged if the Pyxos FT Chip cannot recognize a valid preamble. This primarily occurs on read timeslots when there is no Point in a timeslot. The Pilot always drives all write timeslots, so it is rare to see a missed write timeslot. Missed slots often show up as CRC errors as well as MS errors. In addition to not having a Point to drive a read timeslot, the same conditions that result in CRC errors can cause MS errors. However, in most cases the limits need to be exceeded greatly for these faults to show up as MS errors.

Reset Handling and Error Recovery

You can design a Pilot or Point application to anticipate potential errors and provide error recovery and notification in the case of unrecoverable errors. The following errors can be detected:

- Lost network communication.
- High network communication error rates.
- Host microcontroller unable to communicate with the Pyxos FT Chip.

All error condition monitoring and recovery can be handled by the Pilot. It is in a position to monitor all of these error conditions and a Point may not have enough information to draw reasonable conclusions.

There are several potential reasons that the Pilot might lose communication with a Point. For example, the Point could get reset and lose its configuration, it could be removed from the network, or it could suffer from some application error. More is said about detecting and recovering from these types of errors below.

High network error rates may be caused by excessive electrical noise or wiring problems (for example, an intermittent short). These errors require manual intervention to clear. The Pilot can simply log the errors, and provide user notification. The tolerable error rate may vary by application, but a well-designed system should expect to see very few errors. See the *Protocol Statistics* section for more information on monitoring network error rates.

If a host microcontroller loses communication with the Pyxos FT Chip, it will no longer see network traffic. If this occurs on a Point, this will appear to the Pilot as lost communication, and will be handled by the Pilot. The Pilot itself can monitor for these problems by watching one of the periodic interrupts: EOWT, EORT, and MORT (see the *Interrupts* section below). If an extended period of time has gone by without any frames, then the Pilot application should reset its own Pyxos FT Chip. The timeout does not need to be precise as long as it is longer than a single frame.

Lost Communication Recovery

The Pilot application must monitor the health of the Points in the network to detect communication problems with the Points. The simplest and most reliable method to detect Point failures is for the Pilot to keep track of how many frames have gone by since it has successfully communicated with each Point — either receiving data from the Point, or getting an acknowledgement to a write transaction.

If the Pilot application detects that there might be a problem with a Point, it can explicitly check to see that the Point is still on line. A simple means to do this is to Poll the CONFIG register. If it gets a response, it can then verify that the CONFIG register has appropriate contents.

Once the Pilot application determines that there is a problem with a Point, it must attempt to reset the Point. This has two benefits: if multiple Points were inadvertently configured for the same timeslot, all of them will be reset. The Point's attached microcontroller will also be reset, so that the application can be re-initialized.

After the reset completes, the Pilot must ensure that the Point is properly configured. If the Point was registered using hardwired registration, then the Pilot does not need to perform any registration. However, it must wait for the Point to send its UID as an indication that it has successfully reconfigured (this also ensures that the read timeslot's TID is synchronized). For manual and automatic registrations, it only needs to send the Point's UID in its assigned timeslot. The Point will then use this transaction to configure itself into the proper timeslot.

If the Point cannot be successfully reconfigured, the Pilot application may elect to free the timeslot. This allows other devices to request the timeslot. If a similar device (with the same PID but different UID) requests the timeslot, the Pilot application may consider it to be a replacement device.

If all these attempts fail, the Pilot application can signal an error condition.

Pilot Reset or Reconfiguration

After the Pilot application resets or reconfigures itself, it must send a reset command in every timeslot for at least two frames, to ensure that all Points have been reset as well. The Pilot application will then need to re-register the Points. Re-registration may depend on the Points sending their UIDs, or, if the Pilot already knows the UID of each Point, the Pilot can start the registration process without waiting for the Points. Resetting all of the Points ensures that the TIDs are synchronized and that the Pilot receives all PCVs from the Points.

Interrupts

The Pyxos FT Chip provides an indication when certain events occur on the Pyxos FT network, including new data arrival, sent data delivered, and network timing points. Each of these events can be enabled to activate the interrupt pin (INT~) when the event occurs. The enable is controlled by the IENA (interrupt enable) register. These events are recorded in the ISR (interrupt status register) register. The available events are slightly different for the Pilot and the Point. Table 30 summarizes the events for both the Pilot and the Point, along with the bit index used for both the ISR and IENA.

Table 30 Interrupt Sources

Bit	Pilot		Point	
	Event	Description	Event	Description
31	RCVD	New data in timeslot buffer	SENT	PCV delivered
30	SENT	Timeslot buffer delivered	RCVD	New PCV available
29	Reserved		Reserved	
28	EOWT	End of write-timeslots	EOWT	End of write-timeslots
27	EORT	End of read-timeslots	EORT	End of read-timeslots
26	MORT	Middle of read-timeslots	Reserved	
0 – 25	Reserved		Reserved	

Although these events are called interrupts, and can be reflected on the Pyxos FT Chip INT~ signal, they do not need to be implemented as interrupts on the application processor. The application can periodically monitor the ISR to determine which events have occurred. Or, particular interrupts can be enabled, and the application can poll the INT~ pin before reading the ISR.

Writing a one to a bit in the ISR clears the corresponding interrupt.

Figure 38 shows the relative timing for the interrupts within a frame for both a Pilot and a Point. The timing is shown relative to events occurring on the network; these events are indicated with a dashed line, and are named with a prefixed 'N'. These network events are:

- NSOWT—Start of write timeslots on the network.
- NMOWT—Middle of write timeslots on the network.
- NSOWTS*i*—Start of write timeslot *i* on the network.
- NEOWTS*i*—End of write timeslot *i* on the network.
- NMORT—Middle of read timeslots on the network.
- NEORT—End of read timeslots on the network.
- NEORTS*j*—End of read timeslot *j* on the network.

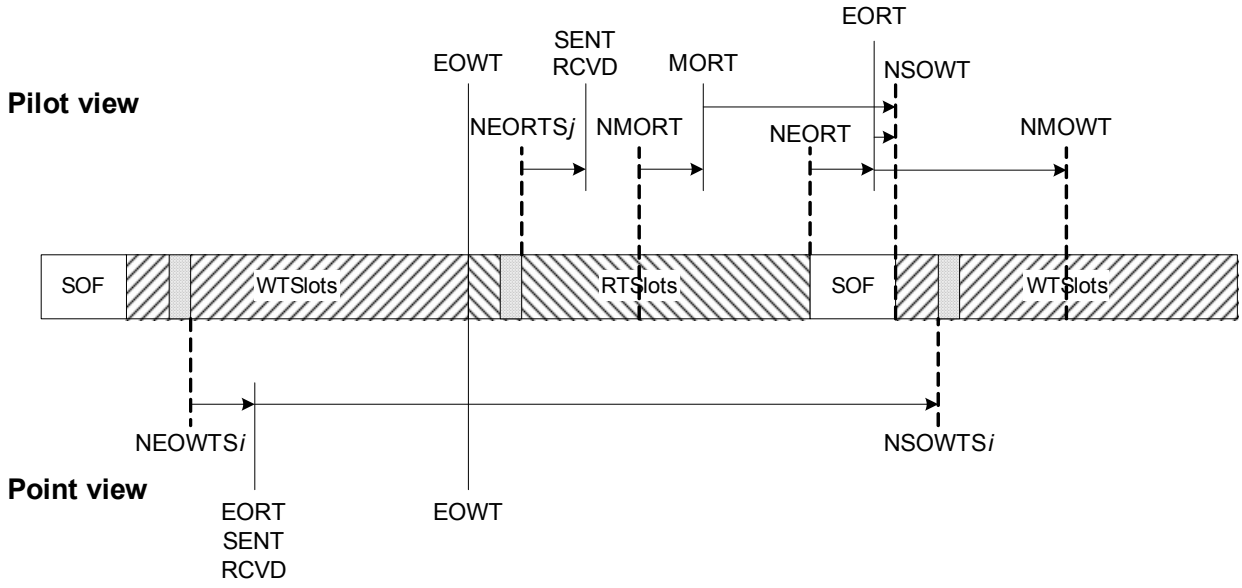

Figure 38 Interrupt Timing Diagram

Table 31 shows the latencies involved from the events occurring on the network to when the event is reflected in an interrupt.

The EOWT interrupt is synchronized with the network. From the Pilot, this is accomplished because it is in control of the timing. The Points track and predict the timing starting with the SOF. Any errors in the timing of the EOWT relative to the end of write timeslots on the network are less than 5µs.

Table 31 Interrupt Latencies

Parameter	Pilot or Point	Network Event	Interrupt	Maximum Latency	Description
t_{TS}	Pilot	NEORTS*j*	SENT RCVD	200µs	Latency for Pilot Pyxos FT Chip to process read timeslot.
t_{MORT}	Pilot	NMORT	MORT	200µs	Latency for Pilot Pyxos FT Chip to process middle read timeslot.
t_{EORT}	Pilot	NEORT	EORT	200µs	Latency for Pilot Pyxos FT Chip to process last read timeslot.
t_{Point}	Point	NEOWTS*i*	EORT SENT RCVD	200µs	Latency for Point Pyxos FT Chip to process its write timeslot.

If an application needs to minimize system latencies, for example from the time an event occurs at a Point to when a response occurs, then each of the component

steps need to be minimized. The network processing time is available (see Table 25 in the *Physical Layer* section).

The application controls the other latencies, such as the time to send a message from the Point and the time for the Pilot to respond. However, the Pyxos FT Chip interrupts can be used to aid in minimizing these latencies. The following sections discuss how to use the interrupts to minimize latencies on the Pilot or a Point. These sections assume that the application needs to handle each message as it arrives. If the application can afford to allow multiple frames to pass before servicing a message, then a more relaxed interrupt handling scheme can be used.

Handling Pilot Interrupts

A Pilot application can use any of the available interrupts to minimize processing requirements or message latencies. However, for most applications, the MORT and EORT interrupts are better suited to the application requirements. The SENT and RCVD interrupts can occur for every read timeslot, while the MORT and EORT each occur only once per frame, but at fixed points within the frame.

A Pilot application can use the SENT and RCVD interrupts to handle each timeslot separately. The SENT and RCVD interrupts are set (if necessary) for each read timeslot as the timeslot is processed. When the Pilot Pyxos FT Chip sees a read timeslot, it sets the RCVD interrupt if it received a new packet, and sets the SENT interrupt if it receives an acknowledgement for data it sent in the corresponding write timeslot.

If the SENT and RCVD interrupts are used, the Pilot application must search the SENT and RCVD flags to determine which timeslot and event caused the interrupt, and then determine how to handle the event. Then, for each timeslot, it will need to process the read data and send out new write data before the next timeslot interrupt. A read timeslot is 416µs (see Table 24 in the *Physical Layer* section), and the maximum latency for the SENT and RCVD interrupts is 200µs (see Table 31 above). So, the Pilot must process each timeslot within 216µs (including the search time).

The other approach is for the Pilot application to use the MORT and EORT interrupts to batch process the timeslots. This adds a little time to the message latencies, but significantly reduces peak processing requirements. The application can choose to use just the EORT interrupt or both the EORT and EOWT.

If the application chooses to use just the EORT interrupt, it must batch process all read timeslot and all write timeslot in the window of time between the EORT and the start of the write timeslots (about 80µs). Few applications will rely on only the EORT.

Using the MORT interrupt gives the application much more time to service the timeslots. When the MORT occurs, the first half of the read timeslots will have been received. The application then has until the start of the write timeslots before it needs to complete processing the read timeslots and creating new data for the write timeslots. Then, when the EORT occurs, the application needs to finish processing the second half of the read timeslots before the second half of the write timeslots begin to be sent.

The application must therefore service the first half of the timeslots in the window between MORT and NSOWT, and the second half of the timeslots in the window between EORT and NMOWT. Table 32 shows how much time is

available in these windows; the second window is slightly larger than the first because the read timeslots are larger than the write timeslots (see Table 25). For the MORT interrupt to NSOWT window, this is calculated as simply ½ of the write timeslots plus the SOF minus the MORT interrupt latency (see Table 31); and for the EORT to NMOWT window, the time is ½ of the read timeslots plus the SOF minus the EORT interrupt latency. The application must allow for any latency in detecting the assertion of the interrupts as well as actual processing time.

Table 32 Available Timeslot Processing Time

Timeslots	MORT to NSOWT (ms)	EORT to NMOWT (ms)
2	0.49	0.43
4	0.91	0.79
6	1.33	1.14
8	1.74	1.50
10	2.16	1.85
12	2.57	2.21
14	2.99	2.56
16	3.41	2.92
18	3.82	3.28
20	4.24	3.63
22	4.65	3.99
24	5.07	4.34
26	5.49	4.70
28	5.90	5.05
30	6.32	5.41
32	6.73	5.76

The EOWT interrupt provides a reliable timing indicator—it occurs on every frame at the same point in the frame. Applications may use this to implement transaction timeout timers, watchdog timers, or other timers that rely on the frame time.

Handling Point Interrupts

A Point can use the SENT, RCVD, or EORT interrupts to service new data as it arrives and queue new data for sending. All three of these interrupts are set at approximately the same point in the frame—after the Pyxos FT Chip has processed a write timeslot that the Point is assigned to.

The SENT interrupt will occur only if the transmit buffer was empty (because the previous read transaction was acknowledged) and the scheduler copies a new PCV to the transmit buffer, clearing a SENT flag. The RCVD interrupt will occur if a new PCV arrives in the timeslot. The EORT interrupt will occur on every frame whether or not a new PCV has been received or a new outgoing transaction has been started.

The Point application can use the SENT and RCVD interrupts to only get notified when new data arrives or an outgoing PCV has been freed up. When the interrupt occurs, the application must search through the SENT and RCVD flags to determine which one caused the interrupt.

If the Point application is using only a small number of PCIs, then it can use just the EORT interrupt instead and check the flags for all of the PCIs that it is using. This avoids the step of reading the ISR to determine the cause of the interrupt.

The EOWT interrupt can be used in the same way as on the Pilot: it provides a reliable timing indicator with a fixed relationship to the data on the network. This can be used to implement various timers.

Managing Unhosted Points

A Pyxos FT Chip can be used without a host microcontroller to implement a Point with unhosted digital I/O requirements. The Pilot must manage the I/O on the unhosted Point—including configuration, reading, and writing.

The COUT pin (clock output) is available on both hosted and unhosted Points. Configuration of this pin for both types of Points is covered in this section as well.

DIO

Some of the Pyxos FT Chip SPI pins are converted to general purpose digital IOs when used in unhosted mode, as shown in Table 33.

Table 33 Pyxos FT Chip DIO Pins

Name	Pin	Direction
MOSI/DIO0	4	Configurable
MISO/DIO1	5	Configurable
CS~/DIO2	1	Configurable
INT~/DIO3	2	Configurable
SCLK~/DI	3	Input

The direction of the first four **DIO** pins is configured over the network by the Pilot. Each of these pins can be configured independently. They can be configured to be either an output (with the value set by the Pilot) or an input. As an input, the value can be polled by the Pilot, or sent automatically when it changes. Tables 34 through 37 provide details on how to configure the **DIO** from the Pilot. DI, always acts as an input.

Table 34 DIO Index

Index Address	Byte 0	Byte 1	Byte 2	Byte 3
0xEF	Bits [31:24]	Bits [23:16]	Bits [15:8]	Bits [7:0]
IO Registers	Data	Direction	Config	Reserved

The DIO Data Register is used to read and write to the **DIO** pins. If a pin is configured as an output, then writing a value to the appropriate bit of this register will change the state of the pin to match. Reading from this register will return the current state of the pin. **DI** is always an input.

Table 35 DIO Data Register

Index Bits	31:29	28	27	26	25	24
Pin	Reserved	SCLK/DI	INT~/DIO3	CS~/DIO2	MISO/DIO1	MOSI/DIO0
Default	Reserved	0	0	0	0	0

The DIO Direction Register is used to configure the **DIO** pins as inputs or outputs. Writing a 0 to a bit in this register sets the corresponding pin to be an input; a 1 sets the pin to be an output. The **DIO** pins are initially inputs.

This register also controls the direction of the COUT pin (which is available for both unhosted and hosted Points). This output, when enabled provides a

buffered clock output that matches the clock input to the Pyxos FT Chip (10MHz). The COUT pin is initially enabled as an output.

Table 36 DIO Direction Register

Index Bits	23:21	20	19	18	17	16
Pin	Reserved	COUT	INT~/DIO3	CS~/DIO2	MISO/DIO1	MOSI/DIO0
Default	Reserved	Output (1)	Input (0)	Input (0)	Input (0)	Input (0)

The DIO Config Register provides configuration control of the **DIO** pins. There is only one field: AUTOUPDATE (bit 8). If AUTOUPDATE is enabled (1), then the **DIO** pin values are sampled on each frame, and if any of the inputs have changed, the updated value is sent to the Pilot. Since the inputs are sampled once per frame, changes occurring more frequently than this will be lost. In addition, inputs must not change at a rate faster than 5MHz; there must be more than 200ns between changes to input pins.

Table 37 DIO Config Register

Index Bits	15:9	8
Field	Reserved	AUTOUPDATE
Default	Reserved	Disabled (0)

Appendix A

Advanced Topics

This chapter discusses advanced topics, including descriptions of the functions you can use to read and write registers that control the operation of a Pyxos FT Chip, and a discussion on the data space and code size consumed by a Pyxos application.

Accessing Pyxos Registers

The Pyxos FT Chip contains several registers that control the operation of the chip. The Pyxos FT API uses these registers to perform work on behalf of the application, such as configuring the chip, determining when values have arrived and when new values can be sent, and reading and clearing statistics. Normally, the Pyxos application will not access the Pyxos registers directly. However, access to the registers is provided in case there is some function that the application must perform that is not directly supported by the API. Modifying these registers directly can cause unanticipated side affects as described in this appendix.

The Pyxos FT API implements two functions, **PyxosReadRegister()** and **PyxosUpdateRegister()**, that are used to read and update a register. The API also defines constants to be used when accessing the registers. The following sections describe the register functions and definitions in detail. For details on what each register does, see Chapter 7, *Pyxos FT Protocol*.

PyxosReadRegister

Reads a Pyxos register.

Syntax:

```
PyxosSts PyxosReadRegister(PyxosPci registerIndex,
                           Byte *pData,
                           PyxosPnvSize size);
```

Remarks: This function reads the contents of the register indicated by the *registerIndex* parameter, and stores the contents in the buffer pointed to by the *pData* parameter. The size of the register, in bytes, is given by the *size* parameter.

PyxosUpdateRegister

Updates a Pyxos Register.

Syntax:

```
PyxosSts PyxosUpdateRegister(PyxosPci registerIndex,
                             const Byte *pData,
                             PyxosPnvSize size);
```

Remarks: This function updates the contents of the register indicated by the parameter *registerIndex* with the contents in the buffer Pointed to by the *pData* parameter. The size of the register, in bytes, is given by the *size* parameter.

Register Definitions

The Pyxos FT API defines several constants that you can use when accessing the Pyxos registers. These can be found in the **PyxosRegisters.h** header file in the [*Pyxos FT EVK*]**Pyxos FT API\include** directory. Table A.1 lists the constants the file contains for each register.

Table 38 Register Constants

Name	Description
`PYXOS_REGI_<registerName>`	Represents the Pyxos Chip index of the register. You can use this value as the *registerIndex* parameter when calling **PyxosReadRegister()** or **PyxosUpdateRegister()**.
`PYXOS_REGS_<registerName>`	Represents the size, in bytes, of the register. You can use this value as the *size* parameter when calling **PyxosReadRegister()** or **PyxosUpdateRegister()**. The Pyxos FT Chip defines each register as 4 bytes in length. However, the Pyxos FT API combines some of these registers into a single logical register. For example, the Pyxos FT Chip defines two 4-byte registers to hold the unique ID. The API treats this as a single 8-byte register.

`PYXOS_REGR_<registerName>`	Defines a subset of the registers. Used internally by the Pyxos API to add the register to a Point's interface. This allows the Pilot to access the register on a Point just as if it were a PNV. These are included in a Point's interface via the **PYXOS_STANDARD_REGISTERS** macro defined in the **PyxosPilot.h** file. The definition of this macro is given below:
	
	`#define PYXOS_STANDARD_REGISTERS \`
	`PYXOS_REGR_UID \`
	`PYXOS_REGR_RESET_CONTROL \`
	`PYXOS_REGR_CONFIG \`
	`PYXOS_REGR_SET_POINT_ONLINE \`
	`PYXOS_REGR_PID`
	
	The Pilot can also access remote registers on a Point, provided that they are part of the Point's program interface definition. To add registers to the Point interface definition, you can modify the definition of the **PYXOS_STANDARD_REGISTERS** to include additional remote register definitions.

The **Register.h** file also includes several constants and macros that you can use to interpret the contents of the registers, often in the form of bit bitmasks. For more information about each of the registers supported by the Pyxos FT Chip, see Chapter 7, *Pyxos FT Protocol*. For details concerning the definitions supplied by the Pyxos API for each register, see the **Register.h** header file.

Code and Data Space Considerations

The code and data space taken by the Pyxos APIs depend on a number of variables:

1. The processor architecture. This affects the size of the code, as well as the size and alignment of the data.
2. Compiler optimizations.
3. Which API features have been included in the application.
4. The number, size, type, and direction of the PNVs
5. The number of Points supported by the Pilot.
6. Hardware support for SPI, which affects the size of the PsAPI.

The following sections provide some guidelines that you can use to estimate the amount of memory required by the Pyxos API.

Pyxos Serial API Footprint

The size of the Pyxos serial driver implementation is hardware-specific. For reference, the implementation shipped with the Pyxos EVK for the AT91SAM7S64 uses about 500 bytes of code space and 64 bytes of data.

Pilot API Footprint

The Pilot API code size compiled for the AT91SAM7S64 is between 5 and 6.5 Kbytes, depending on the particular features included. This estimate does not include the serial API driver or system overhead.

Constant Data

The size of the constant data used by the Pilot API depends on the number of PNVs supported by each type of interface on the Pyxos FT network, and on the size of the structures used to hold this information. Since the size of the structure may depend on the processor used, these numbers may differ on different processors.

You can calculate the constant data consumed by each interface supported by the Pilot with the following formula:

```
sizeof(PyxosPilotPointInterface) +
(Number of PNVs + 5) * sizeof(PyxosPilotPnv)
```

Table 39 lists the size of these data structure on the AT91SAM7S64.

Table 39 Data Structure Sizes for AT91SAM7S64

Structure	Size in Byte on AT91SAM7S64
PyxosPilotPointInterface	40
PyxosPilotPnv	4

Dynamic Data

The Pilot pre-allocates some dynamic data based on the maximum number of Points supported by the Pilot. The size of this memory is calculated as follows:

```
PYXOS_MAX_POINTS *
(sizeof(PyxosPilotPointCache) + 5*sizeof(PyxosPnvCache) + 37)
```

You can calculate the amount of dynamic memory used for a single Point with a given interface as shown below. If you use the heap, this memory is allocated when the interface is set. If you do not use the heap, the API pre-allocates this memory for the maximum number of Points that may be using this interface:

```
(5 + Number of PNVs in this interface) * sizeof(PyxosPnvCache) +
19 +
Maximum size of an input PNV in this  interface +
Total number of bytes used by all PNVs in this interface.
```

Table 40 lists the size of these data structures on the AT91SAM7S64.

Table 40 Data Structure Sizes for AT91SAM7S64

Structure	Size in Byte on AT91SAM7S64
PyxosPilotPointCache	767
PyxosPnvCache	2

These estimates are based on the current definition of PYXOS_STANDARD_REGISTERS.

Point API Footprint

Table 41 lists the approximate code size of the Point API, based on the features included. These estimates are based on an implementation using the AT91SAM7S64. These numbers do not include the serial API driver or system overhead.

Table 41 Point API Code Size

Input PNVs	Output PNVs	Polling	Size (Kbytes)
Yes	No	No	0.8
No	Yes	No	1.2

Input PNVs	Output PNVs	Polling	Size (Kbytes)
Yes	Yes	No	1.3
Yes	Yes	Yes	1.6

Constant Data

The Point API consumes sizeof(PyxosPointPnv) constant data for every PNV supported by the Point. This structure is typically 3 bytes long.

Dynamic Data

The fixed amount of RAM used by the Point API is small, but additional RAM is used based on the Pyxos Points' interface as follows:

1/8 * *total number of PNVs* (rounded up)
+
1/8 * *number of Pyxos Chip indices used by inputs* (rounded up).
+
1/8 * *number of Pyxos Chip indices used by outputs* (rounded up)
+
4 * *number of Pyxos Chip indices used by outputs*

Interrupt Driven Pyxos Programs

The simplest way to use the Pyxos Point and Pilot APIs is to call the event handlers (**PyxosPilotEventHandler()** or **PyxosPointEventHandler()**) periodically in a control loop. Calling the event handler when there are no tasks to perform is generally very fast. The first thing the event handler does is read the Pyxos Chip interrupt line. If the interrupt is not set, it returns. If you are concerned that this read operation is too expensive to perform during each iteration of your control loop, you can use the Pyxos Chip interrupt line to cause a processor interrupt. Set a global variable in the interrupt. In your main control loop, check this variable and call the Pyxos event handler only when the flag has been set.

If your Point or Pilot is typically very busy and cannot run a control loop frequently enough, you may find that calling the event handler from an interrupt routine driven by the Pyxos Chip interrupt pin provides better network throughput. Following are critical design considerations if you call the event handler or any Pyxos API function from an interrupt service routine:

1. The Pyxos API does not protect any of its global data from being accessed simultaneously by an interrupt service routine and by your main function. If you ever call any Pyxos API function from an interrupt service routine, you must ensure that the interrupt does not do so while your main application (or another interrupt) is calling into the Pyxos API. The simplest approach is to disable the Pyxos interrupt handler whenever your application calls any of the Pyxos API functions, or accesses any of the Pyxos API data structures.

2. The Pyxos event handler calls **psRead()** and **psWrite()**. You must make sure that your implementation of the Pyxos serial API supports calling these functions from within an interrupt service routine.

3. All API callback functions are called directly from the Pyxos event handler. If the event handler is called from an interrupt handler, then your callbacks are also being called from an interrupt handler. This may limit the sorts of thing you can do from within your callback handlers.

Defining Point Interfaces Dynamically

Typically, all of the Point interfaces that a Pilot will use are pre-defined when the Pilot is created. However, you can create a Pilot application that constructs Pyxos Point interfaces on the fly, and still use the Pyxos Pilot API. To do so, the Pyxos Pilot application must construct the **PyxosPilotPointInterface** structure and all of its sub-structures, rather than relying on the Pyxos Interface Developer to do so.

This section describes the data structures required by the Pyxos Pilot API to describe a Point's interface, so that you can program your Pilot application to construct those data structures dynamically.

The Pyxos Pilot API supports hosts that use the heap and hosts that do not. When you use the heap, the API allocates dynamic data structures on demand when the application calls the **PyxosPilotSetPointInterface()** function. When you do not use a heap, the Pyxos Interface Developer creates compile time data structures to hold the PNV cache and control data. The interface structures used by the Pilot API differ depending on whether the heap is used or not. The heap method should be used for Pilots that dynamically create Point interfaces since it is better suited to dynamic data structures. This section describes how to create dynamic interfaces using the heap method

Each Point interface is defined by two data structures: **PyxosPilotPointInterface** and an array of **PyxosPilotPnv** values. These data structures are both defined in the **PyxosPilot.h** file and are described below. Fields that are not used with the heap allocation method have been omitted.

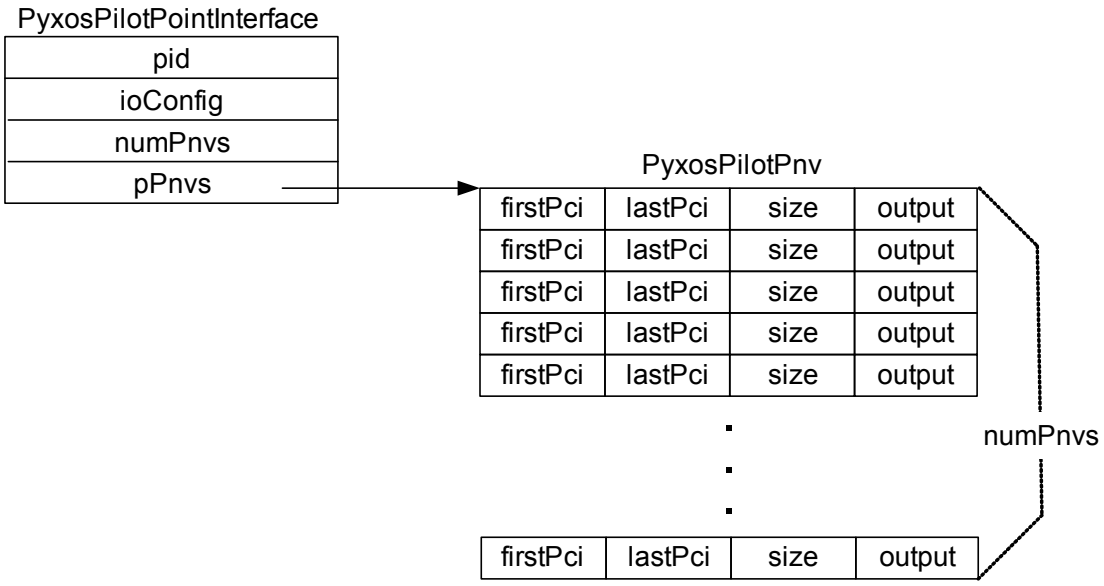

Figure 39 Point Interface Data Structures

The **PyxosPilotPointInterface** data structure is passed to **PyxosPilotSetPointInterface()** and has the following definition:

```
typedef struct PyxosPilotPointInterface
{
        /* The Point's program ID. */
    PyxosProgramId        pid;
    Dword                 ioConfig;
        /* Number of Pyxos Network Variables and Remote
           registers
         */
    PyxosPci              numPnvs;

        /* Pointer to array of Pyxos Network Variables and
           Remote registers.
         */
    const PyxosPilotPnv  *pPnvs;
} PyxosPilotPointInterface;
```

The fields are described below:

pid **PyxosProgramId**

An array of 8 bytes specifying the Point's program ID. This field is typically set to PYXOS_<pointName>_PROGRAM_ID defined in the Point interface header file. If the Point is unhosted, this field should consist of all zeros.

ioConfig **Dword**

For hosted Points, this must be set to **PYXOS_HOSTED_IO_CONFIG** (0xFFFFFFFF). For unhosted Points, set this to the desired initial value of the I/O register. The I/O register contains the following bit masks (defined in PyxosRegisters.h), all of which are ORed together.

1. The direction of each of the four general-purpose digital I/O values. Each **DIO***x* general-purpose **DIO** pin must be either

PYXOS_IO_OUTPUT_DIO_<*x*> or **PYXOS_IO_INPUT_DIO_**<*x*> (see example below).

2. The configuration of the clock out pin. Either **PYXOS_IO_COUT_ENABLED** or **PYXOS_IO_COUT_DISABLED**.

3. Whether the Point sends data on change or not. Either **PYXOS_IO_MODE_POLL** or **PYXOS_IO_MODE_SEND_ON_CHANGE**.

4. The current I/O values. Leave these values set to zero.

EXAMPLE:

An unhosted device has two inputs, **DIO1** and **DIO2** two outputs, **DIO3** and **DIO4**, does not require the clock out pin, and sends updates on change. The value of the **ioConfig** field for this device is:

```
(PYXOS_IO_INPUT_DIO_1 |
 PYXOS_IO_INPUT_DIO_2 |
 PYXOS_IO_OUTPUT_DIO_3 |
 PYXOS_IO_OUTPUT_DIO_4 |
 PXYOS_IO_COUT_DISABLED |
 PYXOS_IO_MODE_SEND_ON_CHANGE)
```

numPnvs **PyxosPci**

The number of PNVs defined on the Point plus the number of registers on the Point that the Pilot needs to access. This is equal to the number of entries in the **PyxosPilotPnv** array described later in this section.

pPnvs **const PyxosPilotPnv *

A pointer to an array of all PNVs defined on the Point, as well as all registers on the Point that the Pilot has to access. This array is described in more detail below.

The **PyxosPilotPnv** array is accessed via the pPnvs field of the **PyxosPilotPointInterface** data structure. This array contains one element for each PNV supported by the Point, plus one element for each register on the Point that the Pilot has to access. The Pilot API requires access to the following registers on each Point: **PYXOS_REGI_UID, PYXOS_REGI_RESET_CONTROL and PYXOS_REGI_CONFIG.** In addition, the Pilot must access the **PYXOS_REGI_UNHOSTED_IO** register for all unhosted Points. The order of the entries in the array is unimportant.

Each element of this array uses the following structure:

```
typedef struct _PyxosPilotPnv
{
   PyxosPci firstPci;   /* The lowest Pyxos Chip index used
                           by the PNV */
   PyxosPci lastPci;    /* The largest Pyxos Chip index
                           used by the PNV */
   Byte     size;       /* The size of the PNV in bytes.
                           Not necessarily divisible by 4.
                        */
```

```
    BitField output : 1; /* 1 if this is a
                            PYXOS_POINT_OUTPUT, 0 if it is a
                            PYXOS_POINT_INPUT.
                         */
    BitField reserved : 7;
} PyxosPilotPnv;
```

firstPci **PyxosPci**

The lowest Pyxos Chip index (PCI) of the Pyxos network variable or register.

lastPci **PyxosPci**

The highest Pyxos Chip index (PCI) of the Pyxos network variable or register.

size **Byte**

The size of the Pyxos network variable or register in bytes.

output **BitField:1**

A one if the Pyxos network variable or register is an output, a zero if it is an input.

The **PYXOS_STANDARD_REGISTERS** macro defines the initialization values for all of the standard registers. The **PYXOS_REGS_UNHOSTED_IO** macro evaluates to the initialization values for the unhosted I/O register. The **PYXOS_<pointName>_NETWORK_VARIABLES** macro defined in the Point interface evaluates to the initialization values for the PNVs defined by the Point. This macro is not typically available to Pilots that define Point interfaces dynamically (otherwise it could have defined the Point interface statically). However this macro can be used as a reference during development to ensure that the data structures created by the Pilot are correct.

Supporting Multiple Pyxos Networks

The Pyxos Pilot API supports only a single Pyxos network at any given time. However, a single host connected to multiple Pyxos chips can act as the Pilot for multiple networks. To do so, you must modify the Pyxos Pilot API to support multiple networks. This section describes, in very general terms, an approach that can be used to modify the Pyxos Pilot API in order to support multiple networks.

Nearly every function and callback defined by the Pyxos Pilot API needs to be changed to include a network identifier. The easiest approach is to define the network identifier as an index from 0 to *n*-1, where *n* is the maximum number of Pyxos networks supported by a single Pilot. The Pilot application must specify this interface ID in all calls to the Pyxos API. The Pyxos API in turn must specify this ID to the Pyxos Serial API (psAPI) and use that information to identify which Pyxos Chip to communicate with.

The Pyxos Pilot API also uses a number of global variables to maintain status information. Except for a few constant variables, and a few variables that are used only temporarily, these must be replicated for each Pyxos network supported. The easiest way to change the Pilot API to support multiple instances of these global variables is replace all of the global variables with a structure

definition with a field corresponding to each variable, and to define an array of these structures, one element per network.

For example, you can use the following definitions to maintain network specific information:

```
typedef Byte NetworkId;

typedef struct _PyxosNetworkData
{
        /* The Point directory, indexed by timeslot.
           Allocated by PyxosPilotAllocateTimeslot().
           Contains Point control information, pointer to
           the Point data cache, the Point's unique ID and
           pointer to the Point's interface record.
         */
    PyxosPilotPointCache
       pyxosPointDirectory[PYXOS_MAX_POINTS];

        /* A bitmask, indexed by timeslot.  A set bit
           indicates that the Pilot should send a new value
           even if the last value was never acknowledged.
         */
    Dword pyxosClearUnackedWrite = 0;

        /* Flag indicating that there is a new PNV update
           or poll request to send. This is used run the
           event handler even if there is no interrupt,
           since there may be no interrupt if there are no
           updates or acknowledgments coming in.
         */
    Bool pyxosNewPnvUpdateToSend = FALSE;



        /* The pyxosPilotWriteFrameCount is set to 0 by
           PyxosPilotInit() and is incremented by
           PyxosPilotEventHandler() each time it processes a
           write frame.  This value may be less than the
           actual number of frames if
           PyxosPilotEventHandler() is not called frequently
           enough.  This value wraps at 0xFFFFFFFF.  This
           value can be used to perform frame relative
           timing.
         */
    Dword pyxosPilotWriteFrameCount;
        /* The following variables record the frame count
           last time MORT and EORT respectively were
           processed.
         */
    static Dword lastFrameToProcessMort;
    static Dword lastFrameToProcessEort;

        /* The Pilot resets all timeslots on startup to
           ensure that the Point's send transaction IDs are
           synchronized with the Pilot's, and to force them
           to resend any updates.  This variable is used to
```

```
            determine how many frames of resets have been
            sent thus far.
           */
        Byte numResetsLeftToSend;
} PyxosNetworkData;

#define MAX_NETWORKS 5
PyxosNetworkData pyxosNetworkData[MAX_NETWORKS];
```

A new **NetworkId** parameter can be added as the first parameter of nearly every Pyxos API function. The example implementation of **PyxosPilotInit()** shown below illustrates how each function can be modified. Code that has been added is in bold.

```
/* Initialize the Pyxos Pilot cache, and configure the
   Pyxos chip.
 */
PyxosSts PyxosPilotInit(NetworkId netId)
{
    PyxosTimeslot timeslot;
    PyxosNetworkData *pNet = &pyxosNetworkData[netId];

    memset(pNet->pyxosPointDirectory, 0,
           sizeof(pNet->pyxosPointDirectory));

    pNet->pyxosPilotWriteFrameCount = 0;
    pNet->lastFrameToProcessMort= 0;
    pNet->lastFrameToProcessEort= 0;

    /* Initialize each timeslot to use the
       pyxosTemporaryPointInterface.
     */
    for (timeslot = 0; timeslot < PYXOS_MAX_POINTS;
         timeslot++)
    {
        PyxosPilotSetPointInterface(netId, timeslot, NULL);
    }

    return PyxosPilotInitPyxosInterface(netId);
}
```

The Pyxos Serial API must also be modified to accept a NetworkId, and select the correct Pyxos FT Chip accordingly. This can be used to drive the Pyxos FT Chip chip select pin (CS~).

Supporting Multiple Pyxos Networks without a Heap

You can modify the Pyxos API to support multiple networks as described in the previous section. If you are not using the heap, additional changes are required as described in this section. If this is the case, the Pyxos Interface Developer creates byte arrays for each type of Point supported by the Pilot that are sized based on the Point's interface and the maximum number of instances of that Point within the network. An extra byte for each instance is reserved to indicate which timeslot the chunk of data is being used for (or 0 if none). At runtime, data is allocated from these arrays, with the first byte of each "chunk" set to *timeslot* +

1. This scheme requires modification to support multiple networks. One fairly straightforward way to change this is to reserve an extra byte for each "chunk" to store the network ID. This requires changing the **PYXOS_DEFINE_POINT_INTERFACE_RECORD** and **PYXOS_DEFINE_UNHOSTED_POINT_INTERFACE** macros (defined in the **PyxosPilot.h** file) to include room for the network ID. For example, refer to the following portion of the **PYXOS_DEFINE_POINT_INTERFACE_RECORD** macro:

```
Byte name##DataCache[(maximumPoints)*                          \
                     (sizeof(PyxosTimeslot) +                  \
                      PYXOS_MAX_INPUT_SIZE(maxInputSize) +     \
                      (pnvBytesPerPoint) +                     \
                      (remoteRegisterSize))] = { 0 };          \
```

Change this portion to:

```
Byte name##DataCache[(maximumPoints)*                          \
                     (sizeof(PyxosTimeslot) +                  \
                      sizeof(NetworkId) +                      \
                      PYXOS_MAX_INPUT_SIZE(maxInputSize) +     \
                      (pnvBytesPerPoint) +                     \
                      (remoteRegisterSize))] = { 0 };          \
```

Similar changes must be made to update the **dataCacheEntrySize** field of the **PyxosPilotPointInterface** structure.

To allocate the cache data properly you must modify **PyxosPilotAllocateCacheData()** to set the network ID as well as the timeslot, and you must modify **PyxosPilotFreeTimeslotData()** to use both the network ID and timeslot to identify the cache data to be freed.

For more details, refer to the macros and functions mentioned above.

Designing Deterministic Systems

The Pyxos Platform is inherently deterministic at the physical communication layer. This is because communication occurs at a well defined, predictable rate, based on the number of timeslots. Because communication with each Point occurs only in its assigned timeslot, there is no possibility of contention. As a result, both the communication rate and latency at the physical level is deterministic.

You can take advantage of the deterministic nature of the Pyxos Platform to create deterministic systems. However, determinism at the physical layer by itself is not sufficient. In order to design a deterministic system, there are a number of factors that you must consider. This section briefly describes some of these factors. However, there may be other factors specific to your application that will affect the determinism of the system as a whole.

Physical Network Reliability

The Pyxos network ensures reliable data delivery. Data sent on the network includes a CRC to protect against data corruption, and all data is acknowledged. If a packet was corrupted, due to noise for example, the data will automatically be retransmitted in the next frame. However, since noise may require data to be

retransmitted it is not possible to predict the transmission rates or latency in a noisy system.

To ensure deterministic network communication at the physical layer, transients should be kept out of the network cable and power supply cable, in order to avoid having single frames corrupted. See the discussion of EMC in the *Pyxos FT Chip Data Book* for more information on this.

Host Responsiveness

The Pyxos platform does not require that either the Pilot or Point applications process data at the Pyxos frame rate. If a Point fails to read a value and the Pilot tries to send a new value to the same index, the Pilot will automatically retry until the Point has successfully read the previous value. Likewise, if the Point sends a value to the Pilot and the Pilot has not read the value, the Point will retransmit the value.

If the Point or Pilot application does not process data as quickly as the network delivers the data, the application processing becomes the data transfer bottleneck. In order to design a deterministic system, you must ensure that both the Pilot and Point applications process data as quickly as the network provides the data. If not, you must ensure that the frequency and latency that they will process the data with is deterministic.

The Pyxos Point and Pyxos Pilot APIs both use the Pyxos Chip interrupt line to tell when data is available. However, when using either API, it is up to the application to call the event handler either frequently enough to keep up with the network or at some slower, but deterministic, frequency. In order to keep up with the network the Point application should call the event handler at least as fast as the frame rate. In order to keep up with the network the Pilot application should call the event handler approximately four times as fast as the frame rate. In either case, calling the event handler more quickly is perfectly acceptable, and has very little overhead. Note that you may also use the Pyxos Chip interrupt line as a processor interrupt, and call the event handler from an interrupt service routine.

For more information about network timing see *Physical Layer* on page 125. For more information regarding use of interrupts when using the API, see *Interrupt Driven Pyxos Programs* on page 166. For more information about Pyxos Chip Interrupts, see *Interrupts* on page 153.

Application Data Rates

The system can only be considered deterministic if the application produces data deterministically. For example, in order for the Pilot to receive a sensor update from a hosted Point with a deterministic frequency, the application on the Point must read the value and update the corresponding Pyxos Network Variable at a deterministic rate. If this rate exceeds the effective network rate, which is gated either by the physical network rate or the processing latency on the host, some of these values must be lost.

The Pyxos Point and Pyxos Pilot APIs allow the application to update Pyxos Network Variables as quickly as they like. The API ensures the delivery of the

latest value of each Pyxos Network Variable. However, if the application produces updates for a given PNV faster than the network can deliver it and any other updates, intermediate values will be lost.

A Pyxos Point application may use the **PyxosPointIsPnvUpdatePending()** function to determine whether or not a value has been set which has not yet been acknowledged by the Pilot. Similarly, a Pilot application can use the **PyxosPilotIsPnvUpdatePending()** function to determine whether or not a value has been set which has not been written to the Point. These functions can be used by the application to prevent overwriting intermediate values. However, the application cannot exceed the effectively network transmission rate, which is based either on the physical transmission rate or the rate at which the hosts process their data.

Polling

It is possible for both the Pilot and the Point to send poll requests. When designing a deterministic system you must consider the impact polling has on network communications. The Poll request itself consumes network bandwidth, and the values sent as a result of the poll consume network bandwidth, assuming that those values would not have been sent anyway.

For more information about polling, see *Polling* on page 140.

Appendix B

Pyxos FT Network Gateways

This appendix provides background information you will need when extending your Pyxos FT network to connect to other Pyxos FT networks, or to a LONWORKS network.

Pyxos FT Network Gateways

You can use a Pyxos FT network as a fully-contained standalone network managed by the Pyxos Pilot. The Pilot is responsible for managing the Pyxos Points on the network, and any communication between the Points. However, you can also extend a Pyxos FT network to:

1. Connect multiple Pyxos FT networks together, sharing data from all the remote sensors and actuators with all the Pilots.

2. Connect to other networks, sharing data from the remote sensors and actuators with other devices on other networks or in other systems.

To provide access to the Pyxos FT network, you can implement a Pyxos FT network gateway in your Pyxos Pilot application. You can use this gateway to connect to other Pilots with their own Pyxos FT networks, LONWORKS networks, or other third-party networks or systems.

Due to the high performance of a Pyxos FT network, a gateway application will typically expose aggregate data or alarm conditions, or will provide high-level functions, rather than providing direct access to individual Points. In this way, the Pyxos FT network can operate autonomously, while providing the ability to monitor and control functions of the Pyxos FT network as a whole.

Since a gateway is implemented as part of a Pilot application, the types of external networks and which functions are provided are completely up to the gateway developer.

The next section describes, in general terms, how you can develop a Pyxos – LONWORKS gateway.

Pyxos – LONWORKS Gateways

To implement a Pyxos – LONWORKS gateway, the Pilot must implement a LONWORKS application device, as well as perform the standard functions of a Pyxos Pilot. You can use an Echelon Smart Transceiver with Echelon's ShortStack® Micro Server and ShortStack API to add LONWORKS connectivity to a Pyxos Pilot. For more complex Pilots you can use a Smart Transceiver with Echelon's Microprocessor Interface Program (MIP) firmware and host API. See the ShortStack Web page at www.echelon.com/shortstack for more information on the ShortStack Developer's Kit.

Device Interface

The device interface for a LONWORKS device consists of its functional blocks, network variables, and configuration properties. A *LONWORKS network variable* is similar to a Pyxos network variable—it is a data item that can be sent from one LONWORKS device to one or more other LONWORKS devices. You will expose selected Pyxos data points and conditions as LONWORKS network variables. Data points may represent PNVs within the Pyxos network, or they may be values calculated by the Pilot based on PNV values. LONWORKS network variables representing PNV values may use the same network variable types as the PNVs they represent. LONWORKS network

variables representing calculated values or other conditions may use different network variable types.

A LONWORKS configuration property is a data item that, like a network variable, is part of the device interface for a device. Configuration properties are used to configure the behavior of a network variable, functional block, or device. For example, you can use a configuration property to indicate a heartbeat rate to ensure that a network variable gets updated periodically, a second configuration property to throttle updates to limit bandwidth consumption, and a third configuration property to indicate a minimum delta required to send an update. You can also use configuration properties to define alarm thresholds for a network variable so that an alarm condition is reported when the value goes out of range.

A *functional block* is a collection of network variables and configuration properties that are used together to perform one task. These network variables and configuration properties are called the *functional block members*.

Functional blocks are defined by *functional profiles*. A functional profile is used to describe common units of functional behavior. Each functional profile defines mandatory and optional network variables and mandatory and optional configuration properties. Each functional block implements an instance of a functional profile. A functional block must implement all the mandatory network variables and configuration properties defined by the functional profile, and can implement any of the optional network variables and configuration properties defined by the functional profile. A functional block may also implement network variables and configuration properties not defined by the functional profile—these are called *implementation-specific* network variables and configuration properties.

A Pyxos – LonWorks gateway must map the PNVs on the Pyxos network to appropriate LONWORKS functional blocks, network variables, and configuration properties.

EXAMPLE:

A Pyxos FT network for an HVAC controller has a temperature sensor Point, a humidity sensor Point, and a valve actuator Point. The LONWORKS interfaces for the Points consist of an **SFPTnodeObject**, **SFPThvacTempSensor**, a **SFPThvacRelativeHumiditySensor**, and a **SFPThvacValvePositioner** functional profile. The two sensor profiles each define a standard LONWORKS network variable output to report the sensor value, and the actuator profile defines two standard network variable inputs to control the valve position, as well as standard network variable outputs to report the current valve state. The Node Object functional block is used to report alarms with a standard **SNVT_alarm_2** alarm output, and is also used to control the other functional blocks with a standard request input and response output. All four profiles define standard configuration properties that are used to control when data is reported, to control when alarm conditions are reported, and to optionally configure the sensors.

Connections

When a Pyxos Pilot includes a Pyxos – LONWORKS gateway, you can connect the LONWORKS network variables implemented by the gateway to other LONWORKS network variables on the LONWORKS network. You can use a connection to establish a data flow from a LONWORKS output network variable to one or more LONWORKS input network variables. These LONWORKS network variables may appear on another Pyxos – LONWORKS gateway device, providing communication between two Pyxos FT networks. Or the LONWORKS network variables may appear on other LONWORKS devices that are not Pyxos network gateways. LONWORKS network variables on the same Pyxos – LONWORKS gateway may also be connected.

EXAMPLE:

A Pyxos FT network consists of 16 sensor Points with simple switches on them, and 16 actuator Points connected to lamps. The sensor Points each support a single **SNVT_switch** output PNV, and the actuators each support a single **SNVT_switch** input PNV.

The Pilot implements a Pyxos – LONWORKS gateway, and exposes the output of each of the sensor Points using an array of 16 **SFPTopenLoopSensor** functional blocks, and exposes the actuator Points as an array of 16 **SFPTopenLoopActuator** functional blocks. You can connect the switches and lamps in the Pyxos FT network by connecting the corresponding LONWORKS network variables with LONWORKS connections. These connections can be contained entirely within a single Pyxos FT Pilot, could span multiple Pyxos FT Pilots, or could connect switches or lamps from other LONWORKS devices to the lamps and switches within the Pyxos FT network.

The illustration below shows two such Pyxos Pilots and Pyxos FT networks. On the left there are three switches and a lamp that belong to one Pyxos FT network. The three Open Loop Sensor functional blocks labeled **switch** represents these switches. The first switch is connected over the LONWORKS network to a network variable on the Open Loop Actuator function block on another Pilot that is used to control a lamp device. As a result, the first switch controls a lamp on another Pyxos FT network.

The second switch is connected to another LONWORKS device used to control a lamp device, so that the second switch controls that lamp. The third switch uses a turn-around connection (a connection between two LONWORKS network variables on the same device) to connect the switch to one of the lamps within the same Pyxos FT network. Turn-around connections are handled by the protocol stack and do not cause traffic to be sent on the LONWORKS network when the switch changes state.

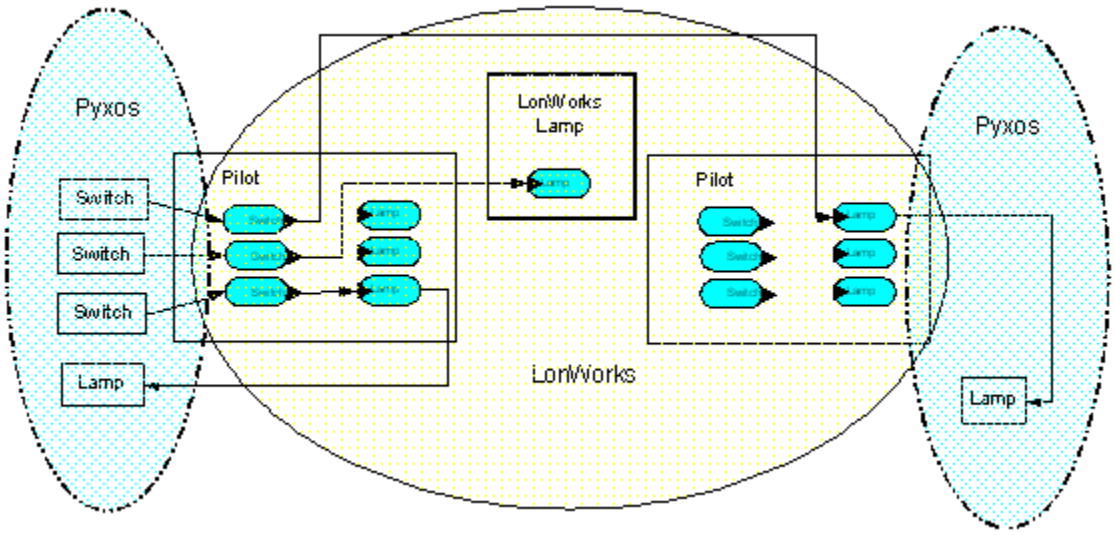

Figure B.1 Example Pyxos – LONWORKS Gateway