



# 9210-2 & XPERT2 DATALOGGERS

---

---

## SDK Manual

---

---

SDK Part No. 8800-1171

Version 3.11

November 17, 2014

Sutron Corporation  
22400 Davis Drive  
Sterling, Virginia 20164  
TEL: (703) 406-2800  
FAX: (703) 406-2801  
WEB: <http://www.sutron.com/>



# Table of Contents

1	INTRODUCTION.....	1
2	INSTALLING AND CONFIGURING THE DEVELOPMENT ENVIRONMENT .....	1
2.1	INSTALLING AND CONFIGURING VISUAL STUDIO 2008 (VS 2008) .....	1
2.2	UNINSTALLING THE DEVELOPMENT ENVIRONMENT.....	2
3	CREATING SLLS .....	2
3.1	OVERVIEW.....	2
3.1.1	Setup Blocks Defined.....	2
3.1.2	Property Pages Defined.....	3
3.1.3	Control Panel Entries Defined.....	4
3.1.4	The Basic Steps to Creating an SLL .....	5
3.1.4.1	Create new project from template .....	5
3.1.4.2	Add Code.....	6
3.1.4.3	Compile, Link, and Download .....	6
3.2	CREATING SETUP BLOCKS.....	6
3.2.1	Constructor.....	6
3.2.1.1	Setup Block Properties .....	6
3.2.2	ShowProperties() .....	7
3.2.3	Initialize() .....	8
3.2.4	Execute().....	9
3.2.4.1	Execute() for Sensor Blocks .....	9
3.2.4.2	Execute() for Passive, Non-Sensor Blocks.....	10
3.2.4.2.1	Identifying Input and Output Indices .....	11
3.2.4.3	Execute() for Scheduled Blocks .....	11
3.2.5	Stop() .....	13
3.2.6	AfterReadSetup().....	13
3.2.7	SetChannelsInuse() .....	13
3.2.8	ClearChannelsInuse().....	13
3.2.9	GetModuleType().....	14
3.2.10	GetDeviceType().....	14
3.2.11	GetModuleProperty().....	14
3.2.12	GetChannelProperty() .....	14
3.2.13	AssignDefaultChannel().....	15
3.2.14	EventExec().....	15
3.2.15	GetIndexOfDataToCalibrate().....	15
3.2.16	SupportsGUICalibrate() .....	15
3.2.17	GUICalibrate() .....	16
3.2.18	SupportsI2CCalibrate() .....	16
3.2.19	I2CCalibrate() .....	16
3.2.20	Calibrate() .....	17
3.2.21	GetScheduleInfo().....	17
3.2.22	Setup Block Icon .....	18
3.2.23	Adding Multiple Blocks to a Single Library.....	18
3.2.24	Adding Inputs and/or Outputs.....	18
3.2.24.1	Update Input/Output Methods.....	18
3.2.24.2	Update Output Buffering Scheme .....	19
3.3	CREATING PROPERTY PAGES.....	19
3.3.1	Adding Multiple Pages to a Single Library .....	19
3.4	CREATING CONTROL PANEL ENTRIES .....	20
3.4.1	Control Panel Buttons .....	20

3.4.2	<i>Making and Saving Changes to Setup Data</i> .....	20
3.5	CREATING “EMPTY” SLLS .....	21
3.6	SIGNALING AN SLL ON APPLICATION INIT/EXIT .....	21
4	APIS .....	21
4.1	ENGINE API.....	21
4.1.1	<i>The Engine Object</i> .....	21
4.1.1.1	hStartEvent .....	21
4.1.1.2	hStopEvent .....	22
4.1.1.3	ModuleList .....	22
4.1.1.4	Run().....	22
4.1.1.5	Stop().....	23
4.1.1.6	IsRunning().....	23
4.1.1.7	SetSchedule().....	23
4.1.1.8	ForceSchedule().....	24
4.1.1.9	LockGUI() .....	25
4.1.1.10	UnlockGUI() .....	25
4.1.1.11	AutoSaveSetup() .....	25
4.1.1.12	LockSetup().....	26
4.1.1.13	UnlockSetup() .....	26
4.1.1.14	StationName.....	26
4.1.1.15	AlarmMgrList .....	27
4.1.1.16	TagList .....	27
4.1.1.17	LockTags().....	27
4.1.1.18	UnLockTags() .....	27
4.1.1.19	InAlarm() .....	28
4.1.1.20	InAlert() .....	28
4.1.1.21	RaiseAlert() .....	28
4.1.1.22	ClearAlert() .....	29
4.1.1.23	ChangeAlarm().....	29
4.1.1.24	ClearAlarm() .....	29
4.1.1.25	EnableAlarm().....	30
4.1.1.26	DisableAlarm().....	30
4.1.1.27	AlarmsEnabled() .....	30
4.1.1.28	GetAlarmStatus() .....	31
4.1.1.29	IOModList.....	31
4.1.1.30	IOModList.GetAnalogIO() .....	31
4.1.1.31	IOModList.GetDigitalIO() .....	31
4.1.1.32	IOModList.GetDisplayIO().....	32
4.1.1.33	IOModList.GetIOMod().....	32
4.1.2	<i>Exported Engine Functions</i> .....	32
4.1.2.1	ChangeNumberDlgInt().....	33
4.1.2.2	ChangeNumberDlgReal() .....	33
4.1.2.3	FileNameDlg().....	33
4.1.2.4	KeypadDlg().....	34
4.1.2.5	MessageDlg() .....	34
4.1.2.6	PasswordDlg() .....	35
4.1.2.7	SetDateTimeDlg() .....	36
4.1.2.8	SetTimeDlg().....	36
4.1.3	<i>CTag Class</i> .....	37
4.1.3.1	CTag() .....	37
4.1.3.2	~CTag() .....	37
4.1.3.3	SetName().....	37
4.1.3.4	CheckName().....	38
4.1.3.5	GetNumValues() .....	38
4.1.3.6	GetAlarm() .....	39

4.1.3.7	SetAlarm()	39
4.1.3.8	GetTag()	39
4.1.3.9	SetTag()	40
4.1.3.10	StartTag()	41
4.1.3.11	StopTag()	41
4.1.3.12	EvalTag()	41
4.1.3.13	IsCurDataTag()	42
4.1.3.14	IsViewableTag()	42
4.1.4	<i>CAlarmMgr Class</i>	42
4.1.4.1	CAlarmMgr()	42
4.1.4.2	~ CAlarmMgr()	43
4.1.4.3	OnRaiseAlert()	43
4.1.4.4	OnClearAlert()	43
4.1.4.5	OnChangeAlarm()	44
4.1.4.6	OnEnableAlarm()	44
4.1.4.7	OnDisableAlarm()	44
4.1.4.8	GetStatus()	45
4.1.4.9	OnEngineRun()	45
4.1.4.10	OnEngineStop()	45
4.2	I/O MODULE API	46
4.2.1	<i>IODevice</i>	46
4.2.1.1	StartRequest()	46
4.2.1.2	StopRequest()	47
4.2.1.3	AuxOnRequest()	47
4.2.1.4	AuxOffRequest()	48
4.2.2	<i>AnalogIO</i>	48
4.2.2.1	SingleVoltageReading()	49
4.2.2.2	DoubleVoltageReading()	49
4.2.2.3	SingleCurrentReading()	50
4.2.2.4	SingleCurrent420maReading()	50
4.2.2.5	SingleResistanceDCReading()	51
4.2.2.6	SingleResistanceACReading()	51
4.2.2.7	SingleThermistorReading()	51
4.2.2.8	RMYoungReading()	52
4.2.2.9	SetConfigurationGain()	52
4.2.2.10	SetConfigurationSingleEnded()	53
4.2.2.11	SetConfigurationDifferential()	53
4.2.2.12	SetConfigurationExcitationHoldOn()	53
4.2.2.13	SetConfigurationExcitationHoldOff()	54
4.2.2.14	SetExcitationChannel()	54
4.2.2.15	SetExcitationVoltage()	54
4.2.2.16	SetExcitationVoltageOn()	55
4.2.2.17	SetExcitationVoltageOff()	55
4.2.2.18	SetFilterNotch()	55
4.2.2.19	SetWarmUpDelay()	56
4.2.2.20	SetPolyAdjust()	56
4.2.2.21	CmdSetAux1()	57
4.2.2.22	CmdPulseOut()	57
4.2.2.23	ReadResistance()	57
4.2.2.24	ReadFrequency()	58
4.2.3	<i>DigitalIO</i>	58
4.2.3.1	ReadCount()	58
4.2.3.2	ReadCountAndTime()	59
4.2.3.3	ReadFilteredInputDataBits()	59
4.2.3.4	ReadAllFilteredInputDataBits()	60
4.2.3.5	SetSamplingSpeed()	60

4.2.3.6	SetLineAsInput()	60
4.2.3.7	SetLineAsOutput()	61
4.2.3.8	SetOutputData()	61
4.2.3.9	InvertIO()	61
4.2.3.10	UnInvertIO()	62
4.2.3.11	SetAsShaftEncoder()	62
4.2.3.12	SetAsCounter()	63
4.2.3.13	ConfigureFilters()	63
4.2.3.14	SetSensitivityHigh()	63
4.2.3.15	SetSensitivityLow()	64
4.2.3.16	AlarmOnSingleEdge()	64
4.2.3.17	AlarmOnBothEdges()	65
4.2.3.18	Configure()	65
4.2.3.19	PulseOut()	65
4.2.3.20	ReadInput()	66
4.2.3.21	ReadFrequency()	66
4.2.4	<i>DisplayIO</i>	67
4.2.4.1	Write()	67
4.2.4.2	WrStringToLCD()	68
4.2.4.3	DisplayLines()	68
4.2.4.4	ShowCursor()	69
4.2.4.5	HideCursor()	69
4.2.4.6	StartBlinkingCursor()	70
4.2.4.7	StopBlinkingCursor()	70
4.2.4.8	ClearDisplay()	70
4.2.4.9	DisplayOff()	71
4.2.4.10	KeyPressed()	71
4.2.4.11	PressKey()	71
4.2.4.12	Flush()	72
4.2.4.13	Read()	72
4.2.4.14	EditFloat()	72
4.2.4.15	EditHEX()	73
4.2.4.16	EditInteger()	73
4.2.4.17	EditString()	74
4.2.4.18	EditTime()	75
4.2.4.19	SelectList()	75
4.2.4.20	OKCancel()	75
4.2.4.21	SetTimeout()	76
4.2.4.22	GetTimeout()	76
4.2.4.23	ResetTimeout()	76
4.2.4.24	TimedOut()	77
4.2.4.25	Error()	77
4.2.5	<i>CIOMod</i>	77
4.2.5.1	GetAnalogPtr()	77
4.2.5.2	GetDeviceType()	78
4.2.5.3	GetDigitalPtr()	78
4.2.5.4	GetModuleNumber()	78
4.2.5.5	GetSerialNo()	79
4.2.5.6	SetEventHandler()	79
4.3	<b>SDI API</b>	79
4.3.1	<i>szSDIAddrSet</i>	79
4.3.2	<i>SendCmd()</i>	80
4.3.3	<i>CollectData()</i>	80
4.3.4	<i>SDIAbort()</i>	81
4.3.5	<i>ClearSDIAbort()</i>	81
4.4	<b>UTILITIES API</b>	81

4.4.1	<i>Report Management</i> .....	82
4.4.1.1	Debug().....	82
4.4.1.2	Warning() .....	82
4.4.1.3	Error().....	83
4.4.1.4	Fatal() .....	83
4.4.1.5	Status() .....	84
4.4.1.6	Maintenance().....	84
4.4.1.7	Note() .....	85
4.4.1.8	SetFilter() .....	85
4.4.1.9	GetFilter().....	86
4.4.1.10	Hook().....	86
4.4.1.11	UnHook().....	87
4.4.2	<i>User Management</i> .....	87
4.4.2.1	CUsers .....	87
4.4.2.1.1	Add().....	87
4.4.2.1.2	Remove() .....	88
4.4.2.1.3	GetUser() .....	88
4.4.2.1.4	UpdateUser() .....	89
4.4.2.1.5	GetUserCount() .....	90
4.4.2.1.6	SetFilter().....	90
4.4.2.1.7	RemoveFilter().....	90
4.4.2.1.8	IsValidUserName().....	90
4.4.2.1.9	IsValidPassword().....	91
4.4.2.1.10	IsValidUserGroup() .....	91
4.4.2.1.11	Commit() .....	92
4.4.2.2	CUser.....	92
4.4.2.2.1	GetName() .....	92
4.4.2.2.2	GetPassword() .....	92
4.4.2.2.3	GetUserGroup() .....	93
4.4.2.2.4	GetTimeoutInterval() .....	93
4.4.2.3	AddCustomGroup().....	93
4.4.2.4	AddCustomCommandParser() .....	94
4.4.3	<i>Serial Communications</i> .....	95
4.4.3.1	CSerialComm() .....	95
4.4.3.2	CSerialComm() .....	95
4.4.3.3	OpenComm().....	96
4.4.3.4	CloseComm() .....	96
4.4.3.5	IsOpen().....	96
4.4.3.6	SetConfiguration().....	96
4.4.3.7	SetCommPort().....	97
4.4.3.8	SetBaudRate() .....	97
4.4.3.9	SetTimeouts().....	98
4.4.3.10	Input Functions .....	98
4.4.3.11	Output Functions.....	99
4.4.3.12	NumberBytesInputBuffer() .....	100
4.4.3.13	KeyPressed() .....	100
4.4.3.14	FlushInput ().....	100
4.4.3.15	GetHandle().....	101
4.4.3.16	WaitOnRx().....	101
4.4.3.17	WaitForTxEmpty ().....	101
4.4.4	<i>Remote Communications</i> .....	101
4.4.4.1	Remote SSP Operations .....	104
4.4.4.1.1	RemoteRequest() .....	104
4.4.4.1.2	RemoteSend() .....	106
4.4.4.1.3	RemoteWaitMessage() .....	106
4.4.4.2	CSocketComm Class.....	108

4.4.4.2.1	CSocketComm()	108
4.4.4.2.2	CSocketComm()	108
4.4.4.2.3	OpenComm()	108
4.4.4.2.4	CloseComm()	109
4.4.4.2.5	IsOpen()	109
4.4.4.2.6	SetConfiguration()	109
4.4.4.2.7	SetCommPort()	110
4.4.4.2.8	SetBaudRate()	110
4.4.4.2.9	SetTimeouts() and SetCommTimeouts()	110
4.4.4.2.10	INPUT FUNCTIONS	111
4.4.4.2.11	OUTPUT FUNCTIONS	111
4.4.4.2.12	WIN32 COMM COMPATIBILITY FUNCTIONS	113
4.4.4.2.13	NumberBytesInputBuffer()	113
4.4.4.2.14	KeyPressed()	114
4.4.4.2.15	FlushInput ()	114
4.4.4.2.16	GetClient ()	114
4.4.4.2.17	WaitOnRx()	115
4.4.4.2.18	WaitForTxEmpty ()	115
4.4.4.2.19	SetCapture()	115
4.4.4.2.20	LockComm()	116
4.4.4.2.21	UnLockComm()	116
4.4.4.2.22	Logout()	116
4.4.4.2.23	SetHost()	117
4.4.4.2.24	GetPortList ()	117
4.4.4.2.25	GetComOptions()	118
4.4.4.2.26	SetComOptions ()	121
4.4.4.2.27	SetExtendedCommands ()	121
4.4.4.2.28	RunCommand ()	122
4.4.5	TResourceKey	122
4.4.6	TTime and TTimeSpan	122
4.4.7	Exported Utils Functions	123
4.4.7.1	StrToTime()	123
4.4.7.2	StrToTimeSpan()	123
4.4.7.3	StrToMS()	123
4.4.7.4	MSTil()	124
4.5	SETUP API	124
4.5.1	Application Exports	124
4.5.1.1	InitSetup()	125
4.5.1.2	ReadSetup()	125
4.5.1.3	WriteSetup()	127
4.5.1.4	GetSetupTag ()	128
4.5.2	CXMLSetup Methods	128
4.5.2.1	WriteStartTag()	128
4.5.2.2	WriteEndTag()	129
4.5.2.3	WriteText()	129
4.6	LOG API	129
4.6.1.1	CLogDesc	130
4.6.1.1.1	CLogDesc()	130
4.6.1.1.2	Create()	130
4.6.1.1.3	Open()	131
4.6.1.1.4	GetName()	131
4.6.1.1.5	SetName()	131
4.6.1.1.6	GetSize()	132
4.6.1.1.7	IsWrap()	132
4.6.1.1.8	IsIgnoreBadData()	132
4.6.1.1.9	InUse()	133



4.6.1.1.10	AppendSensor() .....	133
4.6.1.1.11	AppendNote() .....	133
4.6.1.2	LOG class .....	134
4.6.1.2.1	GetLineCount() .....	134
4.6.1.2.2	Changed() .....	134
4.6.1.2.3	Flush() .....	134
4.6.1.3	LOGCURSOR class .....	135
4.6.1.3.1	LOGCURSOR() .....	135
4.6.1.3.2	~LOGCURSOR() .....	135
4.6.1.3.3	GotoBottom () .....	136
4.6.1.3.4	GotoTop() .....	136
4.6.1.3.5	MoveTo () .....	136
4.6.1.3.6	MoveBy() .....	137
4.6.1.3.7	MoveNext () .....	137
4.6.1.3.8	MovePrev () .....	137
4.6.1.3.9	Search () .....	138
4.6.1.3.10	IsNote () .....	138
4.6.1.3.11	ReadNote () .....	138
4.6.1.3.12	ReadTime () .....	139
4.6.1.3.13	ReadSensor () .....	139
4.6.1.3.14	AtTop () .....	139
4.6.1.3.15	AtBottom () .....	140
4.6.1.3.16	GetCurrentLine() .....	140
4.6.1.3.17	Sync () .....	140
5	CODING GUIDELINES .....	141
5.1	GENERAL .....	141
5.2	GUI GUIDELINES .....	141
6	SAMPLE PROGRAMS, SLL'S, AND BLOCKS .....	142
6.1	TERMINAL SERVER .....	142
6.1.1	<i>TerminalServer.cpp</i> .....	143
6.1.2	<i>TerminalServerMgr.cpp</i> .....	144
6.1.3	<i>TerminalServerMgr.h</i> .....	153
6.1.4	<i>TerminalServerControlPanelEntry.cpp</i> .....	154
6.1.5	<i>TerminalServerControlPanelEntry.h</i> .....	155
6.2	THREADS EXAMPLE .....	156
6.3	ENGINE API EXAMPLES .....	157
6.3.1	<i>hStartEvent and hStopEvent</i> .....	157
6.3.2	<i>ModuleList</i> .....	157
6.3.3	<i>Exported Engine Functions</i> .....	158
6.4	ANALOG I/O .....	158
6.5	DIGITAL I/O – TIPPING BUCKET EXAMPLE .....	160
6.6	SDI API – EXAMPLE .....	162
6.7	REPORT MANAGEMENT API .....	162
6.8	SERIAL COMMUNICATIONS .....	162
6.9	REMOTE COMMUNICATIONS USING SSP .....	163
6.10	LOG API .....	167
6.11	XLITE DISPLAY IO .....	168

**Table of Figures**

Figure 21: Setup Blocks..... 3

Figure 22: Property Pages ..... 4

Figure 23: Control Panel Entries..... 5

Figure 24: Scheduled Pulls and Event-Driven Pushes..... 12

Figure 25: Location of Change Buttons..... 141

Figure 26: Alignment of Standard Buttons..... 142

**Table of Tables**

Table 1: Analog Measure Config Requirements ..... 48

Table 2: Config Parameters Defined..... 49



## 1 Introduction

The Xpert2 Software Development Kit (SDK) enables developers<sup>1</sup> to create Sutron Link Libraries (SLLs)<sup>2</sup> that extend Xpert2's functionality according to the developer's unique needs. The SDK is most often used to create libraries containing custom *setup blocks*, *property pages*, and *control panel entries*, but may also be used to create libraries that manipulate ports, files, peripherals, and any other entity accessible through standard Windows CE operating system API calls.

This document assumes the developer reading it is knowledgeable concerning C++ and Windows programming.

## 2 Installing and Configuring the Development Environment

The SDK is intended for use with Microsoft Visual Studio 2008. This toolset contains the compiler, linker, and operating system libraries necessary to develop for the Xpert platform.

NOTE: prior to Xpert v3.11, Microsoft eMbedded Visual C++ 4.0 was used to develop for Xpert2 and 9210B. This software does not run on any OS above Windows XP, and is becoming very difficult to find. The software's feature set is extremely limited compared to Microsoft Visual Studio 2008. These reasons led to abandoning support for eVC 4.0 beginning with Xpert v3.11.

The steps to install and configure the Visual Studio 2008 development environment follow.

### 2.1 Installing and Configuring Visual Studio 2008 (VS 2008)

The steps for installing and configuring VS 2008 are:

1. Install VS 2008 according to instructions received with the software
2. Uninstall any previous installation of Xpert2 SDK and/or Xpert2 Platform SDK
3. Install Xpert2 Platform SDK (Xpert2PlatformSDK.msi). You need to run the installer from a command prompt with admin privileges for the installation to succeed. Steps:
  - a. Click on command prompt in Start Menu and select "Run as Administrator"
  - b. Change directory to location of Xpert2PlatformSDK.msi
  - c. At prompt, type: `msiexec /i Xpert2PlatformSDK.msi`
4. Add LIB and INCLUDE paths to VS 2008 options to point to the Xpert2 SDK directory (the directory containing the unarchived contents of the zip file you downloaded from Sutron)
  - a. Select Tools – Options – Projects and Solutions
  - b. Select Platform = "Xpert (ARMV4I)"
  - c. Enter the paths indicated for "Include files" and "Library files"

---

<sup>1</sup> Throughout this document, the term *developer* is used to refer to the reader of this document, that is, the person interested in developing an SLL, while the term *user* is reserved for referring to users of the Xpert.

<sup>2</sup> Sutron Link Libraries (SLLs) are standard Windows Dynamic Link Libraries (DLLs) with an extension of "sll".

5. See the “Template” directory in the SDK download for an example project you can use for starting your own project. The template project demonstrates an SLL containing a custom setup block, a custom property page, and a custom control panel entry.

## 2.2 Uninstalling the Development Environment

The development environment can be uninstalled from the “Add or Remove Programs” control panel applet. Select the entry named “Xpert2 SDK” and press the “Change/Remove” button.

## 3 Creating SLLs

“Sutron Link Libraries”, or SLLs, are simply standard Windows dynamic link libraries with the extension “sll” instead of “dll”. This section describes the process of creating SLLs using the Xpert SDK.

### 3.1 Overview

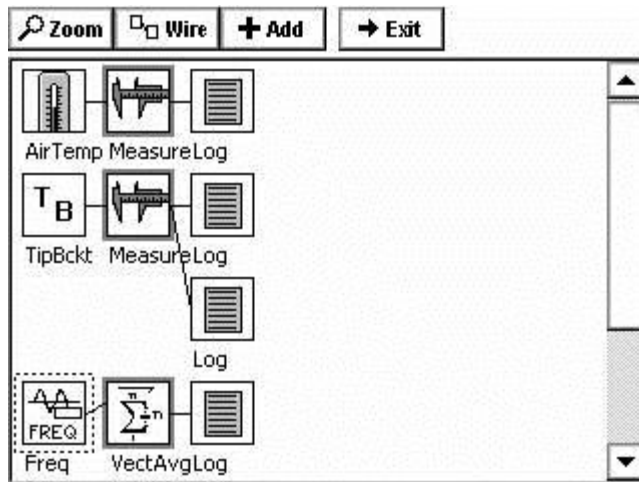
As stated previously, SLLs are most often created to contain *setup blocks*, *property pages*, and/or *control panel entries* designed by the developer for some specific purpose. But what exactly are setup blocks, property pages, and control panel entries? A few words describing each of these things is in order. After that, the steps necessary to create an SLL are defined.

#### 3.1.1 Setup Blocks Defined

A Setup Block is represented visually by an icon on the Xpert graphical setup page. Each block typically has inputs and outputs. The block performs some action on the data it receives (or it may produce data) which it then outputs to blocks connected to its outputs. Hence, the connections between these blocks represent data flowing from block to block.

The Xpert “Engine” is an Xpert software component that controls when data actually flows through the connections between setup blocks. The flows typically occur either due to a schedule submitted to the Engine, or due to some asynchronous event like an alarm generated by a connected device.

The figure below shows an example Xpert graphical setup with a variety of setup blocks. The first line of the example setup has the effect of taking an air temperature measurement at a scheduled time and storing it in a log. To make this actually happen, the Measure setup block was coded to submit an activation schedule to the Engine at recording start and, when activated, “pull” the data from its input which it then “pushes” to its outputs. The AirTemp setup block was coded to take a measurement and output the resulting data when a block connected to its outputs “pulls”. The Log setup block was coded to receive data “pushed” into it and store it in the log identified by the block.



**Figure 1: Setup Blocks**

In the scenario described above, the Measure block is said to be the “active” block, e.g., the block that initiates data flow through a chain of blocks. In this case, the active block is a “scheduled” block (because it schedules itself with the Engine). The other type of active block is an “event-driven” block, where an event such as an I/O module alarm can signal a block to initiate data flow.

Active blocks are special in the sense that there is typically only one in an entire block chain. When active blocks are connected together and one pulls (or pushes) from (or to) another, the pull (or push) is typically ignored. Active blocks are drawn with thick borders to make them easier to identify.

When using the Xpert SDK templates to create a setup block, all of the code necessary to create the shell of a setup block is generated automatically. The developer’s job is to fill-in the details, which usually means making calls into the various APIs to accomplish the desired task. So, the developer must know 1) what API call to make, and 2) where to put it. This document’s “APIs” section helps identify what API calls are necessary. The section “Creating Setup Blocks” helps identify where those calls should go.

### **3.1.2 Property Pages Defined**

A Property Page is a single tab of Xpert’s property sheet interface. For example, the figure below shows the “Main”, “Setup”, “Sensors”, “Data”, “Log”, and “Status” property pages. A page is selected by first selecting the associated tab at the top of the screen.

Simply put, property pages exist to organize related information in a single place (or display pane, more specifically). As part of an extension to Xpert, if a developer wanted to display some set of status data in a single, easily accessible location, a property page would be ideal. Fundamentally, a property sheet is nothing more than a standard Windows dialog.

When using the Xpert SDK template to create a property page, just as with setup blocks, all of the code necessary to create the shell of a property page is generated automatically. The developer's job is to fill-in the details, which usually means displaying and processing standard Windows dialog controls.

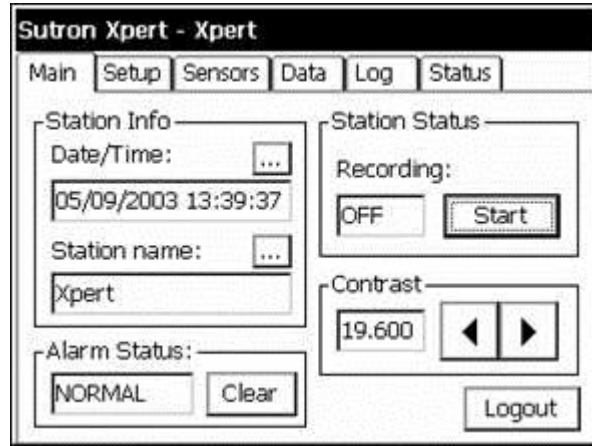


Figure 2: Property Pages

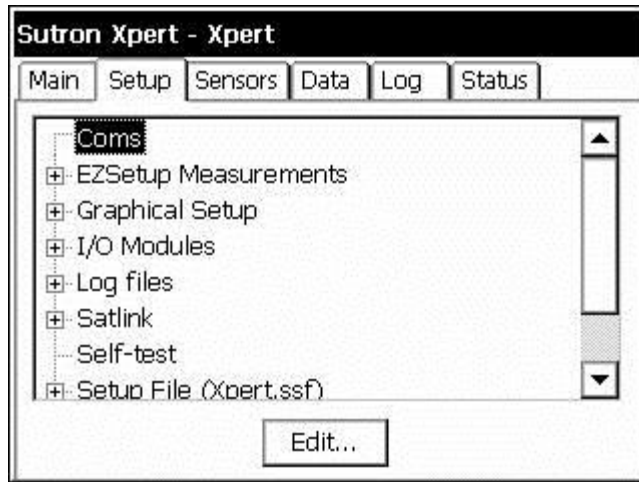
### 3.1.3 Control Panel Entries Defined

A Control Panel Entry is a node of the Xpert's TreeView control panel interface found on the Setup property page. The figure below shows several standard control panel entries: *Graphical Setup*, *I/O Modules*, *Log files*, *Self-test*, and *Setup File*. These entries are "standard" because they are provided by the Xpert application, as opposed to an optional SLL. The entries *Coms*, *EZSetup Measurements*, and *Satlink* are provided by SLLs of similar names.

Control panel entries are typically used to present configuration data to the user for reading and editing. For example, the configuration for an attached device could be accessed here.

When using the Xpert SDK templates to create a control panel entry, just as with setup blocks and property pages, all of the code necessary to create the shell of a control panel entry is generated automatically. The developer's job is to fill-in the details, which usually means displaying and processing standard Windows dialog controls that belong to the dialog created by the user's press of "Edit...".





**Figure 3: Control Panel Entries**

### **3.1.4 The Basic Steps to Creating an SLL**

The basic steps to creating an SLL are:

1. Use the template project delivered with the Xpert SDK to create a project within Visual Studio 2008
2. Add code to perform the functions desired.
3. Compile, link, and download the SLL.

#### **3.1.4.1 Create new project from template**

To create a new project in VS 2008 using the template project provided, do the following:

1. Copy the entire Template directory to a new directory having the name of the new SLL.
2. Rename each file with "Template" in its name to use the name of the new SLL. Don't forget the res subdirectory.
3. For each file in the directory (or for all files in the directory at once, if possible), replace all instances of the string "Template" with the name of the new SLL. Preserve the case of the original string so that, for example, "TEMPLATE\_DLL" is replaced with "NEWNAME\_DLL", while "Template" is replaced with "Newname". Otherwise, you may need to check the project settings for the proper preprocessor define for "NEWNAME\_DLL".
  - a. Don't forget the res subdirectory

By default, the template project contains an example setup block, property page, and control panel entry.

#### 3.1.4.2 Add Code

The template contains areas where source code typically needs to be added or changed with a comment beginning with “TODO”. The developer may want to search on this phrase to ensure all areas that typically require definition have been considered.

The developer’s source code typically makes calls into the various Xpert APIs as well as into the Windows CE operating system API. The Xpert APIs are documented later in this document. The Windows CE operating system API is documented in the online help of Visual Studio 2008.

#### 3.1.4.3 Compile, Link, and Download

To compile and link a release version of the SLL (i.e., one that does not contain debug information), the Solution Configuration must be set to “Release” and the Solution Platform must be set to “Xpert (ARMV4I)”.

Once the correct configurations are selected, press F6 to compile and link the code, i.e., to “build” the SLL. The actual SLL file is placed in a directory named “Xpert (ARMV4I)”, under the main project directory.

After compile and link, download the SLL to the Xpert using the XTerm communications program. The SLL should be downloaded to the Xpert’s “Flash Disk” subdirectory. When the Xpert boots, it loads all SLLs that reside in the Flash Disk subdirectory.

### 3.2 Creating Setup Blocks

Setup blocks are represented in code as classes derived from TModule (see module.h). When creating a setup block using the Xpert SDK, the SDK’s template provides the class specification and overrides of basic methods consistent with the options selected during project creation in the files <block name>.h and <block name>.cpp. These files form the foundation for the developer’s addition of code to perform the processing required for his or her specific application.

To complete the setup block, the developer should review and update (as necessary) the functions provided by the template, and then provide any additional overrides and functions needed. The functions and TModule overrides the developer may need to implement or update are described in the following sections.

#### 3.2.1 Constructor

The setup block constructor is invoked when the setup block is created. It is in the constructor that the class data and properties should be initialized.

##### 3.2.1.1 Setup Block Properties

Properties are something very specific in an Xpert application. A setup block typically has various properties associated with it that are edited using the block’s properties dialog. Some examples are SDI addresses, I/O Module numbers and channels, calibration coefficients, etc.

A property is realized in code as an instance of the class TProperty. The properties are “added” to the block’s property list (which is provided by the base class) in the constructor. Once these properties have been added to the properties list, the properties are saved and restored to and from the setup file automatically.

When a project is created, the template identifies many default properties. The template contains properties used to store output data, as well as properties used to store I/O module and id and channel information. The template contains code in the constructor to initialize these properties and add them to the internal property list.

The developer should declare any additional properties needed in the header file, and then initialize and add the properties to the property list in the constructor.

The example that follows initializes four properties, adds them to the internal property list, and initializes the user-defined attribute `m_iCount` to 0.

```
CMyBlock::CMyBlock()
: TModule(_T("MyBlock"), MyLibSSL.hResource)
{
    // Add analog I/O properties.
    AddProperty(_T("AIOChannel"), m_propAIOChannel = 0);
    AddProperty(_T("AIOModule"), m_propAIOModule = 1);

    // Add digital I/O properties.
    AddProperty(_T("DIOChannel"), m_propDIOChannel = 0);
    AddProperty(_T("DIOModule"), m_propDIOModule = 1);

    // Add SDI properties.
    AddProperty(_T("SDIAddress"), m_propSDIAddress = 0);

    // Initialize class member data.
    m_iCount = 0;
}
```

### 3.2.2 ShowProperties()

Xpert users are used to seeing and editing properties in a dialog box opened via the Edit Properties menu item, gained from selecting a block icon on the setup screen. This dialog is invoked modally within the ShowProperties method. The Xpert application framework calls ShowProperties when it is time to actually show the dialog.

A skeleton of this dialog box is provided by the template and may be found under the “resources” tab of the Visual Studio workspace under the resource id `IDD_<block name>`. The dialog already contains I2C and/or SDI controls, if such were selected during the project’s creation. The developer typically populates the dialog with controls for any properties or data values added outside of the template framework.

The source code for the dialog resides in the files `<block name>Dlg.h` and `<block name>Dlg.cpp`.

The following example shows how the dialog of type `CMyBlockDlg` is used to update properties.

```
bool CMyBlock::ShowProperties(CWnd* pParent)
{
    // This method is called whenever the properties dialog of this block
    // should be shown. The call to this method typically occurs across module
    // boundaries and typically loads resources. Hence, we use TResourceKey
    // to ensure resources are loaded from the correct module.
    TResourceKey key(MyLibSSL);
    CMyBlockDlg dlg(pParent);
```

```

dlg.m_iAIOChannel = m_propAIOChannel;
dlg.m_iAIOModule = m_propAIOModule;
dlg.m_iDIOChannel = m_propDIOChannel;
dlg.m_iDIOModule = m_propDIOModule;
dlg.m_iSDIAddressIdx = m_propSDIAddress;
if (dlg.DoModal() == IDOK)
{
    // TODO: Incorporate data from dialog.
    Engine.IOModList.ClearChannelInUse(ANALOG, m_propAIOModule,
        m_propAIOChannel);
    m_propAIOChannel = dlg.m_iAIOChannel;
    m_propAIOModule = dlg.m_iAIOModule;
    Engine.IOModList.SetChannelInUse(ANALOG, m_propAIOModule,
        m_propAIOChannel);

    Engine.IOModList.ClearChannelInUse(DIGITAL, m_propDIOModule,
        m_propDIOChannel);
    m_propDIOChannel = dlg.m_iDIOChannel;
    m_propDIOModule = dlg.m_iDIOModule;
    Engine.IOModList.SetChannelInUse(DIGITAL, m_propDIOModule,
        m_propDIOChannel);

    m_propSDIAddress = dlg.m_iSDIAddressIdx;
    return true;
}
return false;
}

```

### 3.2.3 Initialize()

Initialize is called by the Xpert application framework for every setup block just after a user presses the Start button, and just before the Engine begins executing scheduled actions. Generally speaking, any data or I/O Module initialization that should occur prior to each start of recording should be performed by Initialize.

The template contains code to reinitialize the block's output data to a default state. The Initialize method is probably a good place to reinitialize any other data added by the developer.

Blocks using Digital I/O Modules often need to initialize their connected device to prepare for performing cyclic actions following start, as in the following example:

```

DigitalIO* pDigIO = Engine.IOModList.GetDigitalIO(m_propDIOModule);
if (pDigIO)
{
    pDigIO->SetAsCounter(m_propDIOChannel);
    pDigIO->SetSamplingSpeed(0.5); //set to max speed, see i2c device spec
    pDigIO->StartRequest();
}
else
    Report.Error(_T("Failed to get digital module.\r\n"));

```

Blocks that want to listen for I/O Module alarms typically need to initialize their connected module to prepare for generating alarms following start:

```

DigitalIO* pDigIO = Engine.IOModList.GetDigitalIO(m_propDIOModule);
if (pDigIO)

```

```

    pDigIO->AlarmOnSingleEdge(m_propDIOChannel);
    pDigIO->SetAlarm(m_propDIOChannel, 1);
    pDigIO->ActivateAlarm(m_propDIOChannel);
    pAlarmHandler = new I2CEVENTHANDLER(m_propDIOChannel, *this);
    pMod->SetEventHandler(*pAlarmHandler);
    pDigIO->StartRequest();
}
else
    Report.Error(_T("Failed to get digital module.\r\n"));

```

To create a “scheduled” block (i.e., one that initiates its own processing, as opposed to responding to the push or pull of other blocks), then add code to register its schedule for execution with the engine as part of the Initialize method. A schedule is registered by calling Engine’s SetSchedule method with the appropriate timing parameters. This call tells Engine to invoke this method’s Execute method according to the schedule provided. See the Engine API for a more detailed description of SetSchedule.

```

Engine.SetSchedule(m_propOffset.AsCString(), m_propInterval.AsCString(),
    *this);

```

### 3.2.4 Execute()

The Execute method is called as part of a push or pull of data through the system initiated by Engine. When the Execute method is invoked, it is a signal to the block to perform the function the block was created to perform. Because sensor blocks have an Execute method that is usually very different from non-sensor and scheduled blocks, they are treated separately in the following.

#### 3.2.4.1 Execute() for Sensor Blocks

The invocation of Execute for a sensor block means that it’s time for the sensor to take its measurement, perform any necessary calculations, and assign the results to the output property data.

Since an invocation of Execute does not happen instantly, it buffers output data to stack data and only assigns data to the output properties once all data has been completely determined. The assignment to the output properties occurs within a data lock in order to ensure a consistent data set.

The basic course of events within the Execute method for a sensor block follows:

1. Prior to taking the measurement:
  - a. Use the passed parameter tScheduled to set the buffered sensor data’s scheduled time value.
  - b. Initialize the buffered sensor data’s actual time value to the current time.
  - c. Initialize the buffered sensor data’s quality flag to bad.
  - d. Initialize the buffered sensor data’s data value to 0.
2. Take the measurement. This largely consists of making calls to either the I2C or SDI API, depending on the way the sensor hardware communicates with Xpert. Note that these calls likely configure the sensor prior to reading its measurement.
3. After the measurement, buffer the results within a data lock. The methods LockData() and UnlockData() are accessible from the TModule base class.

The Execute method of a module taking a simple voltage measurement would be defined as follows. Note that the template code provides all of the following except the part that actually takes the measurement.

```
void CMyBlock::Execute(TTime tScheduled)
{
    // Initialize intermediate Output1 data quality to bad.
    CSensorData Output1Data = m_Output1Data;
    Output1Data.TimeScheduled = tScheduled;
    Output1Data.TimeActual = TTime::GetCurrentTime();
    Output1Data.Quality = CSensorData::BAD;
    Output1Data.Data = 0.0;
    CSensorData RawData = LastData;

    // Get handle to analog module.
    AnalogIO* pAnalogIO = Engine.IOModList.GetAnalogIO(m_propAIOModule);
    if (!pAnalogIO)
    {
        Report.Error(_T("CMyBlock::Execute: Failed to get module.\r\n"));
        return;
    }

    // Take the measurement.
    double Voltage;
    I2CCODE code = pAnalogIO->SingleVoltageReading(m_propAIOModule,
        Voltage);
    if (code == I2C_OK)
    {
        Output1Data.Data = Voltage;
        Output1Data.Quality = CSensorData::GOOD;
    }
    else
        Report.Error(_T("CMyBlock::Execute: Volt reading failed.\r\n"));

    // Buffer output data.
    LockData();
    m_Output1Data = Output1Data;
    UnlockData();
}
```

#### 3.2.4.2 Execute() for Passive, Non-Sensor Blocks

Blocks that are not of type input and not scheduled, typically retrieve data from one or more of the blocks connected as inputs, process the data, and then assign the results to the output property data.

The following example demonstrates a block taking its two inputs and adding them together to produce its output.

```
void CMyBlock::Execute(TTime tScheduled)
{
    // Initialize intermediate Output1 data quality to bad.
    CSensorData Output1Data = m_Output1Data;
    Output1Data.TimeScheduled = tScheduled;
    Output1Data.TimeActual = TTime::GetCurrentTime();
    Output1Data.Quality = CSensorData::BAD;
```

```

Output1Data.Data = 0.0;

// Retrieve inputs.
CSensorData sdInput1, sdInput2;
GetInputData(1, sdInput1); // input1 connected to input point 1
GetInputData(3, sdInput2); // input2 connected to input point 3
Output1Data.Data = sdInput1.AsInteger() + sdInput2.AsInteger();
if (sdInput1.Quality == CSensorData::GOOD &&
    sdInput2.Quality == CSensorData::GOOD &&
    Output1Data.Data.AsInteger() > 0)
    Output1Data.Quality = CSensorData::GOOD;

// Buffer the output data.
LockData();
m_Output1Data = Output1Data;
UnlockData();
}

```

Note how this example tests the quality of the inputs and the range of the result to determine if the result is valid. In general, if something goes wrong in the calculation, or if the inputs or result do not meet expectations, the quality should be set to bad, as it is initially in this example.

Also note the call to `GetInputData()` in the example. This function retrieves the data from the module connected to the specified input. The next paragraph describes how to tell what input to specify.

#### 3.2.4.2.1 Identifying Input and Output Indices

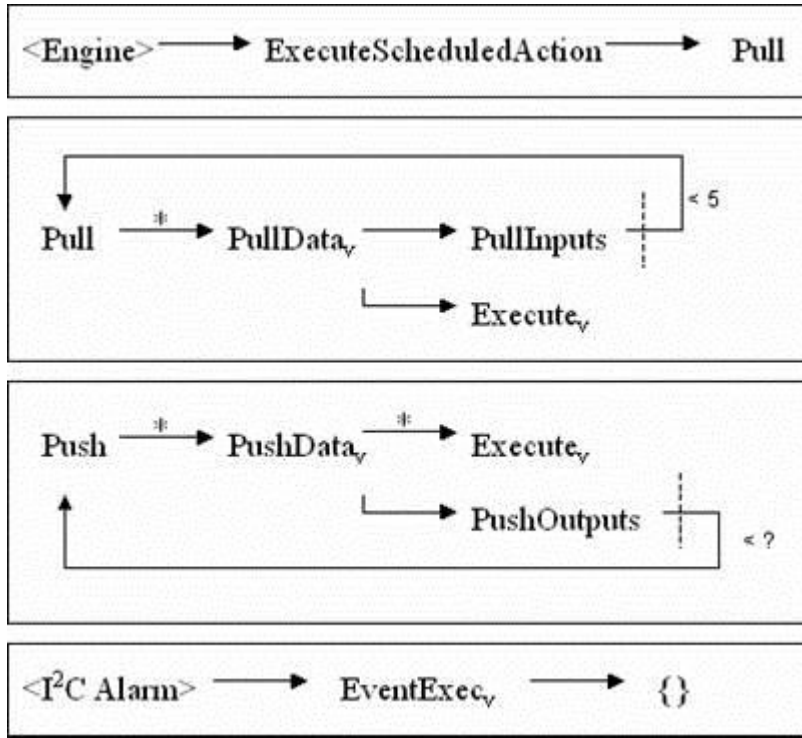
Every block can have up to five inputs and the inputs are enumerated from 0 to 4. When a block has only one input, its connection point is always at point 2. When a block has two inputs, the top input is at point 1 and the bottom input is at point 3. When a block has three inputs, the top input is at point 0, the middle input is at point 2, and the bottom input is at point 4. When a block has four inputs, the top input is at point 0, the first middle input is at point 1, the second middle input is at point 3, and the bottom input is at point 4. When a block has five inputs, the points run 0 to 4, from top to bottom.

Indices are assigned to outputs in the same way they are assigned to inputs. Hence, the center output point is indexed as point 2, and so on.

#### 3.2.4.3 Execute() for Scheduled Blocks

Blocks that are scheduled are “pulled” by the Engine by default when it is time to run. This means that whenever any other block pulls the scheduled block (as would be the case when another scheduled, or in some way active, block performed a pull), the scheduled block would behave as if it were it running on schedule. This may or may not be the desired behavior. Determining as much depends on the unique requirements of the block being developed.

Typically, the desire is to have scheduled blocks ignore pushes and pulls from other blocks, and to only execute when it is told to do so by the engine on schedule. Achieving this behavior is simple, but requires an understanding of how blocks are told to execute by the engine on schedule. The following diagram helps to illustrate how this occurs. Note: in the diagram, an asterisk indicates a lock request (i.e., could result in a block), a subscripted “v” indicates the function is virtual, and a dashed line indicates a change in TModule context (a call to a different block).



**Figure 4: Scheduled Pulls and Event-Driven Pushes**

From the diagram it is apparent that, to stop responding to pulls and pushes by other modules, the virtual functions PullData and PushData can be overridden to do nothing but return. But this has the unwanted side-effect of also ignoring the engine's pull of the block on schedule. To restore this functionality, ExecuteScheduledAction can be overridden to call PullInputs, Execute, and PushOutputs successively, which must occur within a lock of PullLock, as in the following example:

```

void CMyBlock::ExecuteScheduledAction(TTime tScheduled)
{
    TSingleLock SingleLock(PullLock);
    if (SingleLock.Lock(INFINITE))
    {
        PullInputs(tScheduled, false);
        Execute(tScheduled);
        PushOutputs(tScheduled);
    }
}

```



An alternate form of `PushOutputs()` is available which will push only a single output point instead of the default of pushing all of them. For instance to push only the blocks connected up to the middle output (#2), `PushOutputs(tScheduled, 2)` can be called. For instance, one output point might typically be connected to a binary output block while another may be connected to a log block. It may be undesirable to force data to be logged everytime the binary output needs to be changed.

Also apparent from the diagram is that I2C alarms don't actually translate to event-driven pushes as a matter of course. Typically, `EventExec` is overridden to call `PushOutputs` when the alarm information passed-in meets the correct criteria.

### 3.2.5 Stop()

The `Stop` method is called by the Xpert application framework when the user has pressed the `Stop` recording button. If the sensor uses a Digital I/O Module and has made a previous start request, then the `Stop` method should be overridden to request stop, as in the following example:

```
void CMyBlock::Stop()
{
    DigitalIO* pDigIO = Engine.IOModList.GetDigitalIO(m_propDIOModule);
    if (pDigIO)
        pDigIO->StopRequest();
    else
        Report.Error(_T("CMyBlock::Stop: Failed to get module.\r\n"));
}
```

### 3.2.6 AfterReadSetup()

`AfterReadSetup` is called by the framework after a setup file is read in.

```
void CMyBlock::AfterReadSetup(int BaseIndex)
{
    // Place any code that should be run when a new setup is read-in here
    TModule::AfterReadSetup(BaseIndex);
}
```

### 3.2.7 SetChannelsInuse()

`SetChannelsInuse()` is called by the framework after a setup file is read in, or whenever a change to the channel or module property is made. When using I2C I/O devices, override this method to claim channel usage.

```
void CMyBlock::SetChannelsInuse()
{
    Engine.IOModList.SetChannelInUse(DIGITAL, m_propDIOModule,
        m_propDIOChannel);
}
```

### 3.2.8 ClearChannelsInuse()

`ClearChannelsInuse()` is called before the channel or module properties of a block are changed. When using I2C I/O devices, override this method to release channel usage.

```
void CMyBlock::ClearChannelsInuse()
{
    Engine.IOModList.ClearChannelInUse(DIGITAL, m_propDIOModule,
```

```

        m_propDIOChannel);
    }

```

### 3.2.9 GetModuleType()

GetModuleType() is called by the framework to determine the type of the setup block, which can be one of the following as defined by TModule::ModuleType: *INPUT*, *OUTPUT*, *PROCESSING*, *CALCULATION*, *LOGGING*, *TELEMETRY*, or *OTHER*.

```

void CMyBlock::GetModuleType()
{
    return TModule::INPUT;
}

```

### 3.2.10 GetDeviceType()

GetDeviceType() is called by the framework to determine the type of input/sensor block, which can be one of the following as defined by the global enum CIODeviceType: *UNKNOWN\_DEVICE*, *ANALOG*, *DIGITAL*, *DISPLAY*, *SDI12*, *RS232*, *RS485*.

Note: When developing a block that uses an analog or digital device, but not both, then it is possible to simplify the code by deriving your block (TModule) from either TAnalogModule or TDigitalModule. These classes provide default implementations of several methods to reduce the amount of code that needs to be written. The methods provided are: GetDeviceType(), ClearChannelsInuse(), SetChannelsInuse(), GetIOAddrAsString(), AssignDefaultChannel(), GetModuleProperty(), and GetChannelProperty().

```

void CIODeviceType::GetDeviceType()
{
    return ANALOG;
}

```

### 3.2.11 GetModuleProperty()

This method returns a handle to the property that identifies the I/O module used by the block. The When I2C I/O modules are used, override this method to return the indicated property.

This method supports the use of the block in the context of *EZSetup Measurements*.

```

TProperty* CMyBlock::GetModuleProperty()
{
    return &m_propAIOModule;
}

```

### 3.2.12 GetChannelProperty()

This method returns a handle to the property that identifies the device channel, com port, or SDI address used by the block. When I2C I/O modules are used, override this method to return the indicated property.

This method supports the use of the block in the context of *EZSetup Measurements*.

```
TProperty* CMyBlock::GetChannelProperty()
{
    return &m_propAIOChannel;
}
```

### 3.2.13 AssignDefaultChannel()

This is called by the framework when a new input block is created to find and set default channel and module properties of the block. Typically Engine.IOModList.FindChannel() is called to find an unused I2C channel, but this is handled automatically if the block is derived from TAnalogModule or TDigitalModule and only requires a single channel.

```
void CMyBlock::AssignDefaultChannel()
{
    Engine.IOModList.FindChannel(ANALOG, IOModule.AsInteger(),
        IOChannel.AsInteger());
    Engine.IOModList.FindChannel(ANALOG, IOModule.AsInteger(),
        IOExcitationChannel.AsInteger());
}
```

### 3.2.14 EventExec()

EventExec() is called by the Engine when a block has indicated it responds to I/O module alarms (i.e., it has created an instance of I2CEVENTHANDLER; see Initialize above) and an alarm has occurred. Override this method to perform processing when a registered alarm occurs. For example, to have the block output its data to all blocks connected to its outputs, the override might look as follows:

```
void CMyBlock::EventExec(BYTE EventType, UINT32 Time)
{
    PushOutputs(TTime::GetCurrentTime());
}
```

### 3.2.15 GetIndexOfDataToCalibrate()

Override this method to change which output value is used for calibration. The result of GetDefaultOutput() is used by default.

```
int CMyBlock::GetIndexOfDataToCalibrate()
{
    return 0; // Calibrate based on the first output
}
```

### 3.2.16 SupportsGUICalibrate()

Override this method and return true if you wish to allow GUI calibration. The default behavior is to allow GUI calibration if the module supports an Offset or a CalOffset property. See GUICalibrate() for more details.

```
bool CMyBlock::SupportsGUICalibrate()
{

```

```

    return true;
}

```

### 3.2.17 GUICalibrate()

This method is invoked by the framework when the user presses the “Cal...” button on the Sensors page. By default, a message is displayed indicating no calibration is necessary. Override SupportsGUICalibrate() such that it returns true, and this method to guide the user through the calibration procedure. For example:

```

bool CMyBlock:GUICalibrate()
{
    CSensorData sdLastData, sdNewData;
    GetData(GetIndexOfDataToCalibrate(), sdLastData);
    if (sdLastData.IsGood())
    {
        sdNewData = sdLastData;
        if (IDOK == ChangeNumberDlgReal(NULL, sdNewData.Data.AsDouble(),
                                         _T("Enter current value")))
            if (Calibrate(sdLastData, sdNewData))
                return true;
    }
    else
        AfxMessageBox(_T("Quality of current value is not good. ")
                      _T("Please measure first.));

    return false;
}

```

### 3.2.18 SupportsI2CCalibrate()

Override this method and return true if you wish to allow calibration via the I2C Display. The default behavior is to call SupportsGUICalibrate(). See I2CCalibrate() for more details.

```

bool CMyBlock::SupportsI2CCalibrate()
{
    return true;
}

```

### 3.2.19 I2CCalibrate()

This method is invoked by the framework when a user selects calibrate from a connected I2C display. By default, setup blocks do not support this type of calibration. To provide support, override this method to perform the calibration and override TModule::SupportsI2CCalibration() to return true, indicating the setup block supports calibration via the I2C display. For example:

```

bool CMyBlock::I2CCalibrate()
{
    CSensorData sdLastData, sdNewData;
    GetData(GetIndexOfDataToCalibrate(), sdLastData);
    DisplayIO* pDisp = Engine.IOModList.GetDisplayIO(1);
    if (pDisp)

```

```

{
    if (sdLastData.IsGood())
    {
        DisplayIO::EditStatus status;
        sdNewData = sdLastData;
        status = pDisp->EditFloat(sdNewData.Data.AsDouble(), _T("Cur.Val"));
        if (status == DisplayIO::EDIT_OK)
            if (Calibrate(sdLastData, sdNewData))
                return true;
    }
    else
    {
        pDisp->Write(_T("Cur val is bad"));
        pDisp->Write(_T("Measure val first"), 1);
        Sleep(3000);
    }
}
return false;
}

```

### 3.2.20 Calibrate()

Override this method to determine how calibration is performed. By default if a CalOffset property exists then this is updated to reflect the difference between the old and desired value, otherwise the Offset property is adjusted. If neither exists false is returned.

```

bool CMyBlock::Calibrate(CSensorData& sdOld, CSensorData& sdNew)
{
    CalOffset = sdNew.Data.AsDouble() - sdOld.Data.AsDouble()
                + CalOffset->AsDouble();
    LastData = sdNew;
    return true;
}

```

### 3.2.21 GetScheduleInfo()

This method is invoked by the framework for scheduled blocks to retrieve a string describing the block's schedule. This string is displayed in the control panel under *Graphical Setup* for each of the input blocks shown there.

```

CString CMyBlock: GetScheduleInfo()
{
    CString str;
    TTime timeNext = TTime(0);
    if (Engine.IsRunning())
    {
        TTimeSpan tsTime = StrToTimeSpan(Time);
        TTimeSpan tsInterval = StrToTimeSpan(Interval);
        timeNext = NextScheduledTime(TTime::GetCurrentTime(),
                                     tsInterval, tsTime);
    }
    str.Format(_T("%s (Next: %02d:%02d:%02d)"), Interval.AsLPCTSTR(),
              timeNext.GetHour(), timeNext.GetMinute(), timeNext.GetSecond());
    return str;
}

```

### 3.2.22 Setup Block Icon

This icon is stored in a bitmap file in the “/res” project subdirectory. The icon may be replaced with a custom bitmap simply by replacing or editing the bitmap file. The Visual Studio development tool provides bitmap editing tools.

### 3.2.23 Adding Multiple Blocks to a Single Library

1. In <library name>.cpp, modify TFactory::TFactory constructor to initialize its module list with all required blocks. This typically involves updating the definitions for both Modules and ModuleCount. For example:

```
TFactory::TFactory()
{
    static TCHAR* Modules[]={_T("MyFirstBlock"), _T("MySecondBlock")};
    ModuleCount = 2;
    ModuleList  = Modules;
}
```

2. In <library name>.cpp, modify TFactory::CreateModule() to create multiple blocks, based on the index received (which ranges from 0 to ModuleCount-1). For example:

```
TModule* TFactory::CreateModule(int Index)
{
    switch(Index)
    {
        case 0: return dynamic_cast<TModule *>(new CMyFirstBlock());
        case 1: return dynamic_cast<TModule *>(new CMySecondBlock());
        default: return NULL;
    }
}
```

3. Create a bitmap to serve as an icon for the new setup block. Start with a copy of the bitmap included in the template since it already has the correct color depth and size attributes. Add this new bitmap to the project in Visual Studio.
4. Use the source and header files of the setup block template as a starting point for new source and header files for the new setup block. Add these files to the project. Repeat this step for the files to be used for the new block's properties dialog.
5. Create the dialog resource for the properties dialog. Update the resource identifier referenced in the properties dialog header file with the new resource identifier.

### 3.2.24 Adding Inputs and/or Outputs

Adding new inputs or outputs to a block can be done by updating the methods of the block class that define the number of inputs (outputs) exist, whether the inputs (outputs) are active, and the corresponding names. When the number of output points changes, it is usually necessary to update the output point buffering scheme.

#### 3.2.24.1 Update Input/Output Methods

The number of inputs is defined by the method InputCount(). The number of outputs is defined by the method OutputCount().

The method `IsActive()` is used to define whether a given input is active (i.e., connectable). The corresponding method for outputs is `OutputIsActive()`. These methods should return true for any connection point that is active. For example, if input point 2 (the center input<sup>3</sup>) is connectable, then `IsActive(2)` should return true. These methods should return false for any connection point that is not active.

The method `InputName()` is used to define the name, or label, of a given input. The corresponding method for outputs is `OutputName()`. These methods should return a `CString` object containing the name of the input (or output).

#### 3.2.24.2 Update Output Buffering Scheme

When the number of outputs changes, it is usually necessary to update the output point buffering scheme, which consists of buffer variables and accessor methods.

Whenever a block asks for data from another block using `GetInputData()`, the data that is returned comes from the provider block's internal buffer associated with the output. In the simplest case of one output, the variable *LastData*, provided by the `TModule` base class, is used to buffer the output for point 2. The templates code automatically manages this buffering.

When a block has more than one output, separate buffers must be added as class variables for each output. When the values for the outputs are determined (typically during `Execute()`) they must be assigned to the buffered output variables inside a data lock. The accessor methods `GetData()` and `SetData()` serve as the public interface to this data.

### 3.3 Creating Property Pages

The property page template provides an empty property page in the files `<page name>.h` and `<page name>.cpp`. The page is encapsulated within a class named `C<page name>` derived from `CModPropPage`. The developer uses standard Windows' dialog controls to present and obtain data to and from the user.

#### 3.3.1 Adding Multiple Pages to a Single Library

1. In `<library name>.cpp`, modify `TFactory::TFactory` constructor to initialize its page list with all required pages. This typically involves updating the definitions for both `Pages` and `PageCount`. For example:

```
TFactory::TFactory()
{
    static TCHAR* Pages[]={_T("MyFirstPage"), _T("MySecondPage")};
    PageCount = 2;
    PageList = Pages;
}
```

2. In `<library name>.cpp`, modify `TFactory::CreatePage()` to create multiple pages, based on the index received (which ranges from 0 to `PageCount-1`). For example:

---

<sup>3</sup> For a detailed discussion of how to identify the index of an input or output point, see “3.2.4.2.1 Identifying Input and Output Indices”.

```

TModule* TFactory::CreatePage(int Index)
{
    switch(Index)
    {
        case 0: return dynamic_cast<CModPropPage*>(new CMyFirstPage());
        case 1: return dynamic_cast<CModPropPage*>(new CMySecondPage());
        default: return NULL;
    }
}

```

3. Use the template property page source and header files as a starting point for new source and header files for the new property page. Add these files to the project.
4. Create the dialog resource for the property page. Update the resource identifier referenced in the property page header file with the new resource identifier.

### 3.4 Creating Control Panel Entries

The SDK template provides an empty control panel entry in the files <entry name>.h and <entry name>.cpp. The entry is encapsulated within a class named C<entry name> derived from CControlPanelEntry. The developer accesses this entry using standard Windows' tree control routines to present data to the user.

#### 3.4.1 Control Panel Buttons

When a node of a tree is selected, the EditItem method is called which in turn instantiates and invokes modally a dialog intended to allow the user to edit the selected item. This dialog template is encapsulated in a class named C<entry name>Dlg, and appears in the files <entry name>Dlg.h and <entry name>Dlg.cpp. The developer uses standard Windows' dialog controls to present and obtain data to and from the user to edit the selected item.

The control panel also supports two other buttons (labeled New and Delete by default). They may be activated and named by overriding GetButtonLabels() and implemented by overriding NewItem() and DeleteItem() methods.

#### 3.4.2 Making and Saving Changes to Setup Data

The control panel is where the user makes changes to data that is typically related to system setup. It often the case that these changes must occur with recording stopped. Also, it is often the case that these changes need to be saved upon leaving the control panel. The base class of the control panel entry (CControlPanelEntry) provided by the template provides support for these special situations.

When the system is recording, it is using system setup data. If changes are to be made to this data, system recording must be stopped first. This is typically done by calling PromptStopEngine() before the changes are made. This method of CControlPanelEntry prompts the user for permission to stop recording. If the user agrees, recording is stopped by calling Engine.Stop() and then a flag is set (m\_bRecStopped) to true so that the system prompts to turn recording back on once the user leaves the control panel.

When changes are made to system setup data, it is the system's convention to save these changes automatically when the user tabs away from the setup tab/control panel page. To ensure this is done, set m\_bSaveNeeded to true after the changes are made. This causes the system to save the



system setup automatically when the user leaves the control panel (just before prompting to turn recording back on, if it was turned off).

### 3.5 Creating “Empty” SLLs

It is possible to create an SLL that does not contain a setup block, property page, nor control panel entry. Such an SLL could be programmed to perform countless functions given the developer has access to the complete Windows CE API, in addition to the Xpert API. To create an empty SLL, simply start with the template code but exclude templates for setup block, property page, and control panel entry.

### 3.6 Signaling an SLL on Application Init/Exit

It is possible to have Xpert “signal” an SLL at both application initialization and termination by defining and exporting the functions “extern “C” \_\_declspec(dllexport) void AppInit()” and “extern “C” \_\_declspec(dllexport) void AppExit()”, respectively. When the Xpert application starts, it searches for the AppInit function in all loaded SLLs and calls them if found. At application termination (when the user presses “Exit App” on the status page), Xpert searches each SLL for the function AppExit, and executes it if found.

The call to AppInit() is the ideal point from which to spawn new execution threads, should this be required. The call to AppExit() is the ideal point from which to signal those threads to exit. An example of this type of processing is included in the examples section of this document.

## 4 APIs

The set of functions and data available from Xpert libraries are divided into different groups based on source and purpose. Note: See the Visual Studio online help for documentation of the Windows CE API.

### 4.1 Engine API

The Xpert Engine consists of both an engine object and a set of exported functions. The engine object is an instantiation of CEngine named “Engine” that exists following boot. The exported functions are exported as standard “C” functions.

#### 4.1.1 The Engine Object

This section contains descriptions of the engine object’s public methods and data. Since these methods and data belong to the instance of CEngine named “Engine”, they are accessed using the dot qualifier, as in the following examples:

```
Engine.hStartEvent  
Engine.IOModList.GetAnalogIO()  
etc...
```

##### 4.1.1.1 hStartEvent

This event handle is signaled when the user has pressed the Start recording button. It is reset when the user presses Stop. Use WaitForSingleObject() and/or WaitForMultipleObjects() to wait/test this

event handle. (Note: the engine method “IsRunning()” can also be used to determine if this event handle is set).

```
HANDLE hStartEvent
```

Header

Engine/Engine.h

#### 4.1.1.2 hStopEvent

This event handle is signaled when the user has pressed the Stop recording button. It is reset when the user presses Start. Use `WaitForSingleObject()` and/or `WaitForMultipleObjects()` to wait/test this event handle. (Note: the engine method “IsRunning()” can also be used to determine if this event handle is cleared).

```
HANDLE hStopEvent
```

Header

Engine/Engine.h

#### 4.1.1.3 ModuleList

This member is an array containing handles to each of the setup blocks defined on the setup page. Note: both `TObArray` and `TModule` are defined in `module.h`.

```
TObArray<TModule> ModuleList;
```

Header

Engine/Engine.h

#### 4.1.1.4 Run()

This method is used to programatically start recording. Note: This method does not validate the contents of the setup before starting recording.

```
void Run(bool bShowErrors=true);
```

Parameters

- |                          |  |
|--------------------------|--|
| <code>bShowErrors</code> | - Determines whether errors encountered when starting recording should be displayed to user. This value should be set to false when the user is not expected to be watching the screen during recording start. |
|--------------------------|--|

Return Value

None.

Header

Engine/Engine.h

#### 4.1.1.5 Stop()

This method is used to programatically stop recording.

```
void Stop();
```

Parameters

None.

Return Value

None.

Header

Engine/Engine.h

#### 4.1.1.6 IsRunning()

This method is used to determine whether recording is currently stopped or started.

```
BOOL IsRunning();
```

Parameters

None.

Return Value

TRUE if recording is On, false otherwise.

Header

Engine/Engine.h

#### 4.1.1.7 SetSchedule()

This method is used to establish the execution schedule of a TModule object. After the scheduling call is made, the Engine will execute ExecuteScheduledAction() associated with the module provided. The default action of ExecuteScheduledAction is to pull the module's inputs and invoke the module's Execute method.

```
BOOL SetSchedule(  
    CString Offset,  
    CString Interval,  
    TModule& module);
```

```
BOOL SetSchedule(  
    TTimeSpan Offset,  
    TTimeSpan Interval,  
    TModule& module);
```

```
BOOL SetSchedule(  
    TTimeSpan Offset,  
    TTimeSpan Interval,  
    TTimeSpan StartInterval,  
    TModule& module);
```

## Parameters

- |               |  |
|---------------|--|
| Offset        | - A string or timespan representing the offset into the interval in which to invoke the module's ExecuteScheduledAction() routine. The string should be in the format "HH:MM:SS".  |
| Interval      | - A string or timespan representing the interval at which to invoke the module's ExecuteScheduledAction() routine. The string should be in the format "HH:MM:SS".  |
| StartInterval | - A timespan representing the interval at which to invoke the module's ExecuteScheduledAction() routine for the first time. The string should be in the format "HH:MM:SS". For example, a starting interval of "00:15:00" will cause ExecuteScheduledAction() to be invoked on the next 15-minute interval. Thereafter, the function is invoked according to |
| Module        | - A reference to the module being scheduled.   |

## Return Value

TRUE is returned.

## Header

Engine/Engine.h

### 4.1.1.8 ForceSchedule()

This method causes the ExecuteScheduledAction() routine associated with the specified module to execute immediately. The bEvent flag is set to true when passed to ExecuteScheduledAction() to indicate that the event was forced. This allows a block to have both a regular execution interval as well as an event driven trigger input. When the trigger event occurs, Engine.ForceSchedule() can be called to allow the block to process the data immediately.

```
bool ForceSchedule(const TModule& Module);
```

## Parameters

- |        |   |
|--------|---|
| Module | - Reference to the module to execute immediately. Note that the module must be one that has already been scheduled through a call to SetSchedule. |
|--------|---|

## Return Value

True is returned if module is found and successfully notified to execute immediately.

## Header

Engine/Engine.h

#### 4.1.1.9 LockGUI()

This method is used to guarantee exclusive access to the user interface. This method blocks on a semaphore until exclusive access is available. This method is typically used by worker threads that need to update some element of the GUI (e.g., threads that update control panel entries).

```
bool LockGUI();
```

##### Parameters

None.

##### Return Value

Returns false immediately if GUI is disabled (not running). Otherwise, once the lock is obtained, true is returned.

##### Header

Engine/Engine.h

#### 4.1.1.10 UnlockGUI()

This method is used to release exclusive access to the user interface after it has been obtained by calling LockGUI().

```
void UnlockGUI();
```

##### Parameters

None.

##### Return Value

None.

##### Header

Engine/Engine.h

#### 4.1.1.11 AutoSaveSetup()

This method is used to force a save of the active setup to disk.

```
bool AutoSaveSetup(DWORD dwWait = 0);
```

##### Parameters

dwWait	- Access to the setup must be locked before the autosave can complete successfully. This parameter is the number of milliseconds to wait for the lock to become available, if it is not available already. "INFINITE" may be specified to wait indefinitely.
--------	--

##### Return Value

True is returned if the function is successful. False is returned otherwise.

##### Header

Engine/Engine.h

#### 4.1.1.12 LockSetup()

This method is used to lock access to the setup file for either read-only or exclusive write access. Read-only access is granted to any thread that requests it as long as no write access is active. Hence, multiple threads can obtain read access concurrently. Write access is exclusive, causing all other requests, whether read-only or write, to block. The dwWait parameter determines how long a thread desires to wait for the requested access. If the lock is successful, true is returned. If the lock attempt times-out, false is returned. The thread must call UnlockSetup() to release the lock, specifying whether read-only access was originally requested as a parameter.

```
bool LockSetup(DWORD dwWait = 0, bool bReadOnly = true);
```

##### Parameters

- |           |  |
|-----------|--|
| dwWait    | - The number of milliseconds to wait for the lock to become available, if it is not available already. "INFINITE" may be specified to wait indefinitely. |
| bReadOnly | - When true, read-only access is requested. When false, exclusive write-access is requested.   |

##### Return Value

True is returned when the lock is successful. False is returned otherwise.

##### Header

Engine/Engine.h

#### 4.1.1.13 UnlockSetup()

This method is used to unlock access to the setup file.

```
void UnlockSetup(bool bReadOnly = true);
```

##### Parameters

- |           |  |
|-----------|--|
| bReadOnly | - Specify true when read-only access was requested in the call to LockSetup(). Otherwise, specify false. |
|-----------|--|

##### Return Value

None.

##### Header

Engine/Engine.h

#### 4.1.1.14 StationName

A string containing the name of the station.

```
CString StationName
```

##### Header

Engine/Engine.h

#### 4.1.1.15 AlarmMgrList

The AlarmMgrList is an array which contains a list of all the installed alarm managers (CArmMgr objects) in the system. An alarm manager is typically contained in an SLL (for example Coms.sll) and defines what should occur when an alarm or an alert condition occurs and all other aspects of alarm and alert handling. Be sure to always use LockTags() before and UnLockTags() after accessing the AlarmMgrList directly.

```
TObArray<CArmMgr> AlarmMgrList;
```

#### 4.1.1.16 TagList

The TagList is an array which contains a list of all the communication tags in the system known as CTag. Tags may be looked up by name, or the list may be traversed from top to bottom. Tags are used to mark data values in the setup that need to be transmitted via telemetry or displayed to the user (see the “CTag class” section for more information). Since the TagList is a shared resource, all access to the list must be surrounded by calls to LockTags() and UnLockTags().

```
TMapStringToRef<CTag> TagList;
```

#### 4.1.1.17 LockTags()

Must be called before the TagList is accessed or any Tag in the list is used. This is also used to protect access to the AlarmMgrList.

```
void LockTags();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.1.18 UnLockTags()

Must be called when code is done accessing the TagList or Tags from the list, or the AlarmMgrList so that other sections of the code may access the list.

```
void UnLockTags();
```

Parameters

None

Return Value

None

## Header

Engine/Engine.h

### 4.1.1.19 InAlarm()

Returns true if the system is in alarm. An alarm condition is defined as one or more sensors exceeding their alarm limits.

```
bool InAlarm();
```

#### Parameters

None

#### Return Value

True if the system is in alarm.

## Header

Engine/Engine.h

### 4.1.1.20 InAlert()

Returns true if the system is in alert. An alert condition is defined as a state caused by a change in the alarm condition requiring a transmission to occur.

```
bool InAlert();
```

#### Parameters

None

#### Return Value

None

## Header

Engine/Engine.h

### 4.1.1.21 RaiseAlert()

Puts the system in to an Alert state causing transmissions to occur as appropriate.

```
void RaiseAlert(int ComPort=0);
```

#### Parameters

ComPort	- May be used to indicate which port should respond to the Alert. A value of 0 indicates that an Alert should be sent out on all appropriate ports.
---------	---

#### Return Value

None

## Header



Engine/Engine.h

#### 4.1.1.22 ClearAlert()

Clears an existing Alert State, putting a stop to any Alert transmissions as soon as possible.

```
void ClearAlert(int ComPort=0);
```

##### Parameters

ComPort                      - May be used to indicate which port the alert should be cleared for, otherwise all alerts on all ports are cleared.

##### Return Value

None

##### Header

Engine/Engine.h

#### 4.1.1.23 ChangeAlarm()

May be called by an Alarm block to inform the system that the Alarm status has changed.

```
void ChangeAlarm(bool Alarming);
```

##### Parameters

Alarming                      - Should be set true if a sensor has gone in to alarm. This saves a lot of overhead because by definition if any tag is in alarm the entire system is in alarm, otherwise the system state must be determined by examining every tag.

##### Return Value

None

##### Header

Engine/Engine.h

#### 4.1.1.24 ClearAlarm()

Clears all alarm flags in every tag in the system, hence taking the system out of the alarm state (at least until the next measurement cycle begins and tags start to return to the alarm state).

```
void ClearAlarm();
```

##### Parameters

None

##### Return Value

None

##### Header

Engine/Engine.h

#### 4.1.1.25 EnableAlarm()

Enables Alarm/Alert transmissions in the system.

```
void EnableAlarm();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.1.26 DisableAlarm()

Disable Alarm/Alert transmissions in the system.

```
void DisableAlarm();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.1.27 AlarmsEnabled()

Returns the system alarm enable status.

```
bool AlarmsEnabled();
```

Parameters

None

Return Value

True if Alarm/Alert transmissions are enabled.

Header

Engine/Engine.h

#### 4.1.1.28 GetAlarmStatus()

Returns a string representing the status of all the Alarm Managers in the system. They typically report back the time of next transmission, and/or the time of previous ones.

```
CString GetAlarmStatus();
```

##### Parameters

None

##### Return Value

A multi line text message CR/LF delimited prepared by each of the Alarm Managers in the system.

##### Header

Engine/Engine.h

#### 4.1.1.29 IOModList

This member is an array containing handles to each of the connected I/O modules (instances of CIOMod: see iomod.h).

```
CIOModList IOModList;
```

##### Header

Engine/IOMod.h

#### 4.1.1.30 IOModList.GetAnalogIO()

This method is used to retrieve a pointer to the Analog I/O Module indicated by ModuleNumber.

```
AnalogIO* GetAnalogIO(  
    int ModuleNumber);
```

##### Parameters

ModuleNumber      - The one-based id of the desired analog module.

##### Return Value

A pointer to the analog module is returned if the module is found. NULL is returned if the module is not found.

##### Header

Engine/IOMod.h

#### 4.1.1.31 IOModList.GetDigitalIO()

This method is used to retrieve a pointer to the Digital I/O Module indicated by ModuleNumber.

```
DigitalIO* GetDigitalIO(  
    int ModuleNumber);
```

##### Parameters

ModuleNumber        - The one-based id of the desired digital module.

#### Return Value

A pointer to the digital module is returned if the module is found. NULL is returned if the module is not found.

#### Header

Engine/IOMod.h

#### 4.1.1.32 IOModList.GetDisplayIO()

This method is used to retrieve a pointer to the I2C Display Module indicated by ModuleNumber.

```
DigitalIO* GetDisplayIO(  
    int ModuleNumber);
```

#### Parameters

ModuleNumber        - The one-based id of the desired I2C display module.

#### Return Value

A pointer to the display module is returned if the module is found. NULL is returned if the module is not found.

#### Header

Engine/IOMod.h

#### 4.1.1.33 IOModList.GetIOMod()

This method is used to retrieve a pointer to the I/O Module indicated by Type and ModuleNumber.

```
CIOMod* GetIOMod(  
    CIODeviceType Type,  
    int ModuleNumber);
```

#### Parameters

Type                - Possible values are defined by the enum CIODeviceType:  
                      ANALOG, DIGITAL, or DISPLAY.

ModuleNumber        - The one-based id of the desired module.

#### Return Value

A pointer to the IO module is returned if the module is found. NULL is returned if the module is not found.

#### Header

Engine/IOMod.h

### 4.1.2 Exported Engine Functions

This section contains descriptions of the functions exported from the Engine library.

#### 4.1.2.1 ChangeNumberDlgInt()

This function invokes a dialog used to obtain an integer from the user.

```
int ChangeNumberDlgInt(  
    CWnd* pParent,  
    int& nInput);
```

##### Parameters

- |         |                                      |
|---------|--------------------------------------|
| pParent | - Specifies the parent window.       |
| nInput  | - Reference to the number to change. |

##### Return Value

IDOK is returned if the user selected OK. IDCANCEL is returned if the user selected Cancel.

##### Header

Engine/Module.h

#### 4.1.2.2 ChangeNumberDlgReal()

This function invokes a dialog used to obtain a real number from the user.

```
int ChangeNumberDlgReal(  
    CWnd* pParent,  
    realtype& rInput);
```

##### Parameters

- |         |                                      |
|---------|--------------------------------------|
| pParent | - Specifies the parent window.       |
| rInput  | - Reference to the number to change. |

##### Return Value

IDOK is returned if the user selected OK. IDCANCEL is returned if the user selected Cancel.

##### Header

Engine/Module.h

#### 4.1.2.3 FileNameDlg()

This function invokes a dialog used to obtain a file selection from the user.

```
int FileNameDlg(  
    CWnd* pParentWnd,  
    CString& sReturn,  
    LPCTSTR lpszDir = _T("\\"),  
    LPCTSTR lpszFilters = _T("*.***"),  
    LPCTSTR lpszDefaultExt = _T(""),  
    LPCTSTR lpszDefaultName = NULL);
```

##### Parameters

- |            |  |
|------------|--|
| pParentWnd | - Specifies the parent window.               |
| sReturn    | - The name of the file selected by the user. |
| lpszDir    | - The initial directory.                     |

- |                |  |
|----------------|--|
| lpszFilters    | - A file specification used to filter the list of files available for the user's selection.          |
| lpszDefaultExt | - An extension to apply to the user's selection if the selection does not already have an extension. |

#### Return Value

IDOK is returned if the user selected OK. IDCANCEL is returned if the user selected Cancel.

#### Header

Engine/Module.h

#### 4.1.2.4 KeypadDlg()

This function invokes a dialog used to obtain a string from the user.

```
int KeypadDlg(
    CWnd* pParent,
    CString& text,
    CString Caption,
    CString strType = _T("") );
```

#### Parameters

- |         |  |
|---------|--|
| pParent | - Specifies the parent window.   |
| text    | - The string entered by the user.  |
| Caption | - String to display in the dialog title.   |
| strType | - A string describing the type of entry to be retrieved from user. Enter one of two strings: "PASSWORD" –characters typed by user appear as an asterisk ("*") in dialog; or "FILENAME" – the characters of the text entered are validated as filename candidates when "Ok" is pressed. |

#### Return Value

IDOK is returned if the user selected OK. IDCANCEL is returned if the user selected Cancel.

#### Header

Engine/Module.h

#### 4.1.2.5 MessageDlg()

This function invokes a dialog to display a message to the user. The function accepts a time argument that determines how long the dialog will wait for user input before choosing a default response automatically.

```
int MessageDlg(
    CWnd* pParent,
    const LPCTSTR tstrMsg,
    UINT nWaitSec = 0,
    UINT nType = MB_OK,
    UINT nDefaultResponse = NULL);
```

## Parameters

- |                  |  |
|------------------|--|
| pParent          | - Specifies the parent window.   |
| tstrMsg          | - The message to be displayed.   |
| nWaitSec         | - The number of seconds to wait for user input before choosing the default response. A value of 0 indicates wait indefinitely.   |
| nType            | - The type of message box to display to the user.<br>MB_OK – Dialog has “OK” button.<br>MB_OKCANCEL – Dialog has “OK” and “Cancel” buttons.<br>MB_YESNO – Dialog has “Yes” and “No” buttons.<br>MB_YESNOCANCEL – Dialog has “Yes”, “No”, and “Cancel” buttons.   |
| nDefaultResponse | - The response to choose automatically when the user does not respond to the dialog within the time specified. Typically IDOK, IDCANCEL, IDYES, or IDNO, to correspond with the dialog type.<br><br>If this argument is NULL, then the default response is selected based on the type of the dialog:<br>MB_OK – Response is IDOK.<br>MB_OKCANCEL – Response is IDCANCEL.<br>MB_YESNO – Response is IDNO.<br>MB_YESNOCANCEL – Response is IDCANCEL. |

## Return Value

The value of the response either selected by the user or chosen automatically.

## Header

Engine/Module.h

### 4.1.2.6 PasswordDlg()

This function invokes a dialog used to obtain a password from the user. An asterisk character (\*) is displayed for each character the user types.

```
int PasswordDlg(  
    CWnd* pParent,  
    CString& text);
```

## Parameters

- |         |                                     |
|---------|-------------------------------------|
| pParent | - Specifies the parent window.      |
| text    | - The password entered by the user. |

## Return Value

IDOK is returned if the user selected OK. IDCANCEL is returned if the user selected Cancel.

## Header

Engine/Module.h

#### 4.1.2.7 SetDateTimeDlg()

This function invokes a dialog used to prompt the user for time. The time entered by the user is returned in a SYSTEMTIME structure.

```
int SetDateTimeDlg(  
    CWnd* pParentWnd,  
    TTime& time,  
    SYSTEMTIME& SystemTime);
```

##### Parameters

- |            |   |
|------------|---|
| pParentWnd | - Specifies the parent window.                                      |
| time       | - An initial time to display in the dialog.                         |
| SystemTime | - The SYSTEMTIME structure containing the time entered by the user. |

##### Return Value

IDOK is returned if the user selected OK. IDCANCEL is returned if the user selected Cancel.

##### Header

Engine/Module.h

#### 4.1.2.8 SetTimeDlg()

This function invokes a dialog used to obtain a string representation of time from the user.

```
int SetTimeDlg(  
    CString& text,  
    CWnd* pParentWnd,  
    CString& title,  
    bool bMSec = false,  
    bool b23HrLimit = false);
```

##### Parameters

- |            |   |
|------------|---|
| text       | - The time entered by the user.   |
| pParentWnd | - Specifies the parent window.  |
| title      | - A string to display as the title of the dialog box displayed to the user. |
| bMSec      | - If true, milliseconds are shown on the dialog for the user to enter.      |
| b23HrLimit | - If true, the user is not allowed to enter times greater than 24 hours.    |

##### Return Value

IDOK is returned if the user selected OK. IDCANCEL is returned if the user selected Cancel.

##### Header

Engine/Module.h



### 4.1.3 CTag Class

Tags are used to mark data values in the setup that need to be transmitted via telemetry or displayed to the user. The CTag class is an abstract class. An SLL such as the Coms SLL typically creates and defines a CTag object for each item it wishes to make available and overrides the appropriate virtual methods to define the functionality. CEngine::TagList contains the list of all tags in the system.

Be careful when naming tags, as tag names must be unique.

#### 4.1.3.1 CTag()

Constructs a tag.

```
CTag();  
  
or  
  
CTag(Cstring NewName);
```

##### Parameters

NewName	- Defines the initial name for the tag. If a name is specified the tag is automatically added to the TagList.
---------	---

##### Return Value

None

##### Header

Engine/Engine.h

#### 4.1.3.2 ~CTag()

Destroys a CTag, removing it from the TagList if it had been previously added.

```
~CTag();
```

##### Parameters

None

##### Return Value

None

##### Header

Engine/Engine.h

#### 4.1.3.3 SetName()

Changes the name of the tag. Often tag names are user-defined and can be changed on the fly. Calling SetName() whenever this occurs will remove the old entry in the TagList, and create a new entry with the new name.

Tag names must be unique withing the system. Call CTag::CheckName to determine if a given name is unique.

```
void SetName(CString NewName);
```

#### Parameters

NewName - Defines the new name for the tag. If NewName is not empty, then the tag is added to the TagList under the name.

#### Return Value

None

#### Header

Engine/Engine.h

#### 4.1.3.4 CheckName()

Checks to see if the specified name is unique relative to the taglist. If the name is not in the tag list, or if the name is in the taglist for the current CTag, then true is returned.

This method should be called to determine if a potential tag name is unique before the tag is added to the taglist by SetName().

```
bool CheckName(CString strName);
```

#### Parameters

strName - The name to check against the tag list.

#### Return Value

True if the name is not in the tag list, or if the name is in the taglist for the current CTag.

#### Header

Engine/Engine.h

#### 4.1.3.5 GetNumValues()

A tag can contain one or more values. This returns how many the tag will support. Under SSP a tag is expected to have at least two values, where value 0 contains the primary data reading, and value 1 contains the alarm status. In the Xpert the Coms SLL supports value 2 which returns the same data as value 0, but performs a live reading first by performing an EvalTag().

```
virtual int GetNumValues() = 0;
```

#### Parameters

None

#### Return Value

The number of defined values, usually 2.

#### Header

Engine/Engine.h

#### 4.1.3.6 GetAlarm()

Returns the alarm status for the tag directly. The status is also typically available by using GetTag() for value 1. CSensorData::TAlarmStatus contains the definitions of the various possible alarm bits. The bits used by the Xpert include: HiLimitA, LowLimitA, DigitalA. These are the “alarm” bits and indicate that one or more of the three possible alarm condition exists. The HiLimitC, LowLimitC, and DigitalC bits are called the change bits and indicate that the respective alarm bit has changed state resulting in an alert condition. The “Digital” alarm bit is currently used in the Xpert for indicating Rate of Change alarms, while the other two are used for indicating a high or low limit have been exceeded.

```
virtual int GetAlarm() = 0;
```

##### Parameters

None

##### Return Value

An integer representing a bit mask defined by CSensorData::TAlarmStatus.

##### Header

Engine/Engine.h

#### 4.1.3.7 SetAlarm()

The opposite of GetAlarm, SetAlarm can be used to force the alarm state to the specified value. See the discussion of GetAlarm() for information about the possible alarm states.

```
virtual void SetAlarm(int AlarmState) = 0;
```

##### Parameters

AlarmState                      - An integer representing a bit mask defined by CSensorData::TAlarmStatus.

##### Return Value

None

##### Header

Engine/Engine.h

#### 4.1.3.8 GetTag()

GetTag is used to read values from a tag. This is called in response to an SSP Get Tag message, or when the user views tag in the View Data screen.

```
virtual bool GetTag(int ValueNumber, int& DataType, TValue& Data,  
CSensorData::QualityType Quality) = 0;
```

##### Parameters

ValueNumber	- Since a tag can support multiple values, this specifies which one to retrieve. Typically either 0 or 1, but can be more.
DataType	- Returns the SSP data type of the value as defined in TDataType in ssp.h. Here are the supported data types: dt_long           32-bit integers dt_real           double precision floating point dt_alarm          alarm status information dt_longstr       null terminated strings dt_char           a single character dt_boolean       a 1 byte boolean 0/1 value dt_cardinal      a 16-bit unsigned int dt_integer       a 16-bit signed int dt_nil            a null value
Data	- The actual data value, TValue can only represent 32-bit integers, doubles, and strings, so other data types (if used) must be derived from these types.
Quality	- Quality of the data, can be CSensorData::GOOD, CSensorData::BAD, or CSensorData::Undefined.

#### Return Value

Returns false if the ValueNumber wasn't defined.

#### Header

Engine/Engine.h

#### 4.1.3.9 SetTag()

Sets a tag value to the specified data. This is called in response to an SSP Set Tag message, or when the user tries to change a tag in the View Data screen.

```
virtual bool SetTag(int ValueNumber, int DataType, TValue& Data,
    CSensorData::QualityType Quality) = 0;
```

#### Parameters

ValueNumber	- Since a tag can support multiple values, this specifies which one to set. Typically either 0 or 1, but can be more.
DataType	- The SSP data type of the value as defined in TDataType in ssp.h. See the discussion under GetTag() for more information.
Data	- The string, integer, or double precision value to set the specified tag value to.
Quality	- Quality of the data, can be CSensorData::GOOD, CSensorData::BAD, or CSensorData::Undefined.

#### Return Value

Returns false if the ValueNumber wasn't defined.

#### Header

Engine/Engine.h

#### 4.1.3.10 StartTag()

StartTag is called whenever an SSP Start Tag is received for the specified tag. For data only tags it doesn't usually serve a purpose, but for a tag which implemented a control loop, it would typically initialize and begin execution of the thread which performs the control loop activity.

```
virtual bool StartTag() = 0;
```

Parameters

None.

Return Value

Returns false if the operation failed.

Header

Engine/Engine.h

#### 4.1.3.11 StopTag()

StopTag is called whenever an SSP Stop Tag is received for the specified tag. For data only tags it doesn't usually serve a purpose, but for a tag which implemented a control loop, it would typically safely terminate execution of the thread which performs the control loop activity.

```
virtual bool StopTag() = 0;
```

Parameters

None

Return Value

Returns false if the operation failed.

Header

Engine/Engine.h

#### 4.1.3.12 EvalTag()

EvalTag is called whenever an SSP Eval Tag message is received for the specified tag. The purpose is to evaluate the tag, which might mean take a reading, or execute a control function.

```
virtual bool EvalTag() = 0;
```

Parameters

None

Return Value

Returns false if the operation failed.

Header

Engine/Engine.h

#### 4.1.3.13 IsCurDataTag()

A flag indicating whether a tag should be included in SSP Current Data and Alarm messages. Even if this method returns false, the tag is still accessible via other telemetry methods such as the SSP GetTag or SendTag messages.

```
virtual bool IsCurDataTag() = 0;
```

Parameters

None

Return Value

True if the tag should be included in SSP Current Data and Alarm messages.

Header

Engine/Engine.h

#### 4.1.3.14 IsViewableTag()

A flag indicating whether a tag should be displayed in the View Sensor display.

```
virtual bool IsViewableTag() = 0;
```

Parameters

None

Return Value

True if the tag should be included in View Sensor screen.

Header

Engine/Engine.h

### 4.1.4 CAlarmMgr Class

An Alarm Manager is a class which manages alarms and alerts for a telemetry SLL. The Engine maintains a list of all Alarm Managers in the system in CEngine::AlarmMgrList and uses this list to notify all the managers when an alarm or alert occurs. The alarm or alert may be generated either by the standard Xpert Alarm block, or by a custom routine that has hooked into the engine's alert and alarm interface (see Engine.RaiseAlert() and Engine.ChangeAlarm()).

#### 4.1.4.1 CAlarmMgr ()

The constructor for the class automatically adds the instance to the Engine's Alarm Manager List.

```
CAlarmMgr();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.4.2 ~CArmMgr()

The destructor for the class automatically removes the instance from the Engine's Alarm Manager List..

```
~CArmMgr();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.4.3 OnRaiseAlert()

This is a call back method which is called whenever CEngine::RaiseAlert() is called.

```
virtual void OnRaiseAlert(int ComPort);
```

Parameters

ComPort	- The ComPort the alert should be sent out on, or 0 indicating all ports. This is the same parameter that was passed to CEngine::RaiseAlert().
---------	--

Return Value

None

Header

Engine/Engine.h

#### 4.1.4.4 OnClearAlert()

This is a call back method which is called whenever CEngine::ClearAlert() is called.

```
virtual void OnClearAlert(int ComPort);
```

Parameters

ComPort	- The ComPort the alert should be cleared on, or 0 indicating all ports. This is the same parameter that was passed to CEngine::ClearAlert(). Clearing the Alert condition occurs when
---------	--

an Alert (or Alarm) message is acknowledged. Clearing it should prevent further Alert messages from occurring until a new Alert is raised.

Return Value

None

Header

Engine/Engine.h

#### 4.1.4.5 OnChangeAlarm()

This is a call back method which is called whenever CEngine::ChangeAlarm() is called.

```
virtual void OnChangeAlarm();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.4.6 OnEnableAlarm()

This is a call back method which is called whenever CEngine::EnableAlarm() is called.

```
virtual void OnEnableAlarm();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.4.7 OnDisableAlarm()

This is a call back method which is called whenever CEngine::DisableAlarm() is called.

```
virtual void OnDisableAlarm();
```

Parameters

None

Return Value



None

Header

Engine/Engine.h

#### 4.1.4.8 GetStatus()

This is a call back method which is called whenever CEngine::GetAlarmStatus() is called.

```
virtual CString GetStatus();
```

Parameters

None

Return Value

A CR/LF delimited string representing the current state of the alarm manager and any status information that may be helpful to the user.

Header

Engine/Engine.h

#### 4.1.4.9 OnEngineRun()

This is a call back method which is called whenever CEngine::Run() is called.

```
virtual void OnEngineRun();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

#### 4.1.4.10 OnEngineStop()

This is a call back method which is called whenever CEngine::Stop() is called.

```
virtual void OnEngineStop();
```

Parameters

None

Return Value

None

Header

Engine/Engine.h

## 4.2 I/O Module API

The I/O Module API provided by the Xpert application framework is used to communicate with the I/O Modules connected to the Xpert via an I2C bus. The API consists of five different classes: IODevice, AnalogIO, DigitalIO, DisplayIO, and CIOMod. The IODevice is the parent of the AnalogIO and DigitalIO classes, and so contains methods and data that are common to each. The AnalogIO, DigitalIO, and DisplayIO classes represent Analog I/O Modules, Digital I/O Modules, and I2C Display Modules, respectively. CIOMod “wraps” the AnalogIO, DigitalIO, and DisplayIO classes into a more generic class and introduces the concept of an event handler.

The developer typically accesses I/O Modules using a pointer of type AnalogIO or DigitalIO. The template code contains these pointers in places where module manipulation typically occurs (e.g., at the beginning of the TModule::Execute method). Of course, there may be other locations where module access is desired. The following examples demonstrate creating and initializing pointers to access existing modules:

```
DigitalIO* pDigIO = Engine.IOModList.GetDigitalIO(m_IOMOD);
AnalogIO* pAnalogIO = Engine.IOModList.GetAnalogIO(m_IOMOD);
CIOMod* pMod = Engine.IOModList.GetIOMod(DIGITAL, m_IOMOD);
```

### 4.2.1 IODevice

The IODevice class is the parent of the AnalogIO and DigitalIO classes. As such, it contains methods and data that are common to both Analog and Digital I/O Modules.

#### 4.2.1.1 StartRequest()

This method is used to register the module’s need to start. The Engine starts all modules that have requested such when recording is started. Note: “starting” a module translates to configuring the module, commanding the module to run, and starting to listen for unsolicited events.

```
I2CCODE StartRequest();
```

#### Parameters

None.

#### Return Value

On success, I2C\_OK is returned. The possible values for I2CCODE and their meanings are defined as follows (from Engine\i2cmgr.h):

I2C_OK	- Success/no error.
I2C_NAK	- Message received negative acknowledgment.
I2C_TIMEOUT	- Timed-out waiting for response.
I2C_LOST	- Lost arbitration of the bus.
ARBITRATION	
I2C_OVERFLOW	- Unused.
I2C_BUSERROR	- An error occurred on the I2C bus. Specifically, a misplaced start or stop condition was detected.
I2C_RXERROR	- Error during receive. Specifically, receive did not contain start bit.
I2C_SLAVETX	- Data was received in the slave transmit mode. Xpert only

- supports the master transmit mode.
- I2C\_CHECKSUM - Computed checksum did not match received checksum.
- I2C\_STOP - A premature stop was detected when more bytes were expected.
- I2C\_BUSBUSY - Could not access bus.
- I2C\_RESTART - Received start bit indicating arrival of a new message while reading in a message.
- I2C\_BAD\_CHNL - The commanded channel is not valid.

#### Header

Engine\I2CDeviceClass.h

#### 4.2.1.2 StopRequest()

This method is used to register a request to command the i/o module to stop (send it a stop command opcode). When the number of stop requests total the number of previous start requests, the stop is commanded.

```
I2CCODE StopRequest();
```

#### Parameters

None.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for StartRequest().

#### Header

Engine\I2CDeviceClass.h

#### 4.2.1.3 AuxOnRequest()

This method requests that the “Aux” line (switched battery) be turned on. It is a “request” (as opposed to command) since the number of *on* requests versus the number of *off* requests (via AuxOffRequest) determines the state of the line. If more *on* requests than *off* requests have been received, the line will be switched on.

```
I2CCODE AuxOnRequest();
```

#### Parameters

None.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for StartRequest().

#### Header

Engine\I2CDeviceClass.h

#### 4.2.1.4 AuxOffRequest()

This method requests that the “Aux” line (switched battery) be turned off. It is a “request” (as opposed to command) since the number of *on* requests (via AuxOnRequest) versus the number of *off* requests determines the state of the line. If more *off* requests than *on* requests have been received, the line will be switched off.

```
I2CCODE AuxOffRequest();
```

#### Parameters

None.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for StartRequest().

#### Header

Engine\I2CDeviceClass.h

### 4.2.2 AnalogIO

The AnalogIO class contains methods and data specific to Analog I/O Modules. Many of the methods are measurement command methods that depend on the module having been previously configured using one or more of the configuration command methods. The configuration consists of nine (9) parameters including channel, gain, excitation parameters, etc. Not all measurement commands require that all configuration parameters be set. The tables below define which parameters are required by which measurement commands, and which functions are used to set parameters, respectively.

Meas. Command	Configuration Parameter (see key for description)								
	Ch	G	S/D	E	EV	ECh	EH	FN	WD
SingleVoltageReading	X	X	X	X	X	X	X	X	X
DoubleVoltageReading	X	X	X	X	X	X	X	X	X
SingleCurrentReading	X	X		X	X	X	X	X	X
SingleCurrent420maReading	X	X						X	X
SingleResistanceDCReading	X	X	X	X	X	X	X	X	X
SingleResistanceACReading	X	X	X	X	X	X	X	X	X
SingleThermistorReading	X	X	X	X	X	X	X	X	X
RMYoungReading									

Table 1: Analog Measure Config Requirements

Param	Description	Default Value	Functions to Set Parameter
Ch	- Measurement Channel	N/A	None. Provided by measurement command.
G	- Gain	1	SetConfigurationGain()
S/D	- Single or Differential	Single	SetConfigurationSingleEnded(), SetConfigurationDifferential()
E	- Excitation On/Off	OFF	SetExcitationVoltageOn(), SetExcitationVoltageOff()
EV	- Excitation Voltage	0	SetExcitationVoltage()
ECh	- Excitation Channel	0	SetExcitationChannel()

Param	Description	Default Value	Functions to Set Parameter
EH	- Excitation Hold	0	SetConfigurationExcitationHoldOn(), SetConfigurationExcitationHoldOff()
FN	- Filter Notch	60 hz	SetFilterNotch()
WD	- Warm-Up Delay	50 ms	SetWarmUpDelay()

**Table 2: Config Parameters Defined**

Measurements are typically made in TModule::Execute(). While there are several places the configuration could be commanded, the best place to do it is also in TModule::Execute(), just before the commands to take the measurement. This ensures that reconfigurations due to other module users do not interfere.

The following sections describe the methods provided by the AnalogIO class.

#### 4.2.2.1 SingleVoltageReading()

This method is used to perform a voltage measurement on the indicated channel.

```
I2CCODE SingleVoltageReading(
    const CAnalogChannel& channel,
    double& voltage_measurement);
```

##### Parameters

- channel - The channel on which to measure.
- voltage\_measurement - Reference to a double in which to store the measurement.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/AnalogIO.h

#### 4.2.2.2 DoubleVoltageReading()

This method is used to perform a double voltage measurement on the indicated channel.

```
I2CCODE DoubleVoltageReading(
    const CAnalogChannel& voltage_channel,
    double& voltage_measurement,
    double& excitation_measurement);
```

##### Parameters

- voltage\_channel - The channel on which to measure.
- voltage\_measurement - The voltage measured on the voltage channel.
- excitation\_measurement - The voltage measured on the excitation channel.

## Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

## Header

Engine/AnalogIO.h

### 4.2.2.3 SingleCurrentReading()

This method is used to perform a current measurement on the indicated channel.

```
I2CCODE SingleCurrentReading(  
    const CAnalogChannel& channel,  
    double& current_measurement);
```

## Parameters

- |                         |  |
|-------------------------|--|
| channel                 | - The channel on which to measure.                         |
| current_<br>measurement | - Reference to a double in which to store the measurement. |

## Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

## Header

Engine/AnalogIO.h

### 4.2.2.4 SingleCurrent420maReading()

This method is used to perform a passive current measurement on the indicated channel.

```
I2CCODE SingleCurrent420maReading(  
    const CAnalogChannel& channel,  
    double& current_measurement);
```

## Parameters

- |                         |  |
|-------------------------|--|
| channel                 | - The channel on which to measure.                         |
| current_<br>measurement | - Reference to a double in which to store the measurement. |

## Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

## Header

Engine/AnalogIO.h

#### 4.2.2.5 SingleResistanceDCReading()

This method is used to perform a resistance measurement using DC excitation on the indicated channel.

```
I2CCODE SingleResistanceDCReading(  
    const CAnalogChannel& channel,  
    double& resistancedc_measurement);
```

##### Parameters

- |                          |  |
|--------------------------|--|
| channel                  | - The channel on which to measure.                         |
| resistancedc_measurement | - Reference to a double in which to store the measurement. |

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/AnalogIO.h

#### 4.2.2.6 SingleResistanceACReading()

This method is used to perform a resistance measurement using AC excitation on the indicated channel.

```
I2CCODE SingleResistanceACReading(  
    const CAnalogChannel& channel,  
    double& resistanceac_measurement);
```

##### Parameters

- |                          |  |
|--------------------------|--|
| channel                  | - The channel on which to measure.                         |
| resistanceac_measurement | - Reference to a double in which to store the measurement. |

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/AnalogIO.h

#### 4.2.2.7 SingleThermistorReading()

This method is used to perform a thermistor measurement on the indicated channel.

```
I2CCODE SingleThermistorReading(  
    const CAnalogChannel& channel,  
    double& thermistor_measurement);
```

##### Parameters

channel	- The channel on which to measure.
thermistor_	- Reference to a double in which to store the measurement.
measurement	

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/AnalogIO.h

#### 4.2.2.8 RMYoungReading()

This method is used to perform an RMYoung measurement.

```
I2CCODE RMYoungReading(
    UINT32& count,
    UINT32& time);
```

#### Parameters

count	- The count result.
time	- Time of the measured count.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/AnalogIO.h

#### 4.2.2.9 SetConfigurationGain()

This method is used to set the gain configuration component for the indicated channel.

```
void SetConfigurationGain(
    const CAnalogChannel& Channel,
    int Gain);
```

#### Parameters

Channel	- The channel to configure.
Gain	- The gain to set. Valid values are 1 and 16.

#### Return Value

None.

#### Header

Engine/AnalogIO.h



#### 4.2.2.10 SetConfigurationSingleEnded()

This method is used to set the S/D configuration component of the indicated channel to S (single ended).

```
void SetConfigurationSingleEnded(  
    const CAnalogChannel& Channel);
```

##### Parameters

Channel                      - The channel to configure.

##### Return Value

None.

##### Header

Engine/AnalogIO.h

#### 4.2.2.11 SetConfigurationDifferential()

This method is used to set the S/D configuration component of the indicated channel to D (differential).

```
void SetConfigurationDifferential(  
    const CAnalogChannel& Channel);
```

##### Parameters

Channel                      - The channel to configure.

##### Return Value

None.

##### Header

Engine/AnalogIO.h

#### 4.2.2.12 SetConfigurationExcitationHoldOn()

This method is used to set the excitation hold configuration component of the indicated channel to On.

```
I2CCODE SetConfigurationExcitationHoldOn(  
    const CAnalogChannel& Channel);
```

##### Parameters

Channel                      - The channel to configure.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/AnalogIO.h

#### 4.2.2.13 SetConfigurationExcitationHoldOff()

This method is used to set the excitation hold configuration component of the indicated channel to Off.

```
I2CCODE SetConfigurationExcitationHoldOff(  
    const CAnalogChannel& Channel);
```

##### Parameters

Channel                      - The channel to configure.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/AnalogIO.h

#### 4.2.2.14 SetExcitationChannel()

This method is used to set the excitation channel configuration component of the indicated measurement channel.

```
void SetExcitationChannel(  
    const CAnalogChannel& Channel,  
    const CAnalogChannel& ExcitationChannel);
```

##### Parameters

Channel                      - The measurement channel to configure.  
ExcitationChannel           - The channel to use as an excitation channel.

##### Return Value

None.

##### Header

Engine/AnalogIO.h

#### 4.2.2.15 SetExcitationVoltage()

This method is used to set the excitation voltage configuration component of the indicated measurement channel.

```
void SetExcitationVoltage(  
    const CAnalogChannel& Channel,  
    int Voltage);
```

##### Parameters

Channel                      - The measurement channel to configure.  
Voltage                      - The desired excitation voltage. Valid range: -5 to +5.

##### Return Value

None.

Header

Engine/AnalogIO.h

#### 4.2.2.16 SetExcitationVoltageOn()

This method is used to set the excitation voltage On/Off configuration component of the indicated measurement channel to On.

```
void SetExcitationVoltageOn(  
    const CAnalogChannel& Channel);
```

Parameters

Channel                      - The measurement channel to configure.

Return Value

None.

Header

Engine/AnalogIO.h

#### 4.2.2.17 SetExcitationVoltageOff()

This method is used to set the excitation voltage On/Off configuration component of the indicated measurement channel to Off.

```
void SetExcitationVoltageOff(  
    const CAnalogChannel& Channel);
```

Parameters

Channel                      - The measurement channel to configure.

Return Value

None.

Header

Engine/AnalogIO.h

#### 4.2.2.18 SetFilterNotch()

This method is used to set the filter notch configuration component of the indicated channel. Note that whenever the notch is changed, the A/D module must be recalibrated. The Xpert takes care of this recalibration automatically, however, it may take 3-5 seconds for the recalibration to complete. For this reason, don't change the filter notch from the default value unless there is time for this recalibration or it is changed for all the sensors that will be measured.

```
void SetFilterNotch(  
    const CAnalogChannel& Channel,  
    UINT16 FilterNotch);
```

## Parameters

- |             |  |
|-------------|--|
| Channel     | - The channel to configure.                                |
| FilterNotch | - The desired filter notch in Hz. Valid range: 10 to 2000. |

## Return Value

None.

## Header

Engine/AnalogIO.h

### 4.2.2.19 SetWarmUpDelay()

This method is used to set the warm-up delay configuration component of the indicated channel.

```
void SetWarmUpDelay(  
    const CAnalogChannel& Channel,  
    int WarmUpDelay);
```

## Parameters

- |             |  |
|-------------|--|
| Channel     | - The channel to configure.                                  |
| WarmUpDelay | - The desired warm-up delay in ms. Valid range: 0 to 0xffff. |

## Return Value

None.

## Header

Engine/AnalogIO.h

### 4.2.2.20 SetPolyAdjust()

When calibrations are applied to voltage measurements that are close to 0, the result can be less accurate than the raw measurement. This method controls whether the calibration is applied.

```
I2CCODE SetPolyAdjust(  
    const CAnalogChannel& Channel,  
    BOOL poly_adjust);
```

## Parameters

- |             |   |
|-------------|---|
| Channel     | - The channel to configure.   |
| poly_adjust | - Set to TRUE to cause the polynomial calibration to be applied.<br>Set to FALSE to not apply the polynomial calibration. |

## Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

## Header

Engine/AnalogIO.h

#### 4.2.2.21 CmdSetAux1()

This method sets the digital output Aux1 to either high or low.

```
I2CCODE CmdSetAux1(  
    bool High);
```

##### Parameters

**High** - Set to true to cause Aux1 to be set high. Set to false to cause Aux1 to be set low.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/AnalogIO.h

#### 4.2.2.22 CmdPulseOut()

This method causes the digital output Aux1 to pulse either high or low, for an indicated time period, and then revert back to its previous state.

```
I2CCODE CmdPulseOut(  
    bool High,  
    UINT16 PulseWidth_ms);
```

##### Parameters

**High** - Set to true to cause Aux1 to be pulsed high. Set to false to cause Aux1 to be pulsed low.  
**PulseWidth\_ms** - The width of the pulse in ms.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/AnalogIO.h

#### 4.2.2.23 ReadResistance()

This method configures the channel with the specified excitation data and performs a thermistor reading to measure resistance.

```
I2CCODE ReadResistance(  
    const CAnalogChannel& Channel,  
    const CAnalogChannel& nExcitationVoltage,  
    double& ReadData);
```

##### Parameters

**Channel** - The channel on which to measure resistance.

- nExcitationVoltage - The excitation channel.
- ReadData - The resistance result.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/AnalogIO.h

#### 4.2.2.24 ReadFrequency()

This method is used to obtain a frequency measurement from an RMYoung sensor. The analog module has a single channel capable of measuring the output of an RMYoung type wind speed sensor. This type of sensor has a frequency less than 1000 hz and a low level output signal.

```
I2CCODE ReadFrequency(
    int Period,
    bool TakeTwoReadings,
    double& Data);
```

#### Parameters

- Period - When taking two readings, this contains the time in milliseconds between the two readings. Ignored when TakeTwoReadings is false.
- TakeTwoReadings - When true, instructs the method to make two readings with Period amount of time between them. When false, uses the last reading made during the last call to this method as the initial reading.
- Data - The frequency result, in Hz.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/AnalogIO.h

### 4.2.3 DigitalIO

The DigitalIO class contains methods and data specific to Digital I/O Modules. The following sections describe the methods provided by this class.

#### 4.2.3.1 ReadCount()

This method reads the count associated with the indicated channel.

```
I2CCODE ReadCount(
    const CDigitalChannel& Channel,
    UINT32& CountValue);
```

#### Parameters

- |            |   |
|------------|---|
| Channel    | - The channel from which to read the count. |
| CountValue | - The count result.                         |

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/DigIO.h

#### 4.2.3.2 ReadCountAndTime()

This method reads the count associated with the indicated channel, along with the time at which the count was detected. The time is not an absolute time but is relative within each I/O module. The time has units of 1/32768 seconds and can be used to compute frequency (delta counts / delta time).

```
I2CCODE ReadCountAndTime(
    const CDigitalChannel& Channel,
    UINT32& CountValue,
    UINT32& TimeValue);
```

#### Parameters

- |            |  |
|------------|--|
| Channel    | - The channel from which to read the count.    |
| CountValue | - The count result.                            |
| TimeValue  | - The time result in units of 1/32768 seconds. |

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/DigIO.h

#### 4.2.3.3 ReadFilteredInputDataBits()

This method reads the input state of the indicated channel.

```
I2CCODE ReadFilteredInputDataBits(
    const CDigitalChannel& Channel,
    BOOL& Data);
```

#### Parameters

- |         |   |
|---------|---|
| Channel | - The channel to read.                                  |
| Data    | - TRUE if the input is high. FALSE if the input is low. |

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/DigIO.h

#### 4.2.3.4 ReadAllFilteredInputDataBits()

This method reads the input state of all inputs and stores the result as a bitmap within an integer.

```
I2CCODE ReadAllFilteredInputDataBits(  
    int& Data);
```

##### Parameters

- |      |  |
|------|--|
| Data | - An integer representing the input state for all inputs. Test bits using bit mask operations to discover which inputs are high and which are low. |
|------|--|

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/DigIO.h

#### 4.2.3.5 SetSamplingSpeed()

This method is used to set the rate at which the input lines are sampled when the module is running. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
I2CCODE SetSamplingSpeed(  
    double speed_ms);
```

##### Parameters

- |          |  |
|----------|--|
| speed_ms | - The rate in milliseconds at which the input lines should be sampled. Valid range: 0.489 to 1985.9. |
|----------|--|

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/DigIO.h

#### 4.2.3.6 SetLineAsInput()

This method is used to set the indicated channel to an input. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
void SetLineAsInput(  
    const CDigitalChannel& Channel);
```

##### Parameters



Channel - The channel to set as an input.

Return Value

None.

Header

Engine/DigIO.h

#### 4.2.3.7 SetLineAsOutput()

This method is used to set the indicated channel to an output. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
void SetLineAsOutput(  
    const CDigitalChannel& Channel);
```

Parameters

Channel - The channel to set as an output.

Return Value

None.

Header

Engine/DigIO.h

#### 4.2.3.8 SetOutputData()

This method is used to set the indicated channel's output either high or low. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
void SetOutputData(  
    const CDigitalChannel& Channel,  
    bool Value);
```

Parameters

Channel - The channel to set.  
Value - If true, the indicated channel will be set high following the next call to Configure(). If false, the indicated channel will be set low following the next call to Configure().

Return Value

None.

Header

Engine/DigIO.h

#### 4.2.3.9 InvertIO()

This method is used to command the module to invert the indicated channel's input. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
BOOL InvertIO(  
    const CDigitalChannel& Channel);
```

#### Parameters

Channel                      - The channel to invert.

#### Return Value

On success, TRUE is returned, otherwise, FALSE is returned.

#### Header

Engine/DigIO.h

#### 4.2.3.10 UnInvertIO()

This method is used to command the module to uninvert the indicated channel's input. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
BOOL UnInvertIO(  
    const CDigitalChannel& Channel);
```

#### Parameters

Channel                      - The channel to invert.

#### Return Value

On success, TRUE is returned, otherwise, FALSE is returned.

#### Header

Engine/DigIO.h

#### 4.2.3.11 SetAsShaftEncoder()

This method is used to set the indicated channel to act as a shaft encoder input. Always follow this command with a call to Configure() to actually send the new configuration to the module. The shaft encoder input is designed to measure sensors with a quadrature output. The quadrature output uses two channels. The I/O module can measure the quadrature output to track the measurement as it goes up and/or down.

```
void SetAsShaftEncoder(  
    const CDigitalChannel& Channel);
```

#### Parameters

Channel                      - The channel to act as a shaft encoder input.

#### Return Value

None.

#### Header

Engine/DigIO.h

#### 4.2.3.12 SetAsCounter()

This method is used to set the indicated channel to act as a counter input. A counter input is designed for simple switch closure devices such as tipping buckets or frequency type devices such as wind sensors. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
void SetAsCounter(  
    const CDigitalChannel& Channel);
```

##### Parameters

Channel                      - The channel to act as a counter input.

##### Return Value

None.

##### Header

Engine/DigIO.h

#### 4.2.3.13 ConfigureFilters()

This method is used to set the value of the digital filter to the counter associated with the indicated channel. The filter is an up-down counter that counts between 0 and the specified threshold value. The counter will not count up if it is at its upper threshold, and it won't count down when its count is zero. The output of the of the filter only changes state when the counter reaches zero or its upper threshold. Thus, if the output state of the filter is a one, it will stay a one until the counter reaches zero. It will then remain zero until the counter counts up to its upper threshold.

Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
BOOL ConfigureFilters(  
    const CDigitalChannel& Channel,  
    int FilterValue);
```

##### Parameters

Channel                      - The channel to act as a counter input.  
FilterValue                  - The new filter value for the indicated channel. Valid range: 1 to 255.

##### Return Value

On success, TRUE is returned, otherwise, FALSE is returned.

##### Header

Engine/DigIO.h

#### 4.2.3.14 SetSensitivityHigh()

This method is used to set the sensitivity of the indicated RMYoung channel to high. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
BOOL SetSensitivityHigh(  
    const CDigitalChannel& Channel);
```

#### Parameters

Channel - The channel on which to set sensitivity. Must be either 6 or 7.

#### Return Value

On success, TRUE is returned, otherwise, FALSE is returned.

#### Header

Engine/DigIO.h

#### 4.2.3.15 SetSensitivityLow()

This method is used to set the sensitivity of the indicated RMYoung channel to low. Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
BOOL SetSensitivityLow(  
    const CDigitalChannel& Channel);
```

#### Parameters

Channel - The channel on which to set sensitivity. Must be either 6 or 7.

#### Return Value

On success, TRUE is returned, otherwise, FALSE is returned.

#### Header

Engine/DigIO.h

#### 4.2.3.16 AlarmOnSingleEdge()

This method is used to set the alarm associated with the indicated channel to count only a single edge (rising or falling depends on the inversion state). Always follow this command with a call to Configure() to actually send the new configuration to the module.

```
BOOL AlarmOnSingleEdge(  
    const CDigitalChannel& Channel);
```

#### Parameters

Channel - The channel on which to set the alarm edge control.

#### Return Value

On success, TRUE is returned, otherwise, FALSE is returned.

#### Header

Engine/DigIO.h

#### 4.2.3.17 AlarmOnBothEdges()

This method is used to set the alarm associated with the indicated channel to count both edges of a signal edge. Always follow this command with a call to `Configure()` to actually send the new configuration to the module.

```
BOOL AlarmOnBothEdges(  
    const CDigitalChannel& Channel);
```

##### Parameters

Channel                      - The channel on which to set the alarm edge control.

##### Return Value

On success, TRUE is returned, otherwise, FALSE is returned.

##### Header

Engine/DigIO.h

#### 4.2.3.18 Configure()

This method is used to send the current configuration to the module.

```
I2CCODE Configure();
```

##### Parameters

None.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for `IODevice::StartRequest()`.

##### Header

Engine/DigIO.h

#### 4.2.3.19 PulseOut()

This method is used to pulse an output channel, either high or low, for a specified duration. Note that the channel line state returns to that prior to the pulse (e.g., if line was high prior to pulse, and the specified pulse direction is high, then no change in line state would be seen).

The channel must be configured for pulse operation prior to using this function. Call `SetSamplingSpeed()` to set the timer resolution (typically 0.5), call `SetOutputData()` to set the desired pre-pulse state of the line, and call `Configure()` to actually send the line state to the DIO module. The module must be running, so blocks should call `StartRequest()` in `Initialize()`, while non-block programs can call `StartRequest()` followed by `MasterStart()`.

```
I2CCODE PulseOut(  
    const CDigitalChannel& Channel,  
    double TimeMS,  
    bool PulseLow);
```

##### Parameters

- |          |   |
|----------|---|
| Channel  | - The channel to pulse.   |
| TimeMS   | - The number of milliseconds to pulse.  |
| PulseLow | - The direction of the pulse. When true, the channel pulses low, i.e., to ground. |

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/DigIO.h

#### 4.2.3.20 ReadInput()

This method reads and returns the state of the indicated channel.

```
I2CCODE ReadInput(
    const CDigitalChannel& Channel,
    bool Invert,
    bool& Data);
```

#### Parameters

- |         |   |
|---------|---|
| Channel | - The channel to read.  |
| Invert  | - Flag to invert result. If true, the data read from the channel is inverted before returning. If false, no inversion is applied. |
| Data    | - The result of the read and inversion (if applicable).   |

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/DigIO.h

#### 4.2.3.21 ReadFrequency()

This method is used to obtain a frequency measurement from the indicated channel.

```
I2CCODE ReadFrequency(
    const CDigitalChannel& Channel,
    int Period,
    bool TakeTwoReadings,
    double& Data);
```

#### Parameters

- |                 |   |
|-----------------|---|
| Channel         | - The channel on which to measure frequency.  |
| Period          | - When taking two readings, this contains the time in milliseconds between the two readings. Ignored when TakeTwoReadings is false. |
| TakeTwoReadings | - When true, instructs the method to make two readings with Period  |

amount of time between them. When false, uses the last reading made during the last call to this method as the initial reading.

Data                      - The frequency result.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/DigIO.h

### 4.2.4 DisplayIO

The DisplayIO class contains methods and data specific to interacting with the Xlite's front-panel I2C Display (not the GUI display!). The Xlite display has 2 lines of 20 characters each.

Normally, an SLL obtains a pointer to the DisplayIO object representing the display through a callback mechanism. An SLL must contain the following three exports in order for the callback mechanism to work:

```
extern "C" _declspec(dllexport) bool IsMainMenu() {...}
extern "C" _declspec(dllexport) LPCTSTR GetMenuEntry() {...}
extern "C" _declspec(dllexport) bool DisplayProc(DisplayIO* pDisp) {...}
```

**IsMainMenu():** Return true to indicate the sll's exported menu is a main menu, i.e., to be shown instead of the standard menu. If more than one sll defines a new main menu, they are each shown in the order loaded.

**GetMenuEntry():** Return a string to be used as the text displayed for this menu entry in the top-level standard menu. Main menus are still listed as a menu entry in the top-level Xlite menu.

**DisplayProc():** This is the function to be called when the menu is selected. The function should use pDisp to read and write the display. Return true when the standard menu should be shown after this menu is exited. Return false to cause the Xlite display to turn-off after this menu is exited.

See the Samples section for an example of hooking into the Xlite display.

The following sections describe the methods provided by this class.

#### 4.2.4.1 Write()

This method writes a string on the I2C display. When line 0 (the top line) is being written, the bottom line is cleared.

```
void Write(
    LPCTSTR sz,
    int iLine = 0,
    bool bCentered = true);
```

#### Parameters

sz                      - The string to display.  
iLine                   - The number of the line on which to write the string. Valid range: 0 – 1.

- bCentered**
- When “true”, the string written is centered on the display. When “false”, the string is written left justified. This parameter defaults to true.

#### Return Value

None.

#### Header

Engine/ DisplayIO.h

#### 4.2.4.2 WrStringToLCD()

This method writes a string on the I2C display.

```
I2CCODE WrStringToLCD(
    bool bClrB4Wr,
    BYTE byLineNum,
    BYTE byCurPosition,
    LPCTSTR szDispString,
    bool bCenter = true);
```

#### Parameters

- bClrB4Wr**
- When “true”, the display will be cleared before the string is written. When false, the display is not cleared before the string is written.
- byLineNum**
- The number of the line on which to write the string. Valid range: 1 – 2.
- byCurPosition**
- The number of the column on which to write the string. Valid range: 0 – 19.
- szDispString**
- the string to write to display.
- bCenter**
- When “true”, the string written is centered on the display. When “false”, the string is written left justified. This parameter defaults to true.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/ DisplayIO.h

#### 4.2.4.3 DisplayLines()

This method writes strings to both lines of the I2C display.

```
I2CCODE DisplayLines(
    LPCTSTR szLine1,
    LPCTSTR szLine2,
    bool bCenter = true);
```

#### Parameters



- |         |   |
|---------|---|
| szLine1 | - The string to write to line 1 of the display.   |
| szLine2 | - The string to write to line 2 of the display.   |
| bCenter | - When “true”, the strings are written centered on the display.<br>When “false”, the strings are written left justified. This parameter defaults to true. |

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/ DisplayIO.h

#### 4.2.4.4 ShowCursor()

This method shows the cursor at the position specified.

```
I2CCODE ShowCursor(
    BYTE LineNum,
    BYTE CurPosition);
```

#### Parameters

- |             |   |
|-------------|---|
| LineNum     | - The number of the line on which to display the cursor. Valid range: 1 – 2.    |
| CurPosition | - The number of the column on which to display the cursor. Valid range: 0 – 19. |

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/ DisplayIO.h

#### 4.2.4.5 HideCursor()

This method hides the cursor.

```
I2CCODE HideCursor();
```

#### Parameters

None.

#### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

#### Header

Engine/ DisplayIO.h

#### 4.2.4.6 StartBlinkingCursor()

This method blinks the cursor at the position specified.

```
I2CCODE StartBlinkingCursor(  
    BYTE LineNum,  
    BYTE CurPosition);
```

##### Parameters

- |             |   |
|-------------|---|
| LineNum     | - The number of the line on which to blink the cursor. Valid range: 1 – 2.    |
| CurPosition | - The number of the column on which to blink the cursor. Valid range: 0 – 19. |

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/ DisplayIO.h

#### 4.2.4.7 StopBlinkingCursor()

This method stops the cursor from blinking.

```
I2CCODE StopBlinkingCursor();
```

##### Parameters

None.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/ DisplayIO.h

#### 4.2.4.8 ClearDisplay()

This method clears the I2C display.

```
I2CCODE ClearDisplay();
```

##### Parameters

None.

##### Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

##### Header

Engine/DisplayIO.h

#### 4.2.4.9 DisplayOff()

This method turns the I2C display off.

```
I2CCODE DisplayOff();
```

Parameters

None.

Return Value

On success, I2C\_OK is returned. The other possible values for I2CCODE and their meanings are defined in the entry for IODevice::StartRequest().

Header

Engine/ DisplayIO.h

#### 4.2.4.10 KeyPressed()

This method checks to see if a key press from the I2C display is waiting to be processed by a call to Read().

```
bool KeyPressed();
```

Parameters

None.

Return Value

If a key press is waiting, true is returned. Otherwise, false is returned.

Header

Engine/ DisplayIO.h

#### 4.2.4.11 PressKey()

Sends a key to the display, just as if the user had pressed a key.

```
void PressKey(  
    char ch);
```

Parameters

- |    |   |
|----|---|
| ch | - The character to send to the display. The characters representing the standard buttons are '<', '>', and '*'. The standard escape character, used to back out of most display routines, has the value decimal 27. |
|----|---|

Return Value

None.

Header

Engine/ DisplayIO.h

#### 4.2.4.12 Flush()

Deletes any keys in the input buffer that have not yet been processed.

```
void Flush();
```

Parameters

None.

Return Value

None.

Header

Engine/ DisplayIO.h

#### 4.2.4.13 Read()

This method attempts to read a character from the I2C display. If a character is not immediately available, an efficient wait state is entered until either a character arrives, or the specified timeout expires. Note: if a timeout occurs, ResetTimeout() must be called before a call to Read() will succeed.

```
bool Read(  
    char& ch,  
    long lTimeout);
```

Parameters

- |          |   |
|----------|---|
| ch       | - A reference to the variable to contain the character read.          |
| lTimeout | - The number of milliseconds the routine should wait for a character. |

Return Value

If a character is read, true is returned. Otherwise, false is returned.

Header

Engine/ DisplayIO.h

#### 4.2.4.14 EditFloat()

This method manages the editing of a float via the I2C display.

```
EditStatus EditFloat(  
    double& dVal,  
    CString Label,  
    bool bCheck = false,  
    double dLowLim = 0.0,  
    double dHiLim = 0.0);
```

Parameters

dVal	- A reference to the double to edit.
Label	- The text to display as a label, or prompt, as the double is being edited.
bCheck	- When “true”, the editing routine checks the value entered by the user against hi and low limits. When “false”, the value entered by the user is not checked.
dLowLim	- When bCheck is “true”, the editing routine requires the user to enter a value greater than or equal to this low limit. This parameter defaults to 0.0.
dHiLim	- When bCheck is “true”, the editing routine requires the user to enter a value less than or equal to this high limit. This parameter defaults to 0.0.

#### Return Value

On success, EDIT\_OK is returned. If the user cancels the editing process, EDIT\_CANCEL is returned.

#### Header

Engine/ DisplayIO.h

#### 4.2.4.15 EditHEX()

This method manages the editing of a string representing a hexadecimal number via the I2C display.

```
EditStatus EditHex(
    CString& szVal,
    CString Label);
```

#### Parameters

szVal	- A reference to the CString containing the string to edit.
Label	- The text to display as a label, or prompt, as the string is being edited.

#### Return Value

On success, EDIT\_OK is returned. If the user cancels the editing process, EDIT\_CANCEL is returned.

#### Header

Engine/ DisplayIO.h

#### 4.2.4.16 EditInteger()

This method manages the editing of an integer via the I2C display.

```
EditStatus EditInteger(
    int& iVal,
    CString Label,
    bool bCheck = false,
    int iLowLim = 0,
    int iHiLim = 0);
```

## Parameters

- |         |  |
|---------|--|
| iVal    | - A reference to the integer to edit.  |
| Label   | - The text to display as a label, or prompt, as the integer is being edited.   |
| bCheck  | - When “true”, the editing routine checks the value entered by the user against hi and low limits. When “false”, the value entered by the user is not checked. |
| iLowLim | - When bCheck is “true”, the editing routine requires the user to enter a value greater than or equal to this low limit. This parameter defaults to 0.         |
| iHiLim  | - When bCheck is “true”, the editing routine requires the user to enter a value less than or equal to this high limit. This parameter defaults to 0.           |

## Return Value

On success, EDIT\_OK is returned. If the user cancels the editing process, EDIT\_CANCEL is returned.

## Header

Engine/ DisplayIO.h

### 4.2.4.17 EditString()

This method manages the editing of a string via the I2C display.

```
EditStatus EditString(  
    CString& szVal,  
    CString Label,  
    bool bBypassOKCancel = false);
```

## Parameters

- |                 |   |
|-----------------|---|
| szVal           | - A reference to the CString containing the string to edit.   |
| Label           | - The text to display as a label, or prompt, as the string is being edited.   |
| bBypassOKCancel | - When “true”, the edit routine returns with the changed string as soon as the user cursors off the edit area. When “false”, the edit routine prompts for OK/Cancel as usual. This parameter defaults to false. |

## Return Value

On success, EDIT\_OK is returned. If the user cancels the editing process, EDIT\_CANCEL is returned.

## Header

Engine/ DisplayIO.h

#### 4.2.4.18 EditTime()

This method manages the editing of a string representing time via the I2C display.

```
EditStatus EditString(  
    CString& szStr,  
    CString Label);
```

##### Parameters

- |       |   |
|-------|---|
| szStr | - A reference to the CString containing the string to edit.                 |
| Label | - The text to display as a label, or prompt, as the string is being edited. |

##### Return Value

On success, EDIT\_OK is returned. If the user cancels the editing process, EDIT\_CANCEL is returned.

##### Header

Engine/ DisplayIO.h

#### 4.2.4.19 SelectList()

This method displays a list of items from which the user can select. The user iterates through the list using the arrow keys, and makes a selection by pressing the select “\*” key.

```
EditStatus SelectList(  
    LPCTSTR szList[],  
    int iNumItems,  
    int& idx);
```

##### Parameters

- |           |   |
|-----------|---|
| szList    | - An array of strings containing the list items to display.                                   |
| iNumItems | - The number of items in the list.  |
| idx       | - Returns the index of the list item selected. This value will range from 0 to iNumItems – 1. |

##### Return Value

On success, EDIT\_OK is returned and idx contains the index of the item selected. If the list is empty, EDIT\_CANCEL is returned. If a timeout occurs while waiting for the user to select an item, EDIT\_TIMEOUT is returned.

##### Header

Engine/ DisplayIO.h

#### 4.2.4.20 OKCancel()

Prompts the user to select either “OK” or “Cancel” on the bottom line of the display.

```
Bool OKCancel(  
    bool bDefaultOK = true);
```

##### Parameters

**bDefaultOK** - If true, the “OK” text is selected by default. If false, the “Cancel” text is selected by default.

#### Return Value

True is returned if the user selects OK. False is returned if the user selects “Cancel”, or if the request times-out (call `TimedOut()` to determine if a false result occurred due to timeout).

#### Header

Engine/ DisplayIO.h

#### 4.2.4.21 SetTimeout()

This method sets the timeout value used by the DisplayIO API methods when they call `Read()`. The timeout determines how many milliseconds the display will wait for user input.

```
void SetTimeout(  
    long lTimeout);
```

#### Parameters

**lTimeout** - The timeout in milliseconds.

#### Return Value

None.

#### Header

Engine/ DisplayIO.h

#### 4.2.4.22 GetTimeout()

This method returns the timeout value used by DisplayIO API methods to determine how long to wait for user input.

```
long GetTimeout();
```

#### Parameters

None.

#### Return Value

The timeout in milliseconds.

#### Header

Engine/ DisplayIO.h

#### 4.2.4.23 ResetTimeout()

This method resets the timed-out state to not timed-out. Note: this method must be called following a timeout before a call to `Read()` will succeed.

```
void ResetTimeout();
```

#### Parameters



None.

#### Return Value

If a character is read, true is returned. Otherwise, false is returned.

#### Header

Engine/ DisplayIO.h

#### 4.2.4.24 TimedOut()

This method returns the value of the internal timeout state.

```
bool TimedOut();
```

#### Parameters

None.

#### Return Value

If a character read has timed-out (i.e., the user did not respond within the timeout period) with no subsequent call to ResetTimeout(), then true is returned. Otherwise, false is returned.

#### Header

Engine/ DisplayIO.h

#### 4.2.4.25 Error()

This method displays a string and prompts the user to press “\*” to continue.

```
void Error(LPCTSTR szError);
```

#### Parameters

szError                      - The error string to display.

#### Return Value

None.

#### Header

Engine/ DisplayIO.h

### 4.2.5 CIOMod

The CIOMod class is a helper class to the AnalogIO and DigitalIO classes. Its purpose is to contain the methods and data that are common across I/O Modules.

#### 4.2.5.1 GetAnalogPtr()

This method is used to retrieve an analog-compatible pointer to the current module.

```
AnalogIO* GetAnalogPtr();
```

#### Parameters

None.

#### Return Value

An analog-compatible pointer to the current I/O module is returned if the current module is actually of type analog. NULL is returned if the module is not of type analog.

#### Header

Engine/IOMod.h

#### 4.2.5.2 GetDeviceType()

This method is used to retrieve the type of the current device.

```
CIODeviceType GetDeviceType();
```

#### Parameters

None.

#### Return Value

The type of the current module (either CIODeviceType::ANALOG or CIODeviceType::DIGITAL).

#### Header

Engine/IOMod.h

#### 4.2.5.3 GetDigitalPtr()

This method is used to retrieve a digital-compatible pointer to the current module.

```
DigitalIO* GetDigitalPtr();
```

#### Parameters

None.

#### Return Value

A digital-compatible pointer to the current I/O module is returned if the current module is actually of type digital. NULL is returned if the module is not of type digital.

#### Header

Engine/IOMod.h

#### 4.2.5.4 GetModuleNumber()

This method is used to retrieve the one-based id of the current module.

```
int GetModuleNumber();
```

#### Parameters

None.

#### Return Value

The one-based id of the current module.

Header

Engine/IOMod.h

#### 4.2.5.5 GetSerialNo()

This method is used to retrieve the serial number of the current module.

```
int GetSerialNo();
```

Parameters

None.

Return Value

The serial number of the current module.

Header

Engine/IOMod.h

#### 4.2.5.6 SetEventHandler()

This method adds an event handler to the list of event handlers maintained by the current module. The EventExec() method of the TModule registering the event handler should be overridden to perform the event handler function.

```
void SetEventHandler(I2CEVENTHANDLER& EventHandler);
```

Parameters

EventHandler            - A data structure containing a reference to the TModule registering the event handler, and the associated module channel.

Return Value

None.

Header

Engine/IOMod.h

### 4.3 SDI API

The Xpert application framework exports a variable and four functions for use in communicating with SDI-aware devices. These functions and their use is described following:

#### 4.3.1 szSDIAddrSet

This variable contains the set of possible SDI addresses as characters in a constant string. The Xpert SDK template code contains code to use this character array when SDI is selected. The code generated stores the index into this array as a setup block property.

### 4.3.2 SendCmd()

This function sends a command string to the SDI bus and returns after the initial response from the sensor. It is intended for non-measurement commands and interactive user input.

```
SDIerror SendCmd(  
    LPCTSTR szCmd,  
    TCHAR* szReturn,  
    DWORD dwBufferSize,  
    DWORD* pdwReturn);
```

#### Parameters

- |              |   |
|--------------|---|
| szCmd        | - Fully formatted command string to send out the bus. This string should include the address (ex. "aM!", where a is the address). |
| szReturn     | - The buffer in which to store the command response.  |
| dwBufferSize | - Maximum size in bytes of the return buffer.   |
| pdwReturn    | - Pointer to a DWORD in which to store the size of the response.  |

#### Return Value

If the send and response retrieval is successful, then SDI\_OK is returned. Otherwise, an indicator of the error type is returned. See the definition of the enum SDIerror in SDIclass.h for the possible return values and their meaning.

#### Header

SDI/SDIInterface.h

### 4.3.3 CollectData()

This function sends a command string to the SDI bus and waits for a data response. The function does not return until the data is received or a timeout occurs.

```
SDIerror CollectData(  
    LPCTSTR szCmd,  
    double* pdReturn,  
    DWORD dwBufferSize,  
    DWORD* pdwReturn,  
    DWORD dwTimeout,  
    BOOL NoWait);
```

#### Parameters

- |              |  |
|--------------|--|
| szCmd        | - Fully formatted command string to send out the bus. This string should include the address (ex. "aM!", where a is the address).  |
| pdReturn     | - The buffer in which to store the command response.   |
| dwBufferSize | - Maximum size in bytes of the return buffer.  |
| pdwReturn    | Pointer to a DWORD in which to store the size of the response.   |
| dwTimeout    | - Maximum amount of time in milliseconds this function will wait before returning with an error code.  |
| NoWait       | - If TRUE, the bus will be freed after the data request as if the command were concurrent (in order to support sensors that use the "M" command but still run concurrently). |

## Return Value

If the send and data retrieval is successful, then `SDI_OK` is returned. Otherwise, an indicator of the error type is returned. See the definition of the enum `SDIError` in `SDIclass.h` for the possible return values and their meaning.

## Header

`SDI/SDIInterface.h`

### 4.3.4 `SDIAbort()`

This function causes an abort of any in-progress reading initiated by `CollectData` (which may be waiting for the SDI bus to become available).

```
void SDIAbort();
```

## Parameters

None.

## Return Value

None. However, calling this function should cause the in-progress `CollectData` call to return `ABORTED`.

## Header

`SDI/SDIInterface.h`

### 4.3.5 `ClearSDIAbort()`

This function clears the aborted state caused by calling `SDIAbort` so that additional SDI communication attempts can be made.

```
void ClearSDIAbort();
```

## Parameters

None.

## Return Value

None.

## Header

`SDI/SDIInterface.h`

## 4.4 Utilities API

The Utilities API consists of classes and objects used to perform utility type functions, including reporting, power management, user management, and others.

### 4.4.1 Report Management

The Report API displays messages passed to its various messaging methods on the status page. Messages are also output the Xpert terminal port. In addition, the Report API offers hooks allowing the developer to define callback functions called on the occurrence of message types the developer indicates when the callback is registered. The API is accessed through a global instance of the TReport class named “Report”.

#### 4.4.1.1 Debug()

This method is used to submit a debug message to Report. Debug messages are those messages strictly meant to be used by a developer. If the current global filter includes messages of type TMsgLevel::msg\_Debug, then the message will be displayed on the status page and output to the terminal port. Any functions that have been registered as message handlers via Hook(), and have a filter level that includes debug, will be called to process the message.

```
void Debug(  
    LPCTSTR Fmt, ...);
```

##### Parameters

- |     |  |
|-----|--|
| Fmt | - A string containing the format specification for the message. See the Visual Studio help topic “wvsprintf” for a description of the content of the string and the variable arguments that may be passed. |
|-----|--|

##### Return Value

None.

##### Header

Utils/Report.h

#### 4.4.1.2 Warning()

This method is used to submit a warning message to Report. Warning messages are messages indicating that the software detected a potential problem or expected error but can recover gracefully. If the current global filter includes messages of type TMsgLevel::msg\_Warning, then the message will be displayed on the status page and output to the terminal port. Any functions that have been registered as message handlers via Hook(), and have a filter level that includes warning, will be called to process the message.

```
void Warning(  
    LPCTSTR Fmt, ...);
```

##### Parameters

- |     |  |
|-----|--|
| Fmt | - A string containing the format specification for the message. See the Visual Studio help topic “wvsprintf” for a description of the content of the string and the variable arguments that may be passed. |
|-----|--|

##### Return Value

None.

Header

Utils/Report.h

#### 4.4.1.3 Error()

This method is used to submit an error message to Report. Error messages are messages indicating that the software encountered a problem that was unexpected and could not be remedied. The error is not fatal to the operation of the system, however. If the current global filter includes messages of type `TMsgLevel::msg_Error`, then the message will be displayed on the status page and output to the terminal port. Any functions that have been registered as message handlers via `Hook()`, and have a filter level that includes error, will be called to process the message.

```
void Error(  
    LPCTSTR Fmt, ...);
```

Parameters

- |     |  |
|-----|--|
| Fmt | - A string containing the format specification for the message. See the Visual Studio help topic “wvsprintf” for a description of the content of the string and the variable arguments that may be passed. |
|-----|--|

Return Value

None.

Header

Utils/Report.h

#### 4.4.1.4 Fatal()

This method is used to submit a fatal error message to Report. Fatal error messages are messages indicating that the software encountered a problem that was unexpected, could not be remedied, and is likely to be fatal to the system. If the current global filter includes messages of type `TMsgLevel::msg_Fatal`, then the message will be displayed on the status page and output to the terminal port. Any functions that have been registered as message handlers via `Hook()`, and have a filter level that includes fatal, will be called to process the message.

```
void Fatal(  
    LPCTSTR Fmt, ...);
```

Parameters

- |     |  |
|-----|--|
| Fmt | - A string containing the format specification for the message. See the Visual Studio help topic “wvsprintf” for a description of the content of the string and the variable arguments that may be passed. |
|-----|--|

Return Value

None.

## Header

Utils/Report.h

### 4.4.1.5 Status()

This method is used to submit a status message to Report. Status messages are informational messages meant to aid the user in determining that the unit is actually doing something. If the current global filter includes messages of type TMsgLevel::msg\_Status, then the message will be displayed on the status page and output to the terminal port. Any functions that have been registered as message handlers via Hook(), and have a filter level that includes status, will be called to process the message.

```
void Status(  
    LPCTSTR Fmt, ...);
```

#### Parameters

- |     |  |
|-----|--|
| Fmt | - A string containing the format specification for the message. See the Visual Studio help topic “wvsprintf” for a description of the content of the string and the variable arguments that may be passed. |
|-----|--|

#### Return Value

None.

## Header

Utils/Report.h

### 4.4.1.6 Maintenance()

This method is used to submit a maintenance message to Report. Maintenance messages are messages meant to capture the details of a maintenance event performed on the system. If the current global filter includes messages of type TMsgLevel::msg\_Maintenance, then the message will be displayed on the status page and output to the terminal port. Any functions that have been registered as message handlers via Hook(), and have a filter level that includes maintenance, will be called to process the message.

```
void Maintenance(  
    LPCTSTR Fmt, ...);
```

#### Parameters

- |     |  |
|-----|--|
| Fmt | - A string containing the format specification for the message. See the Visual Studio help topic “wvsprintf” for a description of the content of the string and the variable arguments that may be passed. |
|-----|--|

#### Return Value

None.

## Header



Utils/Report.h

#### 4.4.1.7 Note()

This method is used to submit a note message to Report. Note messages are messages meant to capture the details of user site visits. If the current global filter includes messages of type TMsgLevel::msg\_Note, then the message will be displayed on the status page and output to the terminal port. Any functions that have been registered as message handlers via Hook(), and have a filter level that includes note, will be called to process the message.

```
void Note(  
    LPCTSTR Fmt, ...);
```

##### Parameters

- |     |  |
|-----|--|
| Fmt | - A string containing the format specification for the message. See the Visual Studio help topic “wvsprintf” for a description of the content of the string and the variable arguments that may be passed. |
|-----|--|

##### Return Value

None.

##### Header

Utils/Report.h

#### 4.4.1.8 SetFilter()

This method sets the filter level for the function defined by the MsgFunc parameter.

```
void SetFilter(  
    TMsgFunc MsgFunc,  
    int Level);
```

##### Parameters

- |         |   |
|---------|---|
| MsgFunc | - Address of function to which the filter level applies. The function must have been previously registered as a message hook via a call to TReport::Hook(). |
| Level   | - The filter level to apply to the indicated function. The set of possible values is defined by the enum TReport::TMsgLevel in Report.h.                    |

##### Return Value

None.

##### Header

Utils/Report.h

#### 4.4.1.9 GetFilter()

This method returns the filter level of a specific hook or, if no message function is provided, the global filter level that determines which message types are processed.

```
int GetFilter();

int GetFilter(
    TMsgFunc MsgFunc);
```

##### Parameters

- |         |   |
|---------|---|
| MsgFunc | - The address of the function whose filter level should be returned. The function must have been previously registered as a message hook via a call to TReport::Hook(). |
|---------|---|

##### Return Value

If a MsgFunc argument is provided, then the filter level associated with the hook function is returned. If MsgFunc is not provided, this method returns the global filter level that determines which message types are processed (output to the status page and to the terminal port).

##### Header

Utils/Report.h

#### 4.4.1.10 Hook()

This method is used to register a function as a message handler, that is, a function Xpert calls on to handle messages generated through Report. By default, a message handler is registered to handle status, warning, and error messages (see the definition of the enum TMsgLevel for all possible message types). The SetFilter() method can be used to change the types of messages a handler is called-upon to process.

```
void Hook(
    TMsgFunc MsgFunc,
    LPVOID Info);
```

##### Parameters

- |         |   |
|---------|---|
| MsgFunc | - Address of the message handler. An example prototype of the handler follows:<br>void MyHandler(LPVOID Info, int Level, LPCTSTR Msg) |
| Info    | - A pointer that will be passed to the handler whenever it is called upon to handle a message.  |

##### Return Value

If a MsgFunc argument is provided, then the filter level associated with the hook function is returned. If MsgFunc is not provided, this method returns the global filter level that determines which message types are processed (output to the status page and to the terminal port).

##### Header

Utils/Report.h

#### 4.4.1.11 UnHook()

This method is used to stop a function from acting as a message handler. See the description of Hook() for more information on message handlers.

```
void UnHook(  
    TMsgFunc MsgFunc);
```

##### Parameters

MsgFunc                      - Address of the message handler to unhook.

##### Return Value

None.

##### Header

Utils/Report.h

### 4.4.2 User Management

The Xpert maintains a list of users and associated data in a file named “flash disk\users.dat”. This data is manipulated programatically with the help of the classes CUsers and CUser. The class CUsers represents the entire user data set while individual instances of the class CUser are used to contain data specific to single users.

Associated with each user are a name, password, user group, and timeout interval. The user group defines the type of access the user has to the system. The possible values for user group are DATA\_RETRIEVAL\_MODE, INSTALLATION\_MAINTENANCE\_MODE, and SETUP\_MODE. Each of these groups are associated with the similarly named buttons appearing on the Xpert login screen. When users are defined for a particular user group and the associated button is pressed at the login screen, a list of those users is presented for selection.

The timeout interval represents the number of minutes a user can be logged-in without activity before the system goes to sleep.

There are also 10 CUSTOM defined groups CUSTOM\_GROUP1 to CUSTOM\_GROUP10. Custom groups can be created by an SLL with the AddCustomGroup() function and the name of the group will become available to be assigned in the user group setup under the control panel. Both login and command processing can be customized by the Custom SLL. Each Custom SLL to be loaded must have a unique group number. Also see the AddCustomCommandParser() function as well as the CSocketComm class for more information pertaining to creating custom groups.

#### 4.4.2.1 CUsers

The purpose of this class is to provide a convenient mechanism for managing the data contained in the user data file. When an instance of this class is created, the user data file is opened and read. the file is rewritten when the instance is destroyed. The following methods are used to manage the set of users.

##### 4.4.2.1.1 Add()

This method is used to add a user to the user data file.

```

bool Add(
    const CUser& user);

bool Add(
    const CString& strName,
    const CString& strPassword,
    TUserGroup ug,
    unsigned int nTimeout);

```

#### Parameters

- |             |   |
|-------------|---|
| user        | - An instance of CUser containing the user data to add to the data file.  |
| strName     | - The name of the user. The characters used must be from the set available on the Xpert keypad dialog.            |
| strPassword | - The user's password. The characters used must be from the set available on the Xpert keypad dialog.             |
| ug          | - The user's group. Must be one of DATA_RETRIEVAL_MODE (0), INSTALLATION_MAINTENANCE_MODE (1), or SETUP_MODE (2). |
| nTimeout    | - The user's timeout. Valid values: 0, 1, 5, 10, 30, and 60.  |

#### Return Value

When the parameters supplied are valid, the user is added and “true” is returned. If any of the parameters are not valid, then the user is not added and “false” is returned.

#### Header

Utils/Users.h

#### 4.4.2.1.2 Remove()

This method is used to remove a user from the user data file.

```

bool Remove(
    const CString& strUser);

```

#### Parameters

- |         |   |
|---------|---|
| strUser | - The name of the user to remove from the set of users. |
|---------|---|

#### Return Value

If the user is found and removed, “true” is returned. Otherwise, “false” is returned.

#### Header

Utils/Users.h

#### 4.4.2.1.3 GetUser()

This method is used to retrieve the user at a specified index, subject to filtering, if applied. For example, if *i* is set to 3 and no filter is currently applied, then the third user is returned in *u*. If *i* is set to 3 and a filter of data retrieval is applied, then the third user belonging to the data retrieval group is returned in *u*.

```

bool GetUser(
    CUser& u,
    int i);

bool GetUser(
    CUser& u,
    CString strUser);

```

#### Parameters

- |         |  |
|---------|--|
| u       | - A reference to the instance of CUser in which to return the user data, if found. |
| i       | - The index of the user to retrieve, subject to filtering.                         |
| strUser | - The name of the user to retrieve.  |

#### Return Value

If the specified user exists, “true” is returned. Otherwise, “false” is returned.

#### Header

Utils/Users.h

#### 4.4.2.1.4 UpdateUser()

This method is used to update the data associated with a particular user. If the user with the specified name exists, it is updated with the data provided.

```

bool UpdateUser(
    const CString& strUser,
    const CString& strName,
    const CString& strPassword,
    TUserGroup ug,
    unsigned int nTimeout);

```

#### Parameters

- |             |   |
|-------------|---|
| strUser     | - The name of the user to update.   |
| strName     | - The new name to assign to the user. The characters used must be from the set available on the Xpert keypad dialog.                  |
| strPassword | - The password to assign to the user. The characters used must be from the set available on the Xpert keypad dialog.                  |
| ug          | - The user group to assign to the user. Must be one of DATA_RETRIEVAL_MODE (0), INSTALLATION_MAINTENANCE_MODE (1), or SETUP_MODE (2). |
| nTimeout    | - The timeout to assign to the user. Valid values: 0, 1, 5, 10, 30, and 60.   |

#### Return Value

If the user with name strUser exists, “true” is returned. Otherwise, “false” is returned.

#### Header

Utils/Users.h

#### 4.4.2.1.5 GetUserCount()

This method is used to retrieve the number of users currently defined.

```
int GetUserCount();
```

##### Parameters

None.

##### Return Value

Returns the number of users.

##### Header

Utils/Users.h

#### 4.4.2.1.6 SetFilter()

This method is used to set the user group filter to be applied to subsequent calls to GetUser() and GetUserCount().

```
bool SetFilter(  
    TUserGroup ug);
```

##### Parameters

ug                                - The user group to use as a filter.

##### Return Value

If the user group is valid, “true” is returned. Otherwise, “false” is returned.

##### Header

Utils/Users.h

#### 4.4.2.1.7 RemoveFilter()

This method is used to remove the user group filter so that subsequent calls to GetUser() and GetUserCount() operate from the entire user set.

```
void RemoveFilter();
```

##### Parameters

None.

##### Return Value

None.

##### Header

Utils/Users.h

#### 4.4.2.1.8 IsValidUserName()

This method is used to test the validity of the user name provided. A user name is “valid” when it is not empty and all of its characters are selectable from the Xpert keypad dialog.

```
bool IsValidUserName(  
    const CString& str);
```

#### Parameters

str                               - The user name to validate.

#### Return Value

If the string is not empty and all of its characters are selectable from the Xpert keypad dialog, then “true” is returned. Otherwise, “false” is returned.

#### Header

Utils/Users.h

#### 4.4.2.1.9 IsValidPassword()

This method is used to determine if a given string can be used as a password. A string can be used as a password when all of its characters are selectable from the Xpert keypad dialog.

```
bool IsValidPassword(  
    const CString& str);
```

#### Parameters

Str                               - The string to evaluate.

#### Return Value

If all characters in string are selectable from the Xpert keypad dialog, then “true” is returned. Otherwise, “false” is returned.

#### Header

Utils/Users.h

#### 4.4.2.1.10 IsValidUserGroup()

This method is used to determine if the integer provided represents a valid user group.

```
bool IsValidUserGroup(  
    int ug);
```

#### Parameters

ug                               - The integer to evaluate.

#### Return Value

If the integer represents a valid user group, then “true” is returned. Otherwise, “false” is returned.

#### Header

Utils/Users.h

#### 4.4.2.1.11 Commit()

This method is used to commit the set of users in memory to the user data file.

```
bool Commit();
```

##### Parameters

None.

##### Return Value

If the set of users is successfully saved to the user data file, then “true” is returned. Otherwise, “false” is returned.

##### Header

Utils/Users.h

#### 4.4.2.2 CUser

The purpose of this class is to provide a container for the data associated with single users. The following methods are used to operate on instances of this class.

##### 4.4.2.2.1 GetName()

This method is used to retrieve the name of this user.

```
CString GetName();
```

##### Parameters

None.

##### Return Value

Returns the name assigned to this user.

##### Header

Utils/Users.h

##### 4.4.2.2.2 GetPassword()

This method is used to retrieve the password of this user.

```
CString GetPassword();
```

##### Parameters

None.

##### Return Value

Returns the password assigned to this user.

##### Header

Utils/Users.h



#### 4.4.2.2.3 GetUserGroup()

This method is used to retrieve the user group of this user.

```
TUserGroup GetUserGroup();
```

##### Parameters

None.

##### Return Value

Returns the user group assigned to this user, which is an enumeration with the following values: DATA\_RETRIEVAL\_MODE, INSTALLATION\_MAINTENANCE\_MODE, SETUP\_MODE.

##### Header

Utils/Users.h

#### 4.4.2.2.4 GetTimeoutInterval()

This method is used to retrieve the timeout interval of this user.

```
unsigned int GetTimeoutInterval();
```

##### Parameters

None.

##### Return Value

Returns the timeout interval in minutes assigned to this user. Typical values: 0, 1, 5, 10, 30, and 60.

##### Header

Utils/Users.h

#### 4.4.2.3 AddCustomGroup()

This method is used to retrieve the timeout interval of this user.

```
bool AddCustomGroup(  
    TUserGroup Group,  
    LPCTSTR Description,  
    LPFNCUSTOMLOGIN LoginFunc);
```

##### Parameters

- |             |  |
|-------------|--|
| Group       | - A custom group to define. CUSTOM_GROUP1 to CUSTOM_GROUP10.   |
| Description | - Name that will appear in the User control panel setup screen.  |
| LoginFunc   | - A function that will be called whenever a user logs in under a username belonging to the specified custom group. This only applies for logins via the serial port, and not via the LCD display.<br><br>A custom login function might output a report to the user using |

the CSocketComm class API, or it might install a setup a list of custom commands to enhance Remote's builtin functionality.

A custom login function looks like this:

```
bool CustomLoginFunction(TUserGroup Group, LPCTSTR
ComPort, LPCTSTR UserName)
```

The CustomLoginFunction should return 'false' if it has completely handled the user interaction and Remote should hangup. If instead 'true' is returned, Remote will continue on from the login stage to the command prompt.

The login function should complete its work in under 1 minute to avoid a timeout from occurring. If the function requires more than 1 minute, it should spawn a thread to perform the necessary work, and return 'true' immediately.

#### Return Value

Returns true if the group wasn't already defined.

#### Header

Utils/Users.h

#### 4.4.2.4 AddCustomCommandParser()

This method is used to identify a function to be called whenever the user issues a custom command to Remote's command prompt.

```
bool AddCustomCommandParser(
    TUserGroup Group,
    LPFNCUSTOMCOMMANDPARSER CommandParser);
```

Typically a custom comand parser will open the serial port using the CSocketComm class, parse the custom command, and output a message.

A custom command parser function looks like this:

```
bool CustomCommandParser(TUserGroup Group, LPCTSTR ComPort, LPCTSTR UserName,
LPCTSTR Command)
```

The CustomCommandParser should return 'true' if it has completely processed the command and Remote should emit a new command prompt. If instead 'false' is returned then Remote performs no additional output, but does wait for a new command.

The parser should complete it's work in under 1 minute to avoid a timeout from occurring. If the parser requires more then 1 minute, it should spawn a thread to perform the necessary work, and return immediately.

#### Parameters

- Group - A custom group to define. CUSTOM\_GROUP1 to CUSTOM\_GROUP10.

CommandParser - A function which will be called whenever the user enters a custom command in to Remote's command prompt.

Return Value

Returns true if the group wasn't already defined.

Header

Utils/Users.h

### 4.4.3 Serial Communications

The Xpert SDK provides a serial communications API via the class "CSerialComm." This class wraps standard WIN32 serial API functions to simplify the task of communicating over Xpert's serial ports.

#### 4.4.3.1 CSerialComm()

This is the default constructor for the class. The default settings are as follows: COM1, 115200 baud, 8 bits per byte, no parity, one stop bit, and hardware flow control.

```
CSerialComm();
```

Parameters

None.

Return Value

None.

Header

Utils/SerialCommClass.h

#### 4.4.3.2 CSerialComm()

This is an optional constructor for the class allowing the port to be passed in.

```
CSerialComm(const TCHAR* CommPort);
```

Parameters

CommPort - A string identifying the port to use, e.g., "COM1:", "COM2:", etc. Note the use of the colon. Xpert currently supports COM1 through COM5.

Return Value

None.

Header

Utils/SerialCommClass.h

#### 4.4.3.3 OpenComm()

Tries to open the com port. This method must be called before sending or receiving data over the port.

```
bool OpenComm();
```

Parameters

None.

Return Value

Returns true if the port could be opened.

Header

Utils/SerialCommClass.h

#### 4.4.3.4 CloseComm()

Closes the com port. This is performed automatically by the destructor.

```
bool CloseComm();
```

Parameters

None.

Return Value

Returns true if the port was closed, or false if it wasn't open to begin with.

Header

Utils/SerialCommClass.h

#### 4.4.3.5 IsOpen()

Tests whether the port is currently opened.

```
bool IsOpen();
```

Parameters

None.

Return Value

Returns true if the port is open.

Header

Utils/SerialCommClass.h

#### 4.4.3.6 SetConfiguration()

Sets port configuration parameters.

```
void SetConfiguration(
    DWORD BaudRate,
    BYTE ByteSize,
    BYTE Parity,
    BYTE StopBits,
    BOOL FlowCtrl);
```

#### Parameters

- |          |  |
|----------|--|
| BaudRate | - Specifies the baud rate at which the communication device operates. It can be set to an actual baud rate value, or to one of the following constants: CBR_110, CBR_300, CBR_600, CBR_1200, CBR_2400, CBR_4800, CBR_9600, CBR_14400, CBR_19200, CBR_38400, CBR_56000, CBR_57600, CBR_115200, CBR_128000, or CBR_256000. |
| ByteSize | - Specifies the number of bits in the bytes transmitted and received. Typically 7, 8, or 9.  |
| Parity   | - Specifies the parity scheme to be used. Set to one of the following constant values: EVENPARITY, MARKPARITY, NOPARITY, ODDPARITY, or SPACEPARITY.  |
| StopBits | - Specifies the number of stop bits to be used. Set to one of the following constant values: ONESTOPBIT, ONE5STOPBITS, or TWOSTOPBITS.   |
| FlowCtrl | - Specifies whether to use hardware flow control.  |

#### Return Value

None.

#### Header

Utils/SerialCommClass.h

#### 4.4.3.7 SetCommPort()

Sets the CommPort to opened by OpenComm().

```
void SetCommPort(const TCHAR* CommPort);
```

#### Parameters

- |          |   |
|----------|---|
| CommPort | - A string identifying the port to use, e.g., "COM1:", "COM2:", etc. Note the use of the colon. |
|----------|---|

#### Return Value

None.

#### Header

Utils/SerialCommClass.h

#### 4.4.3.8 SetBaudRate()

Sets the baud rate of the port.

```
void SetBaudRate(const DWORD BaudRate);
```

#### Parameters

- BaudRate** - Specifies the baud rate at which the communication device operates. It can be set to an actual baud rate value, or to one of the following constants: CBR\_110, CBR\_300, CBR\_600, CBR\_1200, CBR\_2400, CBR\_4800, CBR\_9600, CBR\_14400, CBR\_19200, CBR\_38400, CBR\_56000, CBR\_57600, CBR\_115200, CBR\_128000, or CBR\_256000.

#### Return Value

None.

#### Header

Utils/SerialCommClass.h

#### 4.4.3.9 SetTimeouts()

Sets the timeout values for reading and writing the port.

```
void SetTimeouts(COMMTIMEOUTS& CommTimeouts);
```

#### Parameters

- CommTimeouts** - This structure is defined in the Win32 API. See the Visual Studio help topic “COMMTIMEOUTS” for a description of the content and purpose of this structure.

#### Return Value

None.

#### Header

Utils/SerialCommClass.h

#### 4.4.3.10 Input Functions

The following functions all read input from the com port allowing various data types to be read.

```
bool GetChar(char * ch);
```

```
bool GetChar(UCHAR * uch);
```

```
bool GetChar(TCHAR * wch);
```

```
int GetStr(UCHAR * Str, int StrLen);
```

```
int GetStr(char * Str, int StrLen);
```

```
int GetStr(TCHAR * Str, int StrLen);
```

```
int Get(void* Buffer, int BufferLen);
```

## Parameters

ch, uch, wch	- A single character of data.
Str	- A string of data.
StrLen	- Number of characters to request.
Buffer	- A raw binary buffer.
BufferLen	- Number of raw bytes to request.

## Return Value

Character functions return *true* if the operation succeeded. String functions return the number of characters processed.

## Header

Utils/SerialCommClass.h

### 4.4.3.11 Output Functions

The following functions all output data to the com port allowing various data types to be sent. The “AndWait()” method does not return until after all data has been sent.

```
bool Put(void* Buffer, int BufferLen);

bool PutChar(const char ch);

bool PutChar(const UCHAR uch);

bool PutChar(const TCHAR wch);

bool PutStr(const char * Str);

bool PutStr(const char * Str, int StrLen);

bool PutStr(const UCHAR * Str);

bool PutStr(const UCHAR * Str, int StrLen);

bool PutStr(TCHAR * Str);

bool PutStr(TCHAR * Str, int StrLen);

bool PutStr(CString Str);

bool PutAndWait(void* Buffer, int BufferLen);
```

## Parameters

ch, uch, wch	- A single character of data.
Str	- A string of data.
StrLen	- Number of characters to send
Buffer	- A raw binary buffer
BufferLen	- Number of raw bytes to send

## Return Value

True if the operation succeeded.

Header

Utils/SerialCommClass.h

#### 4.4.3.12 NumberBytesInputBuffer()

Returns the number of bytes in the input buffer.

```
int NumberBytesInputBuffer();
```

Parameters

None.

Return Value

Number of bytes in the input buffer.

Header

Utils/SerialCommClass.h

#### 4.4.3.13 KeyPressed()

Returns true if there is at least one byte in the input buffer.

```
bool KeyPressed();
```

Parameters

None.

Return Value

True/false.

Header

Utils/SerialCommClass.h

#### 4.4.3.14 FlushInput ()

Removes all bytes in the input buffer.

```
void FlushInput();
```

Parameters

None.

Return Value

None.

Header

Utils/SerialCommClass.h



#### 4.4.3.15 GetHandle()

Retrieves the internal HANDLE used for performing Win32 API-level communications.

```
HANDLE GetHandle();
```

##### Parameters

None.

##### Return Value

Handle of the comm port.

##### Header

Utils/SerialCommClass.h

#### 4.4.3.16 WaitOnRx()

Waits for data to be received. Note that this function clears the input buffer before waiting.

```
void WaitOnRx();
```

##### Parameters

None.

##### Return Value

None.

##### Header

Utils/SerialCommClass.h

#### 4.4.3.17 WaitForTxEmpty ()

Waits for the serial port to finish sending data.

```
void WaitForTxEmpty();
```

##### Parameters

None.

##### Return Value

None.

##### Header

Utils/SerialCommClass.h

### 4.4.4 Remote Communications

The Xpert has an API which allows communication with any of the communication ports managed by the Remote program called SocketCommClass. The API uses Socket communication to send I/O operations over to Remote.exe, which then carries out the requests. Ports can be completely taken over from Remote, or a user SLL can work with Remote to enhance the standard services.

Most of the functions in the API will fail if the connection with the remote socket is closed. One simple way this can occur is if the Remote.exe application is terminated by the user.

Here's a list of some of the things that can be done with the API:

- Monitor any COM port controlled by Remote.
- Take over full or partial control of any COM port controlled by Remote.
- Add new command line options in conjunction with functions in Users.dll
- Add custom login screens, again in conjunction with functions in Users.dll

Each port behaves differently depending on how the port was configured on Remote's command line during startup.

<b>Port Type</b>	<b>Description</b>
COM:	Standard interactive direct connect port for connecting an Xpert to a notebook or desktop computer. Supports login, command line, and SSP. SSP is automatically detected and processed. May also be used to connect to a smart radio or smart modem that for looks like a direct connect and transparently manages connection.
SSP:	Used to interface an Xpert directly to another Xpert or to a base station. Supports only SSP communication. May also be used to connect to a smart radio or smart modem that looks like a direct connect and transparently manages connection.
RADIO:	Used to interface the Xpert to a "dumb" half-duplex multi-drop UHF/VHF radio. Supports only SSP communication. Uses CSMA to prevent collisions.
MODEM:	Used to interface the Xpert to Hayes compatible telephone modem. No provisions are made for power-saving. The modem is expected to be pre-configured such that CD is raised when someone dials in, and so that toggling DTR causes a hangup.
VOICE:	Used to interface to the Sutron Voice modem. This modem has special circuitry that will raise CD and DSR when the phone rings allowing the Xpert to raise DTR to fully power up the modem circuitry. However because the modem is always fully powered down between calls, the only settings it ever remembers are the ones programmed in to it's non-volatile memory.
RS:	Used to interface two Xperts together over a

	multi-drop RS-485 connection. Supports only SSP communication.
--	--

Here are some tables that explain the differences between how the various RS-232 control lines are managed:

<b>Port</b>	<b><u>Effect of DTR</u></b>
COM:	Raised on Connect (DSR or RX Data), Dropped on logout.
SSP:	when transmitting, Dropped after.
RADIO:	Raised when transmitting, followed by a 1sec carrier delay, dropped
MODEM:	Raised normally to allow the modem to operate. Toggled to force a hangup.
VOICE:	Raised to power on the Sutron Speech Modem. Dropped to hangup, or power it down.
RS:	Normally raised to allow RS485 reception, lowered when transmitting.

<b>Port</b>	<b><u>Effect of RTS</u></b>
COM:	H/W handshaking
SSP:	H/W handshaking
RADIO:	Raised to key the radio transmitter, followed by a 1sec carrier delay, then dropped. (Same as DTR)
MODEM:	H/W handshaking
VOICE:	H/W handshaking
RS:	Raised to perform an RS485 transmission, lowered afterwards.

<b>Port</b>	<b><u>Effect of DSR</u></b>
COM:	May be used to wake-up Remote and start a session. Drop it to end the session. Simply simply sending data can also be used start a session.
SSP:	May be used to wake-up Remote and start a session. Drop it to end the session. Simply simply sending data can also be used start a session.
RADIO:	Not used.
MODEM:	Not used.
VOICE:	Not used.
RS:	Not used.

<b>Port</b>	<b><u>Effect of CD (RLSD)</u></b>
-------------	-----------------------------------

COM:	Not used.
SSP:	Not used.
RADIO:	Goes high when a message is being received. A message will not be transmitted while CD is high (this is called CSMA). The Xpert is capable of receiving a message even if CD is low, but it won't be able to prevent collisions on transmit.
MODEM:	Going high indicates the beginning of a session, going low indicates the end.
VOICE:	Going high indicates the beginning of a session, going low indicates the end.
RS:	Not used.

<b>Port</b>	<b>Effect of CTS</b>
COM:	H/W handshaking
SSP:	H/W handshaking
RADIO:	Not used.
MODEM:	H/W handshaking
VOICE:	H/W handshaking
RS:	Not used.

#### 4.4.4.1 Remote SSP Operations

These functions may be called directly to perform SSP operations on the selected COM port. Remote manages the lower levels of the protocol directly and the user need only supply the necessary data structures.

##### 4.4.4.1.1 RemoteRequest()

Send SSP data and wait for an appropriate response. Can be used on a port that has been opened using CSocketComm Class and locked using CSocketComm::LockComm(), but be sure to call CSocketComm::SetComOptions() to enable the flags: ENABLESSP and ENABLESSPPARSER or otherwise reply messages won't be processed. This situation occurs when custom code is needed to create a connection before SSP messaging can occur.

```
int RemoteRequest(int PortIndex, CString Send_To, CString Send_From, DWORD
Send_SeqNum, TSSPData& Send_Data, int Count, OPMATCH Match[], CString&
Reply_To, CString& Reply_From, DWORD& Reply_SeqNum, TSSPData& Reply_Data, int
Retries, int AckDelay);
```

#### Parameters

- |           |   |
|-----------|---|
| PortIndex | - Port number to send the message out. 1 for COM1, 2 for COM2, 3 for COM3, etc. The port selected must be a port controlled by Remote.exe |
| Send_To   | - SSP Unit ID to send the message to. "*" may be selected to indicate that all stations receive the message.                              |

Send_From	- SSP Unit ID that the message is from. Typically Engine.StationName is used here.
Send_SeqNum	- Every SSP Message has a sequence number used to identify duplicate messages (retransmissions). Normally GetFlagSeq() is called to create a correct and varying sequence number.
Send_Data	- An SSP Data packet containing one or more opcodes and associated data. See the SSP Technical Reference for more information about the data format of various SSP opcodes.
Count	- The number of elements in the Match array (see following).
Match	<p>- An array of opcodes to expect in response to the message. Each element in the array contains a sequence number to match (typically 0 to match any sequence) and opcode to match (for instance an OpAck or OpNak is often sent in reply to a message), and a piece of the data may be matched as well. If the data match value is 0, then the field is ignored. The opcode being acked is typically the “data” in an ack or nak message so this is typically passed in the data match field as follows:</p> <p>Ex. OPMATCH Matches[2] = {OPMATCH(0, OpAck, OpAlarm), OPMATCH(0, OpNak, OpAlarm)}</p> <p>Only replies that match an item in this array will be returned, others will be processed in the default manner by Remote.</p>
Reply_To	- The address the reply message was sent to, which should match the Send_From field.
Reply_From	- The source of the reply message, which will normally match the Send_To field (when a message is sent to “*” the Reply_From field will have the unit ID of the station which responded).
Reply_SeqNum	- The sequence number of the reply message.
Reply_Data	- The SSP message itself that was sent in reply.
Retries	- Number of times to retry sending the message if a response isn’t received.
AckDelay	- How long remote should wait for a response before attempting a retry (ms).

#### Return Value

- 3 could not contact remote
- 2 remote could not perform the operation
- 1 timeout waiting for a response
- 0-(Count-1) index of matching message received

#### Header

Engine/RemoteOps.h

#### 4.4.4.1.2 RemoteSend()

Send SSP data and waits for it to go out.

```
bool RemoteSend(int PortIndex, CString Send_To, CString Send_From, DWORD  
Send_SeqNum, TSSPData& Send_Data, int SendDelay);
```

##### Parameters

- |             |   |
|-------------|---|
| PortIndex   | - Port number to send the message out. 1 for COM1, 2 for COM2, 3 for COM3, etc. The port selected must be a port controlled by Remote.exe.  |
| Send_To     | - SSP Unit ID to send the message to. "*" may be selected to indicate that all stations receive the message.  |
| Send_From   | - SSP Unit ID that the message is from. Typically Engine.StationName is used here.  |
| Send_SeqNum | - Every SSP Message has a sequence number used to identify duplicate messages (retransmissions). Normally GetFlagSeq() is called to create a correct and varying sequence number.   |
| Send_Data   | - An SSP Data packet containing one or more opcodes and associated data. See the SSP Technical Reference for more information about the data format of various SSP opcodes.   |
| SendDelay   | - How long to wait for Remote to send the message in (ms). The time it takes to complete sending a message depends on the baud rate, the message length, and how busy the port is with sending or receiving other messages. |

##### Return Value

True if Remote was able to send the message.

##### Header

Engine/RemoteOps.h

#### 4.4.4.1.3 RemoteWaitMessage()

Waits for an SSP Message to be received. Can be used on a port that has been opened using CSocketComm Class and locked using CSocketComm::LockComm(), but be sure to call CSocketComm::SetComOptions() to enable the flags: ENABLESSP and ENABLESSPPARSER or otherwise reply messages won't be processed. This situation occurs when custom code is needed to create a connection before SSP messaging can occur.

```
int RemoteWaitMessage(int PortIndex, CString Wait_To, CString Wait_From, int  
Count, OPMATCH Match[], CString& Reply_To, CString& Reply_From, DWORD&  
Reply_SeqNum, TSSPData& Reply_Data, int Reply_Delay);
```

##### Parameters

PortIndex	- Port number to send the message out. 1 for COM1, 2 for COM2, 3 for COM3, etc. The port selected must be a port controlled by Remote.exe.
Wait_To	- SSP Unit ID of a station to wait for a message address to. Use "*" to select a message addressed to any station.
Wait_From	- SSP Unit ID of a station to wait for a message from. Use "*".
Count	- The number of elements in the Match array (see following).
Match	<p>- An array of opcodes used to filter out and select a specific message to return. Each element in the array contains a sequence number to match (typically 0 to match any sequence) and opcode to match (for instance an OpAck or OpNak is often sent in reply to a message), and a piece of the data may be matched as well. If the data match value is 0, then the field is ignored. The opcode being acked is typically the "data" in an ack or nak message so this is typically passed in the data match field as follows:</p> <p>Ex. OPMATCH Matches[2] = {OPMATCH(0, CurDataReq, 0), OPMATCH(0, OpAlarm, 0)}</p> <p>Only messages that match an item in this array will be returned, others will be processed in the default manner by Remote.</p> <p>OpEOT may be specified (ex. OPMATCH(0, OpEOT, 0)) to specify that any message is acceptable.</p>
Reply_To	- The address the received message was sent to, which should match the Send_From field.
Reply_From	- The source of the received message, which will normally match the Send_To field (when a message is sent to "*" the Reply_From field will have the unit ID of the station which responded).
Reply_SeqNum	- The sequence number of the message received.
Reply_Data	- The SSP message itself that was received.
Reply_Delay	- How long remote should wait for a message before returnin with a timeout error (ms).

#### Return Value

- 3 could not contact remote
- 2 remote could not perform the operation
- 1 timeout waiting for a response
- 0-(Count-1) index of matching message received

#### Header

Engine/RemoteOps.h

#### 4.4.4.2 CSocketComm Class

Create an instance of this class if you wish to take over or monitor a serial port controlled by Remote.

##### 4.4.4.2.1 CSocketComm()

This is the default constructor for the class.

```
CSocketComm();
```

#### Parameters

None.

#### Return Value

None.

#### Header

Utils/SocketCommClass.h

##### 4.4.4.2.2 CSocketComm(const TCHAR\* CommPort)

This is an optional constructor for the class allowing the port to be passed in.

```
CSocketComm(const TCHAR* CommPort);
```

#### Parameters

CommPort                      - The port to access. The Xpert currently supports COM1:, COM2:, COM3:, COM4: and COM5:.

#### Return Value

None.

#### Header

Utils/SocketCommClass.h

##### 4.4.4.2.3 OpenComm()

Tries to open the com port, if successful automatic capture of incoming data will be enabled, but Remote will still be in control of the port.

```
bool OpenComm();
```

#### Parameters

None.

#### Return Value

Returns true if the port could be opened.



## Header

Utils/SocketCommClass.h

### 4.4.4.2.4 CloseComm()

Closes the com port, terminating the socket session with Remote if it was still open. This is performed automatically by the destructor.

```
bool CloseComm();
```

#### Parameters

None.

#### Return Value

Returns true if the port was closed, or false if it wasn't open to begin with.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.5 IsOpen()

Tests whether the port is currently opened.

```
bool IsOpen();
```

#### Parameters

None.

#### Return Value

Returns true if the port is open.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.6 SetConfiguration()

Sets port configuration parameters. (FUTURE USE)

```
void SetConfiguration(DWORD BaudRate, BYTE ByteSize, BYTE Parity, BYTE StopBits, BOOL FlowCtrl);
```

#### Parameters

BaudRate	- All parameters are currently ignored, the port settings are determined by Remote. The default is 115200,n,8,1 with h/w flow control.
ByteSize	
Parity	
StopBits	
FlowCtrl	

## Return Value

None.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.7 SetCommPort()

Sets the CommPort to opened by OpenComm().

```
void SetCommPort(const TCHAR* CommPort);
```

## Parameters

CommPort                    - The port to use.

## Return Value

None.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.8 SetBaudRate()

Sets the baud rate of the port. (FOR FUTURE USE)

```
void SetBaudRate(const DWORD BaudRate);
```

## Parameters

BaudRate                    - The rate to use. Default is 115200.

## Return Value

None.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.9 SetTimeouts() and SetCommTimeouts()

Sets the timeout values for reading and writing the port. SetCommTimeouts is the same as SetTimeouts and exists for easier porting of Win32 code.

```
void SetTimeouts(COMMTIMEOUTS& CommTimeouts);
```

```
BOOL SetCommTimeouts(LPCOMMTIMEOUTS lpCommTimeouts);
```

## Parameters

CommTimeouts                - This structure is defined in the Win32 API. Its use is not an exact match for Socket communications, but it's close. The main difference is that socket communication is done by sending blocks of data at a time, and not single bytes. Additionally, since

all communication with the port is indirect via Remote, additional time should be allowed for data to be sent or received.

#### Return Value

None.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.10 INPUT FUNCTIONS

The following functions all read input from the remote com port allowing various data types to be read.

```
bool GetChar(char * ch);  
  
bool GetChar(UCHAR * uch);  
  
bool GetChar(TCHAR * wch);  
  
int  GetStr(UCHAR * Str, int StrLen);  
  
int  GetStr(char * Str, int StrLen);  
  
int  GetStr(TCHAR * Str, int StrLen);  
  
int  Get(void* Buffer, int BufferLen);  
  
bool Read(TSSPPacket& Packet, DWORD Timeout);
```

#### Parameters

ch, uch, wch	- A single character of data.
Str	- A string of data.
StrLen	- Number of characters to request
Buffer	- A raw binary buffer
BufferLen	- Number of raw bytes to request
Packet	- An SSP Packet
Timeout	- Timeout value in milli-seconds

#### Return Value

True if the operation succeeded.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.11 OUTPUT FUNCTIONS

The following functions all output data to the remote com port allowing various data types to be sent. The “AndWait()” methods do not return until after Remote has sent all the bytes, and they have left the UART fifos.

```

bool Put(void* Buffer, int BufferLen);

bool PutChar(const char ch);

bool PutChar(const UCHAR uch);

bool PutChar(const TCHAR wch);

bool PutStr(const char * Str) {return PutStr((UCHAR*)Str);};

bool PutStr(const char * Str, int StrLen) {return PutStr((UCHAR*)Str,
StrLen);};

bool PutStrAndWait(const char * Str);

bool PutStrAndWait(const char * Str, int StrLen);

bool PutStr(const UCHAR * Str) {return (PutStr(Str, strlen((char*)Str)));};

bool PutStr(const UCHAR * Str, int StrLen);

bool PutStr(TCHAR * Str);

bool PutStr(TCHAR * Str, int StrLen);

bool PutStr(CString Str);

bool PutAndWait(void* Buffer, int BufferLen);

bool PutStrAndWait(const UCHAR * Str);

bool PutStrAndWait(const UCHAR * Str, int StrLen);

bool PutStrAndWait(TCHAR * Str);

bool PutStrAndWait(TCHAR * Str, int StrLen);

bool PutStrAndWait(CString Str);

bool Write(LPVOID lpBuffer, DWORD nNumberOfBytesToRead);

bool WriteAndWait(LPVOID lpBuffer, DWORD nNumberOfBytesToWrite);

bool Write(TSSPPacket& Packet);

bool WriteAndWait(TSSPPacket& Packet);

```

#### Parameters

ch, uch, wch	- A single character of data.
Str	- A string of data.
StrLen	- Number of characters to send
Buffer	- A raw binary buffer
BufferLen	- Number of raw bytes to send
lpBuffer	- A raw binary buffer

nNumberOfBytesT - Number of raw bytes to send.  
oWrite  
Packet - An SSP Packet

#### Return Value

True if the operation succeeded.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.12 WIN32 COMM COMPATIBILITY FUNCTIONS

The following functions all are socket equivalents of the Win32 Comm functions. Basically the Win32 parameters are packetized, sent over to Remote, executed directly on the Com port, and any result is passed back. WaitCommEvent() and Read() are slightly different then their Win32 counterparts in that they support a timeout value (milliseconds).

```
void EscapeCommFunction(DWORD Func);  
  
bool GetCommModemStatus(LPDWORD lpModemStat);  
  
BOOL SetCommMask(DWORD dwEvtMask);  
  
BOOL WaitCommEvent(LPDWORD lpEvtMask, DWORD Timeout = INFINITE);  
  
BOOL GetCommMask(LPDWORD lpEvtMask);  
  
BOOL GetCommState(LPDCB lpDCB);  
  
BOOL SetCommState(LPDCB lpDCB);  
  
BOOL GetCommTimeouts(LPCOMMTIMEOUTS lpCommTimeouts);  
  
bool Read(LPVOID lpBuffer, DWORD nNumberOfBytesToRead, LPDWORD  
lpNumberOfBytesRead, DWORD Timeout);
```

#### Parameters

See Win32 Documentation.

#### Return Value

See Win32 Documentation.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.13 NumberBytesInputBuffer()

Returns the number of bytes available to be read. Capture must be enabled.

```
int NumberBytesInputBuffer();
```

#### Parameters

None.

#### Return Value

Number of bytes.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.14 KeyPressed()

Returns true if there is at least one byte available to be read. Capture must be enabled.

```
bool KeyPressed();
```

#### Parameters

None.

#### Return Value

True/false.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.15 FlushInput ()

Removes all bytes in the receiver buffer.

```
void FlushInput();
```

#### Parameters

None.

#### Return Value

None.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.16 GetClient ()

Retrieves the internal SocketClient class used for performing the socket level communications.

```
TSocketClient& GetClient();
```

#### Parameters

None.

#### Return Value

A reference to the Client object used for socket communications with remote.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.17 WaitOnRx()

Waits for incoming data to arrive.

```
void WaitOnRx();
```

#### Parameters

None.

#### Return Value

None.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.18 WaitForTxEmpty ()

Waits for the remote serial port to finish sending data.

```
void WaitForTxEmpty ();
```

#### Parameters

None.

#### Return Value

None.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.19 SetCapture()

Enables/disables capturing of incoming serial data and state information.

```
bool SetCapture(bool State);
```

#### Parameters

State                      - True to enable capture

#### Return Value

Result of the operation.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.20 LockComm()

Attempts to gain exclusive access to the serial port. If successful, Remote hands over operation of the port. If at anytime the connection is broken or an inactivity timeout occurs, remote will break the connection and take back control. SSP processing and parsing is by disabled by default when a port is locked. They can be enabled using SetComOptions(), this is necessary if RemoteRequest(), RemoteWaitMessage(), or RemoteSend() are to be used.

```
bool LockComm(DWORD Timeout);
```

##### Parameters

Timeout                      - How long to wait for access (ms)

##### Return Value

Result of the operation.

##### Header

Utils/SocketCommClass.h

#### 4.4.4.2.21 UnLockComm()

Returns control of the port back to Remote. If NoHangup is false (the default), then remote will hangup and terminate the conversation, otherwise if the connection is still live, Remote will activate it's command line and SSP functions. For instance, to perform voice AND data answering of the telephone, the Coms application first takes full control of the port with the LockComm() function. If the phone rings, it answers in voice mode. But if the user is actually dialing in over a modem, it switches to answering in data mode, then releases the port back to Remote with UnLockComm(true).

```
bool UnLockComm(bool NoHangup=false);
```

##### Parameters

NoHangup                    - Normally false causing Remote to hangup and terminate an active conversation.

##### Return Value

Result of the operation.

##### Header

Utils/SocketCommClass.h

#### 4.4.4.2.22 Logout()

Instructs remote to Logout a user (go back to the login prompt), and optionally hangup on the user.

```
bool Logout(bool NoHangup=false);
```

##### Parameters



NoHangup - Normally false causing Remote to hangup and terminate an active conversation.

#### Return Value

Result of the operation.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.23 SetHost()

By default OpenComm() will open a session with the copy of Remote.exe running on the local machine (ie localhost), if networking is available SetHost() can be used to connect to a copy of Remote running somewhere else on the internet or network. Call SetHost() before calling OpenComm().

```
void SetHost(LPCTSTR NewHost);
```

#### Parameters

NewHost - A URL or IP address in string form, the default is \_T("localhost")

#### Return Value

Result of the operation.

#### Header

Utils/SocketCommClass.h

#### 4.4.4.2.24 GetPortList ()

Retrieves the list of ports that Remote is managing.

```
bool GetPortList(TSSPData& Reply);
```

#### Parameters

Reply - An SSP Data packet containing the port list information. The port list itself contains a port and a description string such as:

COM1:<0>MODEM1:115200<0>COM3:<0>RADIO3:1200<0>

// It's parsed as follows:

```
TSSPData Reply;
if (Port.GetPortList(Reply))
{
    TprotoOpcode Opcode;
    int Len;
    Reply.Read(Opcode, Len);
    for (;;)
    {
        CString ComStr;
        CString DescStr;
        Reply.Read(ComStr);
```

```

        if (ComStr.IsEmpty())
            break;
        Reply.Read(DescStr);
        // Process the ComStr and DescStr here

    };
}
else
    // an error occurred, we must be connected to the
    server.

```

## Return Value

Result of the operation.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.25 GetComOptions()

Retrieves a number of bit mapped status flag pertaining to the com port.

```
bool GetComOptions(int& Options);
```

## Parameters

### Options

- **IGNORECD**: Remote was waked via Rx data, and hence the state of CD or DTR is ignored for determining whether the connection is still live – or the port is in a mode (ISSPONLY) where CD is not used.

**LOGGINGIN**: A user is logging in.

**LOGGEDIN**: A user has completed log in.

**ISMODEM**: A modem connection.

- DTR is normally kept high
- DTR is dropped in order to hang-up the modem (the settings in an external modem should be configured to allow this)
- CD is expected to go high when a connection is made
- If CD goes low the connection is dropped
- RTS/CTS hardware handshaking is enabled by default, but can be disabled with the /HSn- option in a Remote.opt file(where n is the port number).
- Inactivity timeout is used

**ISRADIO**: A radio connection.

- Inactivity timeout is not used
- Command line is never processed
- A message is not sent until CS is low
- DTR and (by default)y RTS are raised before a message is sent and lowered after the last byte.
- A delay is inserted after DTR is raised and the first byte is

sent. This is called the carrier delay and it allows downstream radios and repeaters time to lock on to the signal.

- RTS/CTS hardware handshaking is disabled by default, but may be enabled by specifying the /HSn+ option in a Remote.opt file (where n is the port number).
- Debug messages will not be sent to the port when this flag is set.

**ISDIRECT:** A direct connection.

- If DSR is high when a message is received then the connection will terminate if DSR drops.
- RTS/CTS hardware handshaking is enabled by default, but can be disabled with the /HSn- option in a Remote.opt file (where n is the port number).
- DTR is raised when a connection is made, and lowered when it has been dropped unless it's been forced on or off all the time by the /DTRn+ or /DTRn- option in the Remote.opt file (n is the port).

**ISVOICE:** A Sutron voice modem.

- An extra 3sec delay is added on power up.
- The following initialization string is sent to the modem when powered up: "ATS0=1E0Q1&D0&C1&W"
- DTR is dropped when a user is logged out, and raised whenever a connection is made. The modem supports automatic power-on when a ring occurs, but will eventually timeout and hangup if DTS is not asserted.

**DEBUGCOMMANDS:** Debug commands enabled (debug on has been issued at the \Flash Disk prompt)

**QUICKPROTOCOL:** SSP Quick Protocol is enabled.

- This means the /C option has been passed after the user name when the last login occurred.
- The command prompt and command parser are enabled.

**DIALIN:** A user has dialed and connected via a modem.

- The serial port may not be taken over for other purposes while a user is dialed in.

**SSPLOGIN:** True when an SSP login has occurred.

- This is a flag that is set when a login occurs via the OpLogin SSP message.
- Debug messages will not be sent to the port when this flag is set.
- Xterm performs logins using SSP.

**ENABLECOMMANDPROMPT:** Command prompt is enabled.

- This enables the \Flash Disk> prompt to be displayed.

- Command parsing can occur even when prompting is disabled. This allows for a custom prompt.
  - Debug messages can be sent to the port when this flag is set.
- ENABLESSP:** SSP Packet handling is enabled. SSP Packets are automatically detected and passed on in their entirety.

**ENABLECOMMANDPARSER:** Command parsing is enabled.

- Any of remote's commands may be issued and will be processed.
- The command parser may be disabled to permit only custom commands to be processed.
- Debug messages can be sent to the port when this flag is set.

**ISSPONLY:** A port which strictly talks SSP, the following options are automatically selected when a connection is made:

- Logging in is not required
- Setup Mode access is granted
- The Inactivity Timeout is disabled
- The status of CD and/or DSR are ignored (will not cause a disconnect)
- The \Flash Disk command prompt is disabled
- SSP is enabled
- SSP Parser is enabled
- DTR is raised before each packet is sent and lowered after
- Debug messages will not be sent to the port when this flag is set.

**ISRS485:** An RS485 connection.

- Debug messages will not be sent to the port when this flag is set.
- RS-485 is half duplex by nature so remote sets the ISSPONLY flag when an RS485 connection is selected, disabling the command parser.
- RTS is set high whenever data is transmitted. This control line enables the RS-485 transmitter.
- DTR is set high whenever data is ready to be received. This control line enables the RS-485 receiver.

**ENABLESSPPARSER:** The SSP Command Parser is enabled.

- The SSP Command Parser processes various default SSP commands.
- The SSP Parser may be disabled to allow only command line commands and/or potentially custom SSP messages to be processed.

Return Value

Result of the operation.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.26 SetComOptions ()

SetComOptions can be used to change the state of Remote.

```
bool SetComOptions (int Options, int Mask);
```

#### Parameters

- |         |   |
|---------|---|
| Options | - A bit map of options to set or clear.<br>Most of the Options are status oriented. See GetComOptions for a list.<br>The options to disable the command parser can be usefull, and also a command prompt is automatically output to the user when the comand prompt is enabled. |
| Mask    | - A bit mask to determine which fields in the options should be set and which ones should be ignored.   |

#### Return Value

Result of the operation.

## Header

Utils/SocketCommClass.h

### 4.4.4.2.27 SetExtendedCommands ()

This command is used by a command line enhancement application which wishes to handled additional commands. It is used in conjunction with the AddCustomCommandParser() and AddCustomGroup() functions in Users.h. It specifies the list of commands which the command line enhancer wishes to support.

```
bool SetExtendedCommands(CString CommandList);
```

#### Parameters

- |             |  |
|-------------|--|
| CommandList | - A list of commands with commas before and after each one.<br><br>For instance, _T(",test,xpert,get,") would cause the commands test, xpert, or get to be passed along to a custom command parser.<br><br>All commands may be interecepted by specifying _T("*") for the CommandList. |
|-------------|--|

#### Return Value

Result of the operation.

## Header

Utils/SocketCommClass.h

#### 4.4.4.2.28 RunCommand ()

Causes Remote to execute a command as if a user had typed it in to the command line.

```
bool RunCommand (CString CommandLine);
```

##### Parameters

CommandLine            - The command to execute. For instance the command `_T("dir")` will cause remote to display a file list to the user.

##### Return Value

Result of the operation.

##### Header

Utils/SocketCommClass.h

### 4.4.5 TResourceKey

When Sutron Link Libraries load resources, they must first indicate from where the resource is to come. This is typically accomplished by creating an instance of TResourceKey, providing the library's handle as a parameter, as in the following example:

```
void CSDI::ShowProperties(CWnd* pParent)
{
    TResourceKey key(DefLibSLL);
    CSDIDlg dlg(pParent);
    dlg.m_iAddressIdx = m_propAddressIdx;
    dlg.m_strCommand = m_propCommand;
    if (dlg.DoModal() == IDOK)
    {
        m_propAddressIdx = dlg.m_iAddressIdx;
        m_propCommand = dlg.m_strCommand;
    }
}
```

In the example, the definition of "key" actually translates to a call to the MFC function `AfxSetResourceHandle()` to set the current resource handle to that of the library. When "key" is destroyed on return from the function, another call to `AfxSetResourceHandle()` occurs to set the resource handle back to its original value. This ensures that any resources used in the context of this function will be loaded from the correct library. If the current resource handle is not set properly, then the wrong resource may be loaded.

The Xpert SDK template code contains TResourceKey definitions in functions where resource loads are known to occur.

TResourceKey is defined in `Utils/ResourceKey.h`.

### 4.4.6 TTime and TTimeSpan

The Xpert (versions 3 and higher) has high-resolution real-time keeping capabilities. TTime and TTimeSpan are each accurate to 1 millisecond.

These two classes are used to represent absolute and relative time, respectively. TTime is very similar to the MFC class CTime, but maintains time to a resolution of 1 millisecond, instead of 1 second. TTimeSpan is very similar to the MFC class CTimeSpan, but maintains timespan to a resolution of 1 millisecond, instead of 1 second.

Be careful of mixing CTime and CTimeSpan with these two classes. They are NOT compatible. TTime has as a reference (i.e., 0-time) of January 1, 1601 versus CTime's reference of January 1, 1970, and has a much greater range.

See the "Utils/Time.h" header for the complete class API.

#### 4.4.7 Exported Utils Functions

This section contains descriptions of the functions exported from the Utils library.

##### 4.4.7.1 StrToTime()

This function converts a string representing time to an actual TTime.

```
TTime StrToTime(const CString& string);
```

###### Parameters

string                      - String having the format "MM/DD/YYYY HH:MM:SS".

###### Return Value

A TTime value containing the time indicated by the string.

###### Header

Utils/Time.h

##### 4.4.7.2 StrToTimeSpan()

This function converts a string representing a time-span to an actual TTimeSpan.

```
TTimeSpan StrToTimeSpan(const CString& string);
```

###### Parameters

string                      - String having the format "HH:MM:SS:MS" (MS, or milliseconds, are optional).

###### Return Value

A TTimeSpan value containing the time-span indicated by the string.

###### Header

Utils/Time.h

##### 4.4.7.3 StrToMS()

This function converts a string representing a time-span to an unsigned integer representing total number of milliseconds in the span.

```
DWORD StrToMS(const CString& string);
```

#### Parameters

- string                      - String having the format “HH:MM:SS:MS” (MS, or milliseconds, are optional).

#### Return Value

An unsigned integer value containing the number of milliseconds in the total span indicated by the string.

#### Header

Utils/Time.h

#### 4.4.7.4 MSTil()

This function computes the number of milliseconds between current time and the time specified.

```
DWORD MSTil(const TTime& tNext);
```

#### Parameters

- tNext                      - The time to use when computing the number of milliseconds from current time.

#### Return Value

An unsigned integer value containing the number of milliseconds difference between current time and the specified time.

#### Header

Utils/Time.h

## 4.5 Setup API

It is often necessary to save and restore data to and from the setup file. Setup blocks have their properties (member data of type TProperty) stored and restored automatically whenever the setup is saved or read, respectively. However, any SLL may “hook” into the setup save and read process by defining and exporting specific functions that use a reference to an instance of CXMLSetup to save and read setup data. The following sections describe the application exports and methods of CXMLSetup required to support setup reading and writing.

### 4.5.1 Application Exports

The SLL application code must define and export the following four functions in order for Xpert to support the SLLs save and read of setup data:

```
bool InitSetup();  
bool ReadSetup(CXMLSetup& XMLSetup);  
bool WriteSetup(CXMLSetup& XMLSetup);  
LPCTSTR GetSetupTag();
```

Descriptions of how each of these functions should be implemented follow.



#### 4.5.1.1 InitSetup()

Implement this function to initialize setup data to a default state. The Xpert application framework calls this function when the user selects *Setup File::New* from the control panel on the Setup page.

```
extern "C" bool _declspec(dllexport) InitSetup();
```

##### Parameters

None -

##### Return Value

Return “true” to indicate a successful init, false otherwise.

##### Example

```
extern "C" bool _declspec(dllexport) InitSetup()
{
    // Set common settings to defaults.
    m_iPort = COM2;
    m_strSatID = _T("00000000");
    m_bInitSatlink = true;

    // Set self-timed and random settings to defaults.
    m_SelfTimedMsgr.InitSetup();
    m_RandomMsgr.InitSetup();

    return true;
}
```

#### 4.5.1.2 ReadSetup()

Implement this function to read the setup data using the XML object provided. The Xpert application framework calls this function when the setup tag defined by the GetSetupTag() function is encountered while reading the setup file. It is this function’s responsibility to read in all of the data it “owns” and to stop when the last value of owned data has been read (i.e., reading past owned data will likely crash the system).

```
extern "C" bool _declspec(dllexport) ReadSetup(CXMLSetup& XMLSetup);
```

##### Parameters

XMLSetup - Provides access to the contents of the setup file. See the section on CXMLSetup for a description of the methods of this object.

##### Return Value

Return “true” to indicate a successful read, false otherwise.

##### Example

```
extern "C" bool _declspec(dllexport) ReadSetup(CXMLSetup& XMLSetup)
{
    CXMLToken XMLToken;
    XMLSetup.ReadNextToken(XMLToken);
    while (XMLToken.nType == CXMLToken::START_TAG)
    {
```

```

CString strTag = XMLToken.GetText();
if (!strTag.Compare(_T("Common")))
    ReadCommonSetup(XMLSetup);
else if (!strTag.Compare(_T("SelfTimed")))
    m_SelfTimedMsgr.ReadSetup(XMLSetup);
else if (!strTag.Compare(_T("Random")))
    m_RandomMsgr.ReadSetup(XMLSetup);
else
{
    // Unknown tag entry. Eat and discard all entries for this tag.
    XMLSetup.ReadNextToken(XMLToken);
    while (XMLToken.nType == CXMLToken::ATTR_NAME)
    {
        CString strName = XMLToken.GetText();
        XMLSetup.ReadNextToken(XMLToken);
        CString str;
        str.Format(_T("Eating unknown token: %s."), strName);
        Report.Warning(str);
        XMLSetup.ReadNextToken(XMLToken);
    }
    XMLSetup.ReadNextToken(XMLToken);
}
return true;
}

bool ReadCommonSetup(CXMLSetup& XMLSetup)
{
    CXMLToken XMLToken;

    // Read and validate common config.
    XMLSetup.ReadNextToken(XMLToken);
    while (XMLToken.nType == CXMLToken::ATTR_NAME)
    {
        CString strName = XMLToken.GetText();
        XMLSetup.ReadNextToken(XMLToken);
        if (!strName.Compare(_T("Port")))
            m_iPort = _ttoi(XMLToken.GetText());
        else if (!strName.Compare(_T("SatID")))
            m_strSatID = XMLToken.GetText();
        else if (!strName.Compare(_T("InitSatlink")))
            m_bInitSatlink = _ttoi(XMLToken.GetText()) ? true : false;
        else
        {
            CString str;
            str.Format(_T("Eating unknown token: %s."), strName);
            Report.Warning(str);
        }
        XMLSetup.ReadNextToken(XMLToken);
    }
    return true;
}

```

The above expects a "Satlink" section having the following format. The partitioning into subsections (Common, SelfTimed, and Random in the example) is required by the Xpert XML parser. The SelfTimed and Random sections are handled by routines not shown here. NOTE: Spaces are not allowed in token

names (e.g., "SatID" is acceptable as a token name, "Sat ID" is not).

```
<Satlink>
  Common
    Port = "0"
    SatID = "00000000"
    InitSatlink = "1"
  />
  <SelfTimed
    . . .
  />
  <Random
    . . .
  />
</Satlink>
```

#### 4.5.1.3 WriteSetup()

Implement this function to store the desired setup data using the XML object provided.

```
extern "C" bool _declspec(dllexport) WriteSetup(CXMLSetup& XMLSetup);
```

##### Parameters

**XMLSetup** - Provides access to the contents of the setup file. See the section on CXMLSetup for a description of the methods of this object.

##### Return Value

Return "true" to indicate a successful save, false otherwise.

##### Example

```
extern "C" bool _declspec(dllexport) WriteSetup(CXMLSetup& XMLSetup)
{
    CString str;

    // Open outermost Satlink entry.
    XMLSetup.WriteStartTag(_T("<Satlink>"));

    // Write common config
    XMLSetup.WriteStartTag(_T("<Common>"));
    str.Format(_T("Port = \"%d\""), m_iPort);
    XMLSetup.WriteText(str);
    str.Format(_T("SatID = \"%s\""), m_strSatID);
    XMLSetup.WriteText(str);
    str.Format(_T("InitSatlink = \"%d\""), m_bInitSatlink);
    XMLSetup.WriteText(str);
    XMLSetup.WriteEndTag(_T(">"));

    // Write self-timed and random configs.
    m_SelfTimedMsgr.WriteSetup(XMLSetup);
    m_RandomMsgr.WriteSetup(XMLSetup);

    // Close outermost Satlink entry.
    XMLSetup.WriteEndTag(_T("</Satlink>"));
}
```

```

    ValidateSetup();
    return true;
}

```

The above results in a "Satlink" section having the following format. The partitioning into subsections (Common, SelfTimed, and Random in the example) is required by the Xpert XML parser. The SelfTimed and Random sections are handled by routines not shown here. NOTE: Spaces are not allowed in token names (e.g., "SatID" is acceptable as a token name, "Sat ID" is not).

```

<Satlink>
  <Common
    Port = "0"
    SatID = "00000000"
    InitSatlink = "1"
  />
  <SelfTimed
    . . .
  />
  <Random
    . . .
  />
</Satlink>

```

#### 4.5.1.4 GetSetupTag ()

Implement this function to return a pointer to a string to be used as the name of the section within the setup file that belongs to the SLL.

```
extern "C" LPCTSTR _declspec(dllexport) GetSetupTag();
```

##### Parameters

None.

##### Return Value

Pointer to a string containing the setup tag.

##### Example

```

extern "C" LPCTSTR _declspec(dllexport) GetSetupTag()
{
    return _T("Satlink");
}

```

## 4.5.2 CXMLSetup Methods

The exports defined by the SLL application code to support setup reading and writing use an instance of CXMLSetup to access the setup file. Descriptions of the required CXMLSetup methods follow.

#### 4.5.2.1 WriteStartTag()

This method writes a section opening tag to the setup file and tracks the tag depth.

```
int CXMLSetup::WriteStartTag(  
    LPTSTR lpszText);
```

**Parameters**

lpszText - Pointer to a string containing the start tag text.

**Return Value**

Integer containing the tag depth.

**Header**

Engine/XMLSetup.h

#### 4.5.2.2 WriteEndTag()

This method writes a section closing tag to the setup file and tracks the tag depth.

```
int CXMLSetup::WriteEndTag(  
    LPTSTR lpszText);
```

**Parameters**

lpszText - Pointer to a string containing the end tag text.

**Return Value**

Integer containing the tag depth.

**Header**

Engine/XMLSetup.h

#### 4.5.2.3 WriteText()

This method writes the provided text to the setup file subject to the current tag depth.

```
void CXMLSetup::WriteText(  
    LPTSTR lpszText);
```

**Parameters**

lpszText - Pointer to a string containing the text to write.

**Return Value**

None.

**Header**

Engine/XMLSetup.h

## 4.6 Log API

The system dll, “LogMgr.dll”, maintains the system’s list of available logs. This list is an instance of the class CLogList, named LogList, which is exported for use by other libraries. The list contains objects of type CLogDesc, which is a class used to encapsulate a single log.

Typically, a developer needs to programatically iterate through the list of available logs. The following is an example of how to do this:

```
POSITION pos = LogList.GetHeadPosition();
while (pos != NULL)
{
    CLogDesc* pLog = LogList.GetNext(pos);
    // Use pLog here...
}
```

Here is an example of retrieving a log by name:

```
CLogDesc* pLog = LogList.GetByName(_T("\\Flash Disk\\system.log"));
// Use pLog here...
```

#### 4.6.1.1 CLogDesc

Instances of this class represent individual logs in the system. To read data from a log you use the LOG and LOGCURSOR classes documented in the next section. You can access to lower-level LOG class via the log member variable. For instance, you could retrieve the number of lines in the log file from the example above with the following code:

```
UINT32 LineCount = pLog->log.GetLineCount();
```

See LOG and LOGCURSOR for more information.

##### 4.6.1.1.1 CLogDesc()

This is the class constructor.

```
CLogDesc(
    TCHAR* lpszName,
    int Size,
    bool Wrap,
    bool IgnoreBadData);
```

#### Parameters

- |               |   |
|---------------|---|
| lpszName      | - Full path of the log file (e.g., _T("\\Flash Disk\\new.log"))   |
| Size          | - The size of the log file in bytes.  |
| Wrap          | - Flag indicating whether the log should “wrap” its data, that is, overwrite old data with new when the log fills up.                       |
| IgnoreBadData | - Flag indicating whether the quality of data logged is relevant (e.g., when “true”, quality of logged data is not shown on View Log page). |

#### Return Value

None.

#### Header

LogMgr/LogDesc.h

##### 4.6.1.1.2 Create()

This method creates the log file.

```
bool Create();
```

Parameters

None.

Return Value

If the log file is successfully created, “true” is returned. Otherwise, “false” is returned.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.3 Open()

This method attempts to open the log file. If the log file does not exist, it is not created.

```
bool Open();
```

Parameters

None.

Return Value

If the log file is successfully opened, “true” is returned. Otherwise, “false” is returned.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.4 GetName()

This method returns the name of the log file.

```
LPTSTR GetName();
```

Parameters

None.

Return Value

A pointer to the internal buffer containing name of the log file.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.5 SetName()

This method sets the name of the log file.

```
void SetName(  
    LPCTSTR lpszName);
```

Parameters

lpszName - Full path of the log file (e.g., \_T("\\Flash Disk\\new.log"))

Return Value

None.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.6 GetSize()

This method returns the size of the log file in bytes.

```
int GetSize();
```

Parameters

None.

Return Value

The size of the log file in bytes.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.7 IsWrap()

This method returns the wrap log attribute (a flag indicating whether new data overwrites old data once a log has been filled).

```
bool IsWrap();
```

Parameters

None.

Return Value

The value of the wrap attribute is returned. When “true”, new data overwrites old data once the log has been filled. When “false”, writes of new data fail once a log has been filled.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.8 IsIgnoreBadData()

This method returns the ignore-bad-data log attribute (a flag indicating whether the quality assigned to data is relevant).

```
bool IsIgnoreBadData();
```

Parameters

None.

Return Value



The value of the ignore-bad-data attribute is returned. When “true”, the quality of a log is not displayed in View Log.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.9 InUse()

This method returns an indication as to whether the log is in use by the current setup, where “in use” means a log-type setup block with this log as its selection exists in the setup. When logs are in use, their attributes cannot be manipulated, nor can they be deleted.

```
bool InUse();
```

Parameters

None.

Return Value

When the log is in use by the setup, “true” is returned. Otherwise, “false” is returned.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.10 AppendSensor()

This method adds sensor data to the log.

```
BOOL AppendSensor(  
    CString Sensor,  
    CSensorData& data);
```

Parameters

- |        |   |
|--------|---|
| Sensor | - The name of the sensor being logged.  |
| data   | - The data to log. See Engine/Module.h for the definition of the CSensorData class. |

Return Value

None.

Header

LogMgr/LogDesc.h

#### 4.6.1.1.11 AppendNote()

This method adds a note (a text string) to the log.

```
BOOL AppendNote(  
    LPCTSTR szNote);
```

Parameters

szNote - The note to log.

Return Value

None.

Header

LogMgr/LogDesc.h

#### 4.6.1.2 LOG class

This is the lower level representation of a log file. The functionality not typically performed by the CLogDesc class is documented here.

##### 4.6.1.2.1 GetLineCount()

Retrieves the number of lines in the log file.

```
UINT32 GetLineCount();
```

Parameters

None.

Return Value

The number of lines in the log file.

Header

Logger/Logger.h

##### 4.6.1.2.2 Changed()

Is used to detect whether a change has been made to a log file..

```
bool Changed(UINT32& Counter);
```

Parameters

Counter - The Log keeps track of how many changes occur with a counter. If the Log's internal count differs from the count passed in, then Changed() returns true, and the new count is stored in Counter. Counter is typically a class variable and not a local variable. It can be initialized by calling Changed() and ignoring the result.

Return Value

True if the log has changed since the last time Changed() was called with the specified Counter variable.

Header

Logger/Logger.h

##### 4.6.1.2.3 Flush()

Saves any unsaved log data out to disk.

```
void Flush(bool Force=true);
```

#### Parameters

- Force
- Data is normally saved based on a timer. If Force is false, then new data will not actually be written out to disk unless the timer has expired (the default is once per minute). Setting Force to true causes data to be written out immediately.

#### Return Value

None.

#### Header

Logger/Logger.h

### 4.6.1.3 LOGCURSOR class

The LOGCURSOR class is used to navigate and read data from a log file. It's primary function is to try to maintain a specific location in the log file even though new lines may be added and old ones deleted. Because data may be deleted dynamically, an absolute line number in the log has little meaning. For instance, the information at line 500 will not be the same after 50 lines are deleted. So typically a LOGCURSOR is used to navigate in a relative sense, such as moving forwards and backwards from a certain location. A log cursor also buffers a block of data from the log, capturing the state of a section of the log at a specific time. So even if the current line is rolled out of the log and deleted, the cursor will not become corrupt. The cursor resyncs with the log at various times (rebuffering data as needed) such as whenever the cursor is moved to a new location, or the Sync() method is called.

#### 4.6.1.3.1 LOGCURSOR()

Is used to construct a log cursor.

```
LOGCURSOR(LOG& log);
```

#### Parameters

- log
- A reference to the log to be navigated.

#### Return Value

None.

#### Header

Logger/Logger.h

#### 4.6.1.3.2 ~LOGCURSOR()

Destroys a log cursor.

```
~LOGCURSOR();
```

#### Parameters

None.

#### Return Value

None.

#### Header

Logger/Logger.h

#### 4.6.1.3.3 GotoBottom ()

Move to the most recent data in the log.

```
bool GotoBottom();
```

#### Parameters

None.

#### Return Value

Returns false if there is a fatal error in the log and can't find the first line.

#### Header

Logger/Logger.h

#### 4.6.1.3.4 GotoTop()

Moves to the oldest data in the log.

```
bool GotoTop();
```

#### Parameters

None.

#### Return Value

Returns false if there is a fatal error in the log and can't find the last line.

#### Header

Logger/Logger.h

#### 4.6.1.3.5 MoveTo ()

Moves to a line in the log.

```
bool MoveTo(int AbsLine);
```

#### Parameters

AbsLine                      - An absolute line number between 0 and GetLineCount()-1

#### Return Value

Returns false if a move before the beginning of the log or after the end of the log occurs.

## Header

Logger/Logger.h

### 4.6.1.3.6 MoveBy()

Move a relative number of lines in the log.

```
bool MoveBy(int RelLine);
```

#### Parameters

RelLine                      - Number of lines to move forward in time (positive values) or backwards in time (negative values) in the log.

#### Return Value

Returns false if a move before the beginning of the log or after the end of the log occurs.

## Header

Logger/Logger.h

### 4.6.1.3.7 MoveNext ()

Moves to the next line in the log, skipping any bad.

```
bool MoveNext;
```

#### Parameters

None.

#### Return Value

False when a move after the bottom of the log is attempted.

## Header

Logger/Logger.h

### 4.6.1.3.8 MovePrev ()

Moves to the previous line in the log, skipping any bad blocks.

```
bool MovePrev();
```

#### Parameters

None.

#### Return Value

False when a move before the top of the log is attempted.

## Header

Logger/Logger.h

#### 4.6.1.3.9 Search ()

Search backwards from the end of the log for the first data which is before the passed Time, and sets the cursor to point to the data after that point which would be at or after the Time. If no data is found false is returned and the position is set to the top.

```
bool Search(INT64 Time);
```

##### Parameters

Time - A time and date stamp compatible with TTime::GetTicks()

##### Return Value

False if no data could be located.

##### Header

Logger/Logger.h

#### 4.6.1.3.10 IsNote ()

Returns true if the current record is a note, false if it contains Data. Always call this first in order to determine whether to call ReadNote() or ReadSensor() next, and so the data can be prepared.

```
bool IsNote();
```

##### Parameters

None.

##### Return Value

True if the Cursor is pointing to a log line that contains a note, and not sensor data.

##### Header

Logger/Logger.h

#### 4.6.1.3.11 ReadNote ()

Read a log note from the current record. Always call IsNote() before calling this. The caller must allocate space for the note, the size depends on the longest note Appended (typically 128 characters).

```
bool ReadNote(INT64& Time, TCHAR* Note);
```

##### Parameters

Time - A variable to receive the time stamp of the log line. It can be converted to a TTime as follows:  
TTime TimeStamp(Time);

Note - A preallocated TCHAR array to receive the log note, allow enough space for 128 characters.

##### Return Value

False if there is a problem, such as no data was available.

## Header

Logger/Logger.h

### 4.6.1.3.12 ReadTime ()

Reads just the current time of the current line, if the current. line is invalid, then it returns the base time of the bad block which contains the line.

```
INT64 ReadTime();
```

## Parameters

None.

## Return Value

A log time stamp for the current line compatible with TTime.

## Header

Logger/Logger.h

### 4.6.1.3.13 ReadSensor ()

Reads sensor data from the current record. Always call IsNote() before calling this. The caller must allocate space for the Name, Units and Num, the size depends on the longest Names & Units stored in the log. Allocate room for at least 128 characters for each TCHAR array.

```
bool ReadSensor(INT64& Time, TCHAR* Name, TCHAR* Units, UINT8& Quality,  
TCHAR* Num)
```

## Parameters

- |         |  |
|---------|--|
| Time    | - A variable to receive the time stamp of the log line. It can be converted to a TTime as follows:<br>TTime TimeStamp(Time);                                     |
| Name    | - The name of the sensor logged.   |
| Units   | - The units the sensor was measured in.  |
| Quality | - The quality flag associated with the sensor. Possible values are contained in CSensorData::QualityType and include 0 for GOOD, 1 for BAD, and 2 for UNDEFINED. |
| Num     | - A measured value of the sensor that was logged for the specified time stamp.   |

## Return Value

False if there is a problem, such as no data was available.

## Header

Logger/Logger.h

### 4.6.1.3.14 AtTop ()

Detects whether the current line is the first line in the log..

```
bool AtTop();
```

**Parameters**

None.

**Return Value**

True if the cursor points to the oldest line in the log.

**Header**

Logger/Logger.h

#### 4.6.1.3.15 AtBottom ()

Detects whether the current line is the last line in the log.

```
bool AtBottom();
```

**Parameters**

None.

**Return Value**

True if the cursor points to the newest line in the log.

**Header**

Logger/Logger.h

#### 4.6.1.3.16 GetCurrentLine()

Returns the line number the cursor is position at, 0..LineCount-1.

```
int GetCurrentLine();
```

**Parameters**

None.

**Return Value**

The line number of the current line. This value may change as lines are added to the log and old lines are removed.

**Header**

Logger/Logger.h

#### 4.6.1.3.17 Sync ()

Synchronizes the Cursor with the Log, the Move commands do this automatically but other commands do not. For instance if GetCurrentLine() returned 100, and then 5 lines were deleted from the log, GetCurrentLine() would keep returning 100 until a move occurs or Sync() is called, at which point the value would change to 95.

```
void Sync();
```



Parameters

None.

Return Value

None.

Header

Logger/Logger.h

## 5 Coding Guidelines

### 5.1 General

When creating dialog boxes for the Xpert, the dialogs should be instances of classes derived from TDialog, whenever possible. The TDialog class was created to fix a CE bug that causes dialogs to occasionally become inaccessible by disappearing behind other windows. TDialog is a class defined in Engine\Module.h.

### 5.2 GUI Guidelines

This section provides a set of guidelines governing the design of the Graphical User Interface (GUI) of Xpert applications. These guidelines should be followed in order to present the user with a consistent interface:

1. Set dialog fonts to Tahoma 10pt. Note that DLU's (the units used in the development tools and referred to below) are relative to the font size. So if you're using a font other than Tahoma 10pt., these sizes no longer apply.
2. Set dialog margins to 3 DLU's.
3. Change buttons:
  - a. Set the caption of change buttons to "...".
  - b. When the field label is above the data field, size the change button to 12x9 DLU's and place it right justified to the data field.
  - c. When the field label precedes the data field, size the change button to 14x12 DLU's, and place it to the right of the data field.
  - d. Some examples:

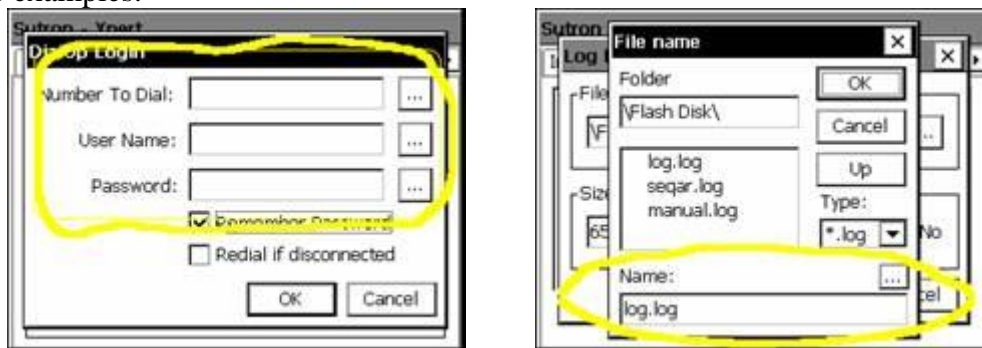
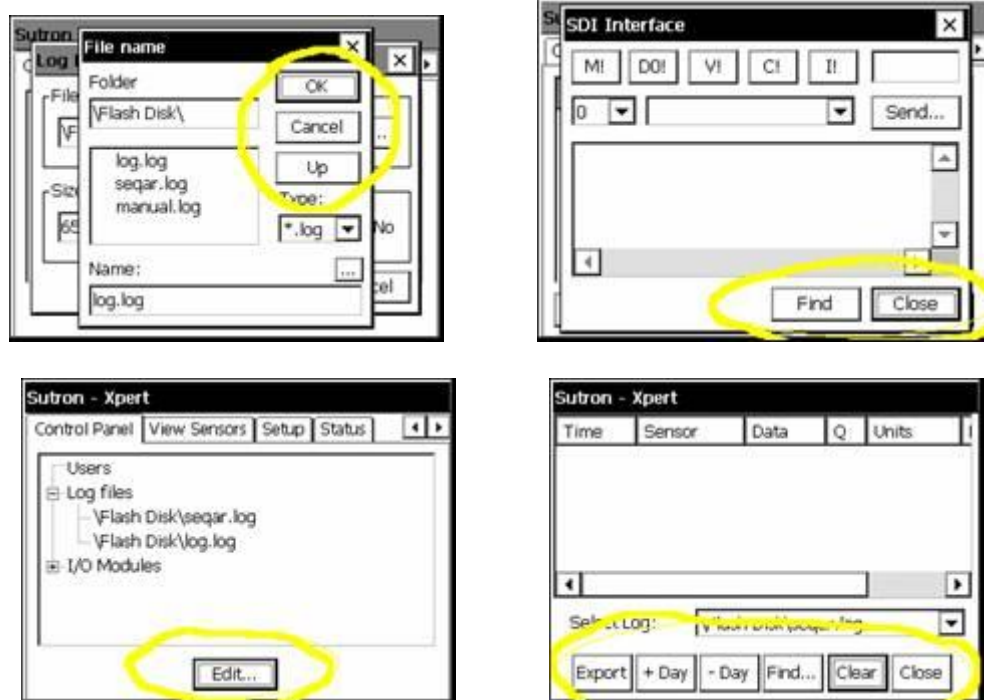


Figure 5: Location of Change Buttons

4. Set standard buttons to 36x12 DLU's.
5. Set the height of edit controls to 12 DLU's.

6. Dialog buttons:
  - a. The standard placement of dialog buttons is top, right, and vertically aligned, with OK on top, followed by Cancel, followed by any other buttons (excluding control-specific buttons like the change button, of course). If Close exists in place of OK and Cancel, then Close should be on top.
  - b. The alternative to the standard dialog button placement is bottom, justified right (or center\*), and horizontally aligned, with OK left of Cancel, and Cancel left of any other buttons. If Close exists in place of OK and Cancel, then Close should be rightmost. \*Center is typically used when there is either only one button, or enough buttons to span about 75% of the horizontal.
  - c. Some examples:



**Figure 6: Alignment of Standard Buttons**

7. The maximum size of a dialog in DLUs is 179x105.
8. The size of a property page in DLUs is 175x105.
9. Setup block icons should have a border no thicker than a single line, if any, so as not to cause the block to appear as active (the Xpert application handles darkening the borders of active blocks).

## 6 Sample Programs, SLL's, and Blocks

This section contains samples designed to help users understand how to use the various APIs.

### 6.1 Terminal Server

TerminalServer is a demonstration project which shows how the command line of Remote.Exe can be extended. The procedure involves the following steps:

1. TerminalServer creates a custom user group, and hooks in to Remote's login and command processing.
2. You need to then go in to the XPert's control panel and create a new user, and specify the group "TerminalServer".
3. Then the next time you login that user in to remote, you should see a custom prompt:

Hello \*USERNAME, Terminal Server is Active. Use Help for a list of commands including extended ones.

\Flash Disk>

4. You may then use any of the standard remote commands, or any of the extended commands.

Extended commands implemented by this SLL include:

HELP	- Displays both the old and new commands.
GET	- Retrieves and displays log data
LOGFILE	- Switches log files for the get command (default is ssp.log)
LOGTIME	- Specifies the retrieval time for the get command
LOGDATE	- Specifies the retrieval date for the get command
LIST	- Displays the last value logged for a sensor or sensor(s)
READ	- Measures and displays readings for a sensor or sensor(s)

### 6.1.1 TerminalServer.cpp

This is the DLL's main source file containing the definition ofDllMain().

Primarily we install the .SLL in this file, we create our new custom group here. We use CUSTOM\_GROUP1, different .SLL's can be made that support different groups by simply changing this to CUSTOM\_GROUP2-10. We also specify that this custom group has a custom command parser.

```
// TerminalServer.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include <afxdllx.h>

#include "../Utils/Users.h"
#include "TerminalServerMgr.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// GetFileVersion()

extern "C" _declspec(dllexport) LPCTSTR GetFileVersion()
{
    // Note: when we upgrade to WCE 3.0, change this to retrieve FileVersion
}
```

```

    // entry of VS_VERSION_INFO.
    return _T("1.2.0.4");
}

bool TerminalServerFunction(TUserGroup Group, LPCTSTR ComPort, LPCTSTR UserName)
{
    return TerminalServerMgr.TerminalServerFunction(Group, ComPort, UserName);
};

bool TerminalServerParser(TUserGroup Group, LPCTSTR ComPort, LPCTSTR UserName, LPCTSTR Command)
{
    return TerminalServerMgr.TerminalServerParser(Group, ComPort, UserName, Command);
};

////////////////////////////////////
// DllMain()

AFX_EXTENSION_MODULE TerminalServerSLL = { NULL, NULL };

extern "C" int APIENTRY
DllMain(HANDLE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        TRACE0("TerminalServer.SLL Initializing!\n");

        // Extension DLL one-time initialization
        if (!AfxInitExtensionModule(TerminalServerSLL, (HINSTANCE)hInstance))
            return 0;

        // Insert this DLL into the resource chain
        // NOTE: If this Extension DLL is being implicitly linked to by
        // an MFC Regular DLL (such as an ActiveX Control)
        // instead of an MFC application, then you will want to
        // remove this line from DllMain and put it in a separate
        // function exported from this Extension DLL. The Regular DLL
        // that uses this Extension DLL should then explicitly call that
        // function to initialize this Extension DLL. Otherwise,
        // the CDynLinkLibrary object will not be attached to the
        // Regular DLL's resource chain, and serious problems will
        // result.

        new CDynLinkLibrary(TerminalServerSLL);

        if (!AddCustomGroup(CUSTOM_GROUP1, _T("TerminalServer"), TerminalServerFunction))
            return 0;
        AddCustomCommandParser(CUSTOM_GROUP1, TerminalServerParser);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
    {
        TRACE0("TerminalServer.SLL Terminating!\n");
        // Terminate the library before destructors are called
        AfxTermExtensionModule(TerminalServerSLL);
    }
    return 1;    // ok
}

```

## 6.1.2 TerminalServerMgr.cpp

This is where most of the work of the application occurs.

There are two methods for creating custom reports or custom commands. If a response can be generated quickly (in under a minute) then the processing can be done in the TerminalServerFunction or TerminalServerParser methods directly. This is in fact how this demo works. If more time is needed, then these functions should create a thread for doing the processing, and return true immediately.

While it may appear at first glance that we are accessing the com ports directly we are in fact using a special class called CSocketCommClass which let's us share any serial port which Remote.exe has open using Interprocess Communication via Windows Sockets.

Methods defined:

void Start(); - Skeleton code not used, but intended for thread processing.

void Stop(); - Skeleton code not used, but intended for thread processing.

bool ReadSetup(CXMLSetup& XMLSetup); - Example code for allowing the module to read it's properties from the setup.

bool WriteSetup(CXMLSetup& XMLSetup); - Example code for allowing the module to read it's properties from the setup.

bool TerminalServerFunction(TUserGroup Group, LPCTSTR ComPort, LPCTSTR serName); - This function is called whenever the custom user logs in on one of remote's ports.

bool TerminalServerParser(TUserGroup Group, LPCTSTR ComPort, LPCTSTR UserName, LPCTSTR Command); - This function is called whenever the user attempts to use a custom command.

```
// TerminalServerMgr.cpp: implementation of the CTerminalServerMgr class.
//
///////////////////////////////////////////////////////////////////
//
// Other SLL's should copy how Setups are handled in this SLL

#include "stdafx.h"

#include "TerminalServerMgr.h"
#include "../Engine/Engine.h"
#include "../Utils/Report.h"
#include "../Utils/SocketCommClass.h"
#include "../LogMgr/LogDesc.h"

#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////
// CTerminalServerMgr - Interface

CTerminalServerMgr::CTerminalServerMgr()
{
#ifdef SAMPLECODE
    m_hStopEvent = CreateEvent(NULL, TRUE, TRUE, NULL);
    PropertyList.Add(_T("SampleProperty"), SampleProperty);
#endif
    for (int i=0; i<NUM_PORTS; i++)
    {
        pCursor[i] = NULL;
        LogFile[i] = _T("\\FLASH DISK\\SSP.LOG");
        LogTime[i] = TTime::GetCurrentTime();
    };
}

CTerminalServerMgr::~CTerminalServerMgr()
{
#ifdef SAMPLECODE
```

```

        CloseHandle(m_hStopEvent);
    #endif
    for (int i=0; i<NUM_PORTS; i++)
    {
        if (pCursor[i])
            delete pCursor[i];
    };
}

void CTerminalServerMgr::Start()
{
    #ifdef SAMPLECODE
        ResetEvent(m_hStopEvent);
    #endif
}

void CTerminalServerMgr::Stop()
{
    #ifdef SAMPLECODE
        // Stop child thread.
        SetEvent(m_hStopEvent);
    #endif
}

bool CTerminalServerMgr::ReadSetup(CXMLSetup& XMLSetup)
{
    #ifdef SAMPLECODE
        CXMLElement XMLToken;
        XMLSetup.ReadNextToken(XMLToken);
        CString Property = XMLToken.GetText();
        while (XMLToken.nType == CXMLElement::START_TAG)
        {
            TProperty* pValue;
            CString Tag = XMLToken.GetText();
            if (Tag == _T("General"))
            {
                XMLSetup.ReadNextToken(XMLToken);

                while (XMLToken.nType == CXMLElement::ATTR_NAME)
                {
                    Property = XMLToken.GetText();
                    XMLSetup.ReadNextToken(XMLToken);
                    CString str = XMLToken.GetText();
                    if ((Index >= 0) && PropertyList.Lookup(Property, pValue))
                    {
                        int idx = str.ReverseFind(TCHAR('#'));
                        if (idx != -1)
                        {
                            int iType = _ttoi(str.Right(str.GetLength() - idx - 1));
                            str = str.Left(idx);
                            pValue->SetType(iType);
                            if (pValue->IsInteger())
                                *pValue = _ttoi(str);
                            else if (pValue->IsDouble())
                                *pValue = _tctod(str, NULL);
                            else
                                *pValue = str;
                        }
                        else
                            *pValue = str;
                    }
                    else
                        *pValue = str;
                }
            }
            else
            {
                // The attribute is unrecognized and the module does not support dynamic
                // property creation, so discard the property.
                CString out;
                out.Format(_T("CTerminalServerMgr::ReadSetup eating unknown token: %s."),
                    Property);
            }
        }
    #endif
}

```

```

        Report.Warning(out);
        XMLSetup.ReadNextToken(XMLToken);
    }
    //read the next
    XMLSetup.ReadNextToken(XMLToken);
};
};
XMLSetup.ReadNextToken(XMLToken);
}
#endif
return true;
}

bool CTerminalServerMgr::WriteSetup(CXMLSetup& XMLSetup)
{
#ifdef SAMPLECODE
    // Open TerminalServerMgr entry.
    XMLSetup.WriteStartTag(_T("<TerminalServerMgr>"));

    CString Str;
    TProperty* Property;

    XMLSetup.WriteStartTag(_T("<General>"));
    POSITION Pos = PropertyList.GetStartPosition();
    while (PropertyList.GetNext(Pos, Str, Property))
        if ((Property->Attributes & TProperty::NOSAVE) == 0)
        {
            CString out;
            out.Format(_T("%s = \"%s#%d\"", Str, Property->AsLPCTSTR(), Property->GetType()));
            XMLSetup.WriteText(out);
        };
    XMLSetup.WriteEndTag(_T(">"));

    // Close outermost entry.
    XMLSetup.WriteEndTag(_T("</TerminalServerMgr>"));
#endif

    return true;
}

/*
Here's where we do the work of generating a custom report.
In this case we are just setting things up for a custom command line.

For anything that might take more then a minute, we'd want to create a seperate thread
for each report that needs to be generated, and immediately return true from this function.
*/
bool CTerminalServerMgr::TerminalServerFunction(TUserGroup Group, LPCTSTR ComPort,
        LPCTSTR UserName)
{
    CSocketComm Port;

    Port.SetCommPort(ComPort);

    // Open port.
    if (Port.OpenComm())
    {
        // Here we tell remote which commands we are going to support.
        // We can intercept all commands including the builtin ones by placing an asterisk
        // _T("*,get,...") at the begging of the command list.
        Port.SetExtendedCommands(_T(",get,read,list,logfile,logtime,logdate,help,");
        CString Txt;
        // Let the user know the terminal server is active.
        Txt.Format(_T("Hello %s, Terminal Server is Active. Use Help for a list of ")
            _T("commands including extended ones.\r\n"), UserName);
        Port.PutStr(Txt);
        // Force a command prompt to be displayed
        Port.SetComOptions(ENABLECOMMANDPROMPT, ENABLECOMMANDPROMPT);
        Port.CloseComm();
    }
}

```

```

        return true; // This will cause Remote to keep the command line running
    }
    else
    {
        Report.Warning(_T("TerminalServerFunction(): Failed to open communications port."));
    }
    return false; // this will cause Remote to hangup and get ready for a new user.
};

/*
Here's the function which is called whenever a custom command is entered at Remote.exe's
command line by the user. Basically we open the com port (via sockets), parse the command,
and output any response we wish the user to see.
*/

bool CTerminalServerMgr::TerminalServerParser(TUserGroup Group, LPCTSTR ComPort,
        LPCTSTR UserName, LPCTSTR Command)
{
    int PortIndex = ComPort[3] - '1';
    if ((PortIndex < 0) || (PortIndex >= NUM_PORTS))
        return true;

    CSocketComm Port;

    Port.SetCommPort(ComPort);

    // Open port.
    if (Port.OpenComm())
    {
        // Parse the Command, in to the Lower case root of the command, and the Parameters
        CString Cmd = Command;
        CString CmdRoot;
        CString Parm;
        int SpacePos;
        SpacePos = Cmd.Find(' ');
        if (SpacePos <= 0)
            CmdRoot = Cmd;
        else
        {
            CmdRoot = Cmd.Left(SpacePos);
            Parm = Cmd.Mid(SpacePos+1);
        };

        CString LowerCmdRoot = CmdRoot;
        LowerCmdRoot.MakeLower();

        // Parse each of the commands we support
        if (Cmd.CompareNoCase(_T("get")) == 0) // GET COMMAND
        {
            // We need a cursor to walk thru the log, do we have one already?
            if (pCursor[PortIndex] == NULL)
            {
                // No, let's see if we can find a match for the requested logfile and make a cursor
                CLogDesc* pLog = LogList.GetByName(LogFile[PortIndex]);
                if (pLog)
                {
                    pCursor[PortIndex] = new LOGCURSOR(pLog->log);
                    // If we have don't have a valid time just go to the top of the log,
                    // otherwise search for the specified date and time
                    if (LogTime[PortIndex].GetTime() == 0)
                        pCursor[PortIndex]->GotoTop();
                    else
                        pCursor[PortIndex]->Search(LogTime[PortIndex].GetTime());
                }
            };
            if (pCursor[PortIndex])
            {
                INT64 Time64;
                UINT8 Quality;
                TCHAR Sensor[128];

```



```

TCHAR Units[128];
TCHAR Number[128];
TCHAR Qual;
CString Txt;

// Retrieve and display 20 lines from the log file
for(int Lines=1; Lines<=20; Lines++)
{
    if (pCursor[PortIndex]->IsNote())
    {
        // Log notes need to be handled differently from sensor data
        if (pCursor[PortIndex]->ReadNote(Time64, Sensor))
        {
            TTime Time(Time64);
            Txt.Format(_T("%02d-%02d-%4d %02d:%02d:%02d %s\r\n"), Time.GetMonth(),
                Time.GetDay(), Time.GetYear(), Time.GetHour(), Time.GetMinute(),
                Time.GetSecond(), Sensor);

            // Send the log note out the com port to the user
            Port.PutStr(Txt);
        }
    }
    else if (pCursor[PortIndex]->ReadSensor(Time64, Sensor, Units, Quality, Number))
    {
        // Process sensor data and the quality information
        TTime Time(Time64);
        switch (Quality)
        {
            case CSensorData::GOOD:
                Qual = 'G';
                break;
            case CSensorData::BAD:
                Qual = 'B';
                break;
            case CSensorData::UNDEFINED:
            default:
                Qual = 'U';
                break;
        }
        Txt.Format(_T("%02d-%02d-%4d %02d:%02d:%02d %-12s %10s %s (%c)\r\n"),
            Time.GetMonth(), Time.GetDay(), Time.GetYear(), Time.GetHour(),
            Time.GetMinute(), Time.GetSecond(), Sensor, Number, Units, Qual);

        // Send the sensor data out the com port to the user
        Port.PutStr(Txt);
    };
    if (!pCursor[PortIndex]->MoveNext())
        break;
};
// If we've hit the end of the log, let the user know
if (pCursor[PortIndex]->AtBottom())
    Port.PutStr("[END OF LOG]\r\n");
}
else
{
    // User specified a log file that isn't in the system
    Port.PutStr("Could not find logfile.\r\n");
};
}
else if (LowerCmdRoot.Compare(_T("logfile")) == 0) // LOGFILE COMMAND
{
    // If the user didn't specify a path or extension, tack them on
    if (Parm.Find('\\') < 0)
        Parm = _T("\\Flash Disk\\") + Parm;
    if (Parm.Find('.') < 0)
        Parm += _T(".LOG");

    // Just remember which file was selected
    LogFile[PortIndex] = Parm;

    // Delete any existing Cursor, so the Get command will search for new data

```

```

        if (pCursor[PortIndex])
        {
            delete pCursor[PortIndex];
            pCursor[PortIndex] = NULL;
        };
    }
    else if (LowerCmdRoot.Compare(_T("logtime")) == 0) // LOGTIME COMMAND
    {
        // Parse out the time and store it
        int Hours=0, Minutes=0, Seconds=0;
        if (_stscanf((LPCTSTR) Parm, _T("%d:%d:%d"), &Hours, &Minutes, &Seconds) == 3)
            LogTime[PortIndex] = TTime(LogTime[PortIndex].GetYear(),
                LogTime[PortIndex].GetMonth(), LogTime[PortIndex].GetDay(), Hours,
                Minutes, Seconds);
        else
            LogTime[PortIndex] = 0;
        if (pCursor[PortIndex])
        {
            delete pCursor[PortIndex];
            pCursor[PortIndex] = NULL;
        };
    }
    else if (LowerCmdRoot.Compare(_T("logdate")) == 0) // LOGDATE COMMAND
    {
        // Parse out the date and store it
        int Month=1, Day=1, Year=2000;
        if (_stscanf((LPCTSTR) Parm, _T("%d-%d-%d"), &Month, &Day, &Year) == 3)
        {
            if (Year < 99)
                Year += 2000;
            LogTime[PortIndex] = TTime(Year, Month, Day, LogTime[PortIndex].GetHour(),
                LogTime[PortIndex].GetMinute(), LogTime[PortIndex].GetSecond());
        }
        else
            LogTime[PortIndex] = 0;
        if (pCursor[PortIndex])
        {
            delete pCursor[PortIndex];
            pCursor[PortIndex] = NULL;
        };
    }
    else if (LowerCmdRoot.Compare(_T("list")) == 0) // LIST COMMAND
    {
        // Go thru the list of every module in the system
        for (int i = 0; i<Engine.ModuleList.GetSize(); i++)
        {
            TModule& m = Engine.ModuleList.GetAt(i);
            TProperty* p;
            // We're looking for Log Modules
            if ((m.GetModuleType() == TModule::LOGGING) && m.GetProperty(_T("SensorName"), p))
            {
                // If the user wanted only a certain sensor, then check the SensorName
                // property and skip if this isn't the one requested.
                if (!Parm.IsEmpty() && (Parm.CompareNoCase(p->AsCString()) != 0))
                    continue; // Skip this sensor, it wasn't requested
                TCHAR Qual;
                CSensorData Data;
                // Retrieve the last data stored in to this log module
                m.Get(2, Data);
                switch (Data.Quality)
                {
                    case CSensorData::GOOD:
                        Qual = 'G';
                        break;
                    case CSensorData::BAD:
                        Qual = 'B';
                        break;
                    case CSensorData::UNDEFINED:
                    default:
                        Qual = 'U';
                        break;
                }
            }
        }
    }
}

```

```

    }
    CString Txt;
    // Format up the data and display to the user
    Txt.Format(_T("%-12s %10s %s (%c)\r\n"), p->AsLPCTSTR(),
        Data.Data.AsLPCTSTR(), Data.Units, Qual);
    Port.PutStr(Txt);
}
}
}
else if (LowerCmdRoot.Compare(_T("read")) == 0) // READ COMMAND
{
    // Go thru the list of every module in the system
    for (int i = 0; i < Engine.ModuleList.GetSize(); i++)
    {
        TModule* m = &Engine.ModuleList.GetAt(i);
        TProperty* p;
        // We're looking for Log Modules
        if ((m->GetModuleType() == TModule::LOGGING) && m->GetProperty(_T("SensorName"), p))
        {
            // If the user wanted only a certain sensor, then check the SensorName
            // property and skip if this isn't the one requested.
            if (!Parm.IsEmpty() && (Parm.CompareNoCase(p->AsCString()) != 0))
                continue; // Skip this sensor, it wasn't requested
            // Take a peek at the module just to the left of this log module in the setup
            TNode& Node = m->Inputs[2];
            // Is it a Measure or Average module? (Note: some other types of processing
            // can't be handled like VectAvg)
            while ((Node.Module->Name == _T("Measure")) ||
                (Node.Module->Name == _T("Average")))
            {
                // Yes it is, so instead of trying to measure this, move on to module to left
                m = Node.Module;
                Node = Node.Module->Inputs[2];
            };
            // Now we can pull data from the module to cause a reading to be taken
            Node.Module->Pull(TTime::GetCurrentTime(), true);
            TCHAR Qual;
            CSensorData Data;
            // Now retrieve the data measured and display it
            m->GetInputData(2, Data);
            switch (Data.Quality)
            {
                case CSensorData::GOOD:
                    Qual = 'G';
                    break;
                case CSensorData::BAD:
                    Qual = 'B';
                    break;
                case CSensorData::UNDEFINED:
                default:
                    Qual = 'U';
                    break;
            }
            CString Txt;
            Txt.Format(_T("%-12s %10s %s (%c)\r\n"), p->AsLPCTSTR(),
                Data.Data.AsLPCTSTR(), Data.Units, Qual);
            Port.PutStr(Txt);
        }
    }
}
else if (LowerCmdRoot.Compare(_T("help")) == 0) // HELP COMMAND
{
    Port.PutStr("[[[ STANDARD COMMANDS ]]]\r\n");
    // Here we use RunCommand to have remote display it's standard help screen.
    Port.RunCommand(_T("\rhelph")); // the \r at the beginning prevents recursion here
    Port.PutStr("[[[ EXTENDED COMMANDS ]]]\r\n");
    Port.PutStr("GET\t\t\tRetrieves a screen full of log data\r\n");
    Port.PutStr("LOGFILE file\t\tSets the log file to use for Get\r\n");
    Port.PutStr("LOGTIME hh:mm:ss\tSets the start time for Get\r\n");
    Port.PutStr("LOGDATE mm-dd-yyyy\tSets the start date for Get\r\n");
    Port.PutStr("LIST [sensor]\t\tShows the last value logged for a sensor\r\n");
}
}

```

```

        Port.PutStr("READ [sensor]\\t\\tMeasures and displays current sensor reading\\r\\n");
    }
    else // Program error. An unimplemented command was ran.
    {
        CString Txt;
        Txt.Format(_T("Hello %s! Unimplemented custom command: %s.\\r\\n"), UserName, Command);
        Port.PutStr(Txt);
    };
    // When we're all done, we close the port.
    Port.CloseComm();
}
else
{
    Report.Warning(_T("TerminalServerParser(): Failed to open communications port.));
}
return true; // this will cause Remote to emit a command prompt
};

// The global alarm manager object.
CTerminalServerMgr TerminalServerMgr;

////////////////////////////////////
// App Init and Exit Callbacks

extern "C" __declspec(dllexport) void AppInit()
{
    TerminalServerMgr.Start();
}

extern "C" __declspec(dllexport) void AppExit()
{
    TerminalServerMgr.Stop();
}

////////////////////////////////////
// Setup Callbacks

extern "C" __declspec(dllexport) bool ReadSetup(CXMLSetup& XMLSetup)
{
    if (!TerminalServerMgr.ReadSetup(XMLSetup))
    {
        Report.Warning(_T("Error interpreting TerminalServer configuration from setup.));
        return false;
    }
    else
        return true;
}

extern "C" __declspec(dllexport) bool WriteSetup(CXMLSetup& XMLSetup)
{
    if (!TerminalServerMgr.WriteSetup(XMLSetup))
    {
        Report.Warning(_T("Error writing TerminalServer configuration to setup.));
        return false;
    }
    else
        return true;
}

extern "C" __declspec(dllexport) LPCTSTR GetSetupTag()
{
    return _T("TerminalServerMgr");
}

```

### 6.1.3 TerminalServerMgr.h

Header file for TerminalServerMgr.cpp.

```
// TerminalServerMgr.h : header file for TerminalServerMgr of Alarm.dll
//

#ifndef _TerminalServer_MGR_INCLUDED_
#define _TerminalServer_MGR_INCLUDED_

#ifdef _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "../Engine/XMLSetup.h"
#include "../Engine/Engine.h"
#include "../Logger/Logger.h"

const NUM_PORTS = 4;

////////////////////////////////////
// CTerminalServerMgr
//
class CTerminalServerMgr
{
    //////////////////////////////////
    // Interface
    //////////////////////////////////
public:

    // Constructors and destructor.
    CTerminalServerMgr();
    virtual ~CTerminalServerMgr();

    // Start alarm processing.
    void Start();

    // Stop alarm processing.
    void Stop();

    // Read configuration data from setup.
    bool ReadSetup(CXMLSetup& XMLSetup);

    // Write configuration data to setup.
    bool WriteSetup(CXMLSetup& XMLSetup);

    bool TerminalServerFunction(TUserGroup Group, LPCTSTR ComPort, LPCTSTR UserName);
    bool TerminalServerParser(TUserGroup Group, LPCTSTR ComPort, LPCTSTR UserName, LPCTSTR
Command);

    //////////////////////////////////
    // Implementation
    //////////////////////////////////
protected:
    friend class CTerminalServerControlPanelEntry;
    TPropertyList PropertyList;
    LOGCURSOR* pCursor[NUM_PORTS];
    CString LogFile[NUM_PORTS];
    TTime LogTime[NUM_PORTS];

#ifdef SAMPLECODE
    TProperty SampleProperty;
#endif

#ifdef SAMPLECODE
    // Event handles used to coordinate thread termination.

```

```

        HANDLE m_hStopEvent;
    #endif
};

//The global alarm mgr object
extern CTerminalServerMgr TerminalServerMgr;

#endif // !defined(_TerminalServer_MGR_INCLUDED_)

```

#### 6.1.4 TerminalServerControlPanelEntry.cpp

This demo only includes the skeleton of how to implement a control panel entry for an SLL. It could be enhanced to enable/disable various features or options of the Terminal Server.

```

// TerminalServerControlPanelEntry.cpp : Defines control panel entry class.
//

#include "stdafx.h"

#include "TerminalServerMgr.h"
#include "TerminalServer.h"
#include "TerminalServerControlPanelEntry.h"
#include "Resource.h"
#include "../Engine/Engine.h"
#include "../Utils/Report.h"
#include "../Utils/ResourceKey.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

////////////////////////////////////
// CTerminalServerControlPanelEntry

CTerminalServerControlPanelEntry::CTerminalServerControlPanelEntry()
{
    m_pTree = NULL;
    m_hRootItem = NULL;
}

bool CTerminalServerControlPanelEntry::EditItem(CWnd* pParent, HTREEITEM hItem, CTreeCtrl* pTree)
{
    // This method is called when the user selects edit with a node of the
    // tree selected. Return true only when the tree should be rebuilt as a
    // result of handling the incoming item (e.g., when the text of an item
    // changes, or a new sub-item is added/deleted, etc.).
    // Set member variable m_bSaveNeeded to true when changes are made that
    // need to be saved to the setup file.
    TResourceKey key(TerminalServerSLL);

    // Prompt the user with a dialog of configuration parameters if the handle passed
    // in belongs to TerminalServer.
    if (hItem == m_hRootItem)
    {
        #ifdef SAMPLECODE
        // Show dialog with configuration information
        CTerminalServerDlg Dlg(pParent);

        if (IDOK == Dlg.DoModal())
        {
            if (PromptEngineStop())
            {

```

```

        // Incorporate changes.

    }

    // Set flag to cause automatic setup save.
    m_bSaveNeeded = true;
}
#endif
}
else
{
    // Handle sub items if used.

}

return false;
}

void CTerminalServerControlPanelEntry::InsertItems(CTreeCtrl* pTree)
{
    // Insert the TerminalServer branch into the control panel tree.
    if (Engine.LockGUI())
    {
        m_pTree = pTree;
        m_hRootItem = pTree->InsertItem(_T("TerminalServer"));
        // Insert sub-items if they exist
        // m_hSubItem = pTree->InsertItem(_T("TerminalServer SubItem"), m_hRootItem);

        Engine.UnlockGUI();
    }
}

extern "C" __declspec(dllexport) void CreateControlPanelEntry(CControlPanelEntry** ppEntry)
{
    CTerminalServerControlPanelEntry* pEntry = new CTerminalServerControlPanelEntry();
    *ppEntry = dynamic_cast<CControlPanelEntry*>(pEntry);
}

```

## 6.1.5 TerminalServerControlPanelEntry.h

The Header file for TerminalServerControlPanelEntry.h.

```

#ifndef _TerminalServerControlPanelEntry_H_
#define _TerminalServerControlPanelEntry_H_

#include "../engine/module.h"          // For CControlPanelEntry

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

1. An instance of the class CTerminalServerControlPanelEntry is used to manage
   TerminalServer configuration parameters from the control panel.

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

// CTerminalServerControlPanelEntry

class CTerminalServerControlPanelEntry : public CControlPanelEntry
{
    // Constructors and destructor.
public:
    CTerminalServerControlPanelEntry();
    virtual ~CTerminalServerControlPanelEntry() {};

    // Attributes

```

```

private:

    // Overrides
public:
    virtual void EditItem(CWnd* pParent, HTREEITEM hItem, CTreeCtrl* pTree);
    virtual void InsertItems(CTreeCtrl* pTree);

    // Methods
public:
private:
    // HTREEITEM m_hSubItem;
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#endif // _TerminalServerControlPanelEntry_H_

```

## 6.2 Threads Example

This example demonstrates the correct way to start and stop threads created by a DLL.

```

HANDLE g_hStopEvent = NULL;
HANDLE g_hThread = NULL;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Thread function

UINT Thread(LPVOID pParam)
{
    // Continue work loop until signaled to stop.
    while (WaitForSingleObject(g_hStopEvent, 0) == WAIT_TIMEOUT)
    {
        // Typically, sleep until time to do work. Sleep is required
        // at some point in order to give other threads time to run.
        // Sleep may be interrupted by an event (engine start or some
        // other signal) by using either WaitForSingleObject() or
        // WaitForMultipleObjects().
        DWORD dwSleep = 1000;
        Sleep(dwSleep);

        // Do work...
    }

    return 0;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// App Init and Exit Callbacks

extern "C" __declspec(dllexport) void AppInit()
{
    // Called when Xpert boots. Objective is to start thread.
    g_hStopEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    // Reset event used by thread to know when to stop.
    ResetEvent(g_hStopEvent);

    // Spawn thread.
    CWinThread* pThread = AfxBeginThread(Thread, NULL);
    if (pThread)
        g_hThread = pThread->m_hThread;
}

extern "C" __declspec(dllexport) void AppExit()
{

```



```

// Called when Xpert shuts-down. Objective is to stop thread.
if (g_hThread)
{
    // Signal thread to quit.
    SetEvent(g_hStopEvent);

    // Wait for thread to complete.
    WaitForSingleObject(g_hThread, INFINITE);
    g_hThread = NULL;
}
CloseHandle(g_hStopEvent);
}

```

## 6.3 Engine API Examples

The following examples demonstrate using various Engine API functions.

### 6.3.1 hStartEvent and hStopEvent

1. Start and stop events can be used to detect engine start and stop, as may be necessary outside scope of a setup block (which uses Initialize() and Stop()).
2. Example of event wait:

```

if (WaitForSingleObject(Engine.hStopEvent, dwWait) != WAIT_TIMEOUT)
    // Detected engine stop!

```

### 6.3.2 ModuleList

1. List of all modules defined by the current setup.
2. Example of access:

```

// The following function searches all setup blocks for a block named
// strBlockName, having a property named strPropName. If such a block is
// found, pProp is set to point to the property, iBlock is set to the
// index of the block within ModuleList, and true is returned. Otherwise,
// false is returned.
bool FindProperty(const CString& strBlockName, const CString& strPropName,
    TProperty* pProp, int& iBlock)
{
    for (int i = 0; i < Engine.ModuleList.GetSize(); i++)
    {
        TModule* pMod = &Engine.ModuleList.GetAt(i);
        if (!pMod->Name.Compare(strBlockName))
        {
            iBlock = i;
            TPropertyEnum penum(*pMod);
            CString strStoredName;
            TProperty* pStoredProp;
            while (true == penum.Get(strStoredName, pStoredProp))
            {
                if (strStoredName == strPropName)
                {
                    pProp = pStoredProp;
                    return true;
                }
            }
        }
    }
}

```

```

    }
}

// Not found if we reach here.
return false;
}

```

### 6.3.3 Exported Engine Functions

1. Interface dialogs provide easy way to get data from user in touchscreen environment.
2. The following example shows how to get an integer from the user, while validating the integer's range. The user is allowed to cancel the edit at any time.

```

void OnBtnChangeInt()
{
    UpdateData(TRUE);
    int iNewInt = m_iStoredInt;
    if (IDOK == ChangeNumberDlgInt(this, iNewInt))
    {
        while (1 > iNewInt || iNewInt > 999)
        {
            AfxMessageBox(_T("Please enter an integer between 1 and 999."));
            if (IDOK != ChangeNumberDlgInt(this, iNewInt))
                return;
        }
        m_iStoredInt = iNewInt;
        UpdateData(FALSE);
    }
}

```

## 6.4 Analog I/O

1. Initialize device for making required measurement in `Inititalize()`. This ensures manual measurements while recording is on will work as expected.
2. Initialize device again prior to making measurement in `Execute()`. This ensures any device reconfiguration does not corrupt measurement.
3. Table in Analog I/O section of this document defines what config parameters need to be defined for each type of measurement.
4. Solar Radiation Sensor Example

```

CSolarRad::CSolarRad() : TModule(_T("SolRad"), DefLibSLL.hResource)
{
    AddProperty(_T("AIOModule"), m_propAIOModule = 1);
    AddProperty(_T("AIOChannel"), m_propAIOChannel = 0);
    AddProperty(_T("Units"), m_propUnits = _T("W/m2"));
    AddProperty(_T("Calibration"), m_propCalibration = 1.);

    LastData.Data = 0;
    LastData.TimeScheduled = TTime::GetCurrentTime();
    LastData.TimeActual = TTime::GetCurrentTime();
    LastData.Quality = CSensorData::UNDEFINED;
    LastData.Units = m_propUnits;
    LastData.SensorID = Function;
    LastData.DeviceTime = 0;
}

CSolarRad::~CSolarRad()

```

```

{
    // Release channel.
    Engine.IOModList.ClearChannelInUse(ANALOG, m_propAIOModule, m_propAIOChannel);
}

void CSolarRad::ShowProperties(CWnd* pParent)
{
    TResourceKey key(DefLibSLL);
    CSolarRadDlg dlg(pParent);

    // Initialize dialog data.
    dlg.m_iIOMod = m_propAIOModule;
    dlg.m_iChannel = m_propAIOChannel;
    dlg.m_strUnits = m_propUnits;
    dlg.m_rCal = m_propCalibration;
    if (dlg.DoModal() == IDOK)
    {
        // Incorporate data from dialog.
        Engine.IOModList.ClearChannelInUse(ANALOG, m_propAIOModule, m_propAIOChannel);
        m_propAIOModule = dlg.m_iIOMod;
        m_propAIOChannel = dlg.m_iChannel;
        m_propUnits = dlg.m_strUnits;
        m_propCalibration = dlg.m_rCal;
        Engine.IOModList.SetChannelInUse(ANALOG, m_propAIOModule, m_propAIOChannel);
    }
}

void CSolarRad::Execute(TTime tScheduled)
{
    // Initialize instance of sensor data.
    CSensorData RawData = LastData;
    RawData.Quality = CSensorData::BAD;
    RawData.Data = 0;
    RawData.TimeScheduled = tScheduled;
    RawData.TimeActual = TTime::GetCurrentTime();

    // Get a pointer to the I2C device.
    AnalogIO* pAnalogIO = Engine.IOModList.GetAnalogIO(m_propAIOModule);
    if (!pAnalogIO)
    {
        Report.Error(_T("CSolarRad::Execute: Failed to get analog device."));
    }
    else
    {
        // Measure voltage.
        double rVoltage;
        pAnalogIO->LockConfig();
        pAnalogIO->SetConfigurationGain(m_propAIOChannel, 1);
        pAnalogIO->SetPolyAdjust(m_propAIOChannel, TRUE);
        pAnalogIO->SetConfigurationExcitationHoldOff(m_propAIOChannel);
        pAnalogIO->SetConfigurationSingleEnded(m_propAIOChannel);
        pAnalogIO->SetExcitationVoltage(m_propAIOChannel, 5);
        pAnalogIO->SetExcitationChannel(m_propAIOChannel, 0);
        pAnalogIO->SetExcitationVoltageOff(m_propAIOChannel);
        pAnalogIO->SetPeriod(m_propAIOChannel, 10);
        pAnalogIO->SetSamples(m_propAIOChannel, 1);
        I2CCODE code = pAnalogIO->SingleVoltageReading(m_propAIOChannel, rVoltage);
        pAnalogIO->FreeConfig();
        if (code != I2C_OK)
        {
            Report.Error(_T("CSolarRad::Execute: Failed to get data from IO device."));
        }
        else
        {
            // Compute radiation.
            RawData.Data = rVoltage * 1000.0 / m_propCalibration.AsDouble();
            RawData.Quality = CSensorData::GOOD;
        }
    }

    // Buffer output data.
}

```

```

    LockData();
    LastData = RawData;
    UnlockData();
}

```

## 6.5 Digital I/O – Tipping Bucket Example

```

CTippingBucket::CTippingBucket() : TModule(_T("TipBckt"), DefLibSLL.hResource)
{
    AddProperty(_T("Channel"),          Channel = 0);
    AddProperty(_T("IODeviceName"),     ModName = 1);
    AddProperty(_T("FilterValue"),      FilterValue = 3);
    AddProperty(_T("Offset"),           Offset = 0);
}

CTippingBucket::~CTippingBucket()
{
    Engine.IOModList.ClearChannelInUse(DIGITAL, ModName, Channel);
}

void CTippingBucket::ShowProperties(CWnd* pParent)
{
    TResourceKey key(DefLibSLL);
    CTippingBucketDlg dlg(pParent);

    dlg.m_Channel = Channel;
    dlg.m_FilterValue = FilterValue;
    dlg.m_IOMOD = ModName;
    if (dlg.DoModal() == IDOK)
    {
        // Update engine with regard to what module/channels combo is now in use.
        Engine.IOModList.ClearChannelInUse(DIGITAL, ModName, Channel);
        Channel = dlg.m_Channel;
        FilterValue = dlg.m_FilterValue;
        ModName = dlg.m_IOMOD;
        Engine.IOModList.SetChannelInUse(DIGITAL, ModName, Channel);
    }
}

void CTippingBucket::Initialize()
{
    // Initialize buffered output data.
    LastData.TimeScheduled = TTime::GetCurrentTime();
    LastData.TimeActual = TTime::GetCurrentTime();
    LastData.SensorID = Function;
    LastData.Quality = CSensorData::UNDEFINED;
    LastData.Data = 0;
    LastData.Units = _T("MM");

    DigitalIO* pDigIO = NULL;
    pDigIO = Engine.IOModList.GetDigitalIO(ModName);

    // Configure device channel.
    pDigIO->SetAsCounter(Channel);
    pDigIO->InvertIO(Channel);
    pDigIO->ConfigureFilters(Channel, FilterValue);
    pDigIO->SetSensitivityLow(Channel);
    pDigIO->StartRequest();
}

void CTippingBucket::Execute(TTime tScheduled)
{
    // Initialize sensor data.
    CSensorData Data = LastData;
    sdTips.TimeScheduled = tScheduled;
    sdTips.TimeActual = TTime::GetCurrentTime();
    sdTips.Quality = CSensorData::BAD;
    sdTips.Data = 0.0;
}

```

```

// Get io module object.
DigitalIO* pDigIO = Engine.IOModList.GetDigitalIO(ModName);
if (pDigIO)
{
    UINT32 nTips;
    if (pDigIO->ReadCount(Channel, nTips) == I2C_OK)
    {
        sdTips.Data = nTips;
        sdTips.Quality = CSensorData::GOOD;
    }
}

// Buffer output data.
LockData();
LastData = sdTips;
UnlockData();
}

void CTippingBucket::Stop()
{
    DigitalIO* pIO = Engine.IOModList.GetDigitalIO(ModName);
    if (pIO)
        pIO->StopRequest();
    else
        Report.Error(_T("Failed to get handle to IO module."));
}

bool CTippingBucket::Calibrate()
{
    // TModule override called when user invokes calibration from View Sensors.
    double dOrigVal, dNewVal;
    dOrigVal = dNewVal = LastData.Data.AsDouble();
    if (IDOK == ChangeNumberDlgReal(NULL, dNewVal, _T("Enter current value")))
    {
        if (Calibrate(dOrigVal, dNewVal))
            return true;
    }
    return false;
}

bool CTippingBucket::I2CCalibrate()
{
    // TModule override called when user invokes calibration from Xlite display.
    DisplayIO::EditStatus Status;
    double OrigVal, NewVal;
    DisplayIO* pDisp = Engine.IOModList.GetDisplayIO(1);
    if (pDisp)
    {
        OrigVal = NewVal = LastData.Data.AsDouble();
        Status = pDisp->EditFloat(NewVal, _T("Cur.Val"), true, -1000, 5000);
        if (Status == DisplayIO::EDIT_OK)
        {
            if (Calibrate(OrigVal, NewVal))
                return true;
        }
    }
    return false;
}

```

## 6.6 SDI API – Example

```
void CSDI::Execute(TTime tScheduled)
{
    // Initialize raw output data.
    CSensorData sdRawData;
    sdRawData.Data = 0;
    sdRawData.TimeScheduled = tScheduled;
    sdRawData.TimeActual = TTime::GetCurrentTime();
    sdRawData.Quality = CSensorData::BAD;
    sdRawData.Units = m_propUnits;

    // Output command and get return data.
    int iRet;
    double dData[MAX_SDI_MEASUREMENTS];           // Results stored here.
    DWORD dwNumMeasurements;                       // Number of measurements taken.
    DWORD dwSDITimeout;                            // Max time in milliseconds to wait for result.
    iRet = CollectData(_T("OM!"), dData, MAX_SDI_MEASUREMENTS * sizeof(double),
        &dwNumMeasurements, dwSDITimeout, FALSE);
    if (!iRet)
    {
        sdRawData.Data = dData[0];
        sdRawData.TimeActual = TTime::GetCurrentTime();
        sdRawData.Quality = CSensorData::GOOD;
    }

    // Buffer output data.
    LockData();
    LastData = sdRawData[0];
    UnlockData();
}
```

## 6.7 Report Management API

Example of “hooking” into reporting mechanism.

```
void MyMessageHook(LPVOID pInfo, int iLevel, LPCTSTR szMsg)
{
    // pInfo contains info provided during Hook().
    // iLevel indicates level/type (status, error, debug, etc.) of message.
    // szMsg contains message.
}

void HookUp()
{
    // Tell report manager to give me all non-debug messages.
    Report.Hook(MyMessageHook, NULL);
    Report.SetFilter(MyMessageHook, TReport::msg_All & ~TReport::msg_Debug);
}

void UnHook()
{
    Report.UnHook(MyMessageHook);
}
```

## 6.8 Serial Communications

Example of using CSerialComm class in a sensor block’s execute method. The presumption is the sensor is controlled serially.

```
// Header contains the following:
// CSerialComm m_port;
// TProperty m_propComPort;
```

```

void CMySerialSensor::Execute(TTime tScheduled)
{
    // <raw output data initialization not shown>

    // Make measurement.
    m_port.SetCommPort( T("COM1:"));
    m_port.SetConfiguration(CBR_4800, 8, NOPARITY, ONESTOPBIT, FALSE);
    m_port.OpenComm();
    m_port.FlushInput();

    COMMTIMEOUTS CommTimeouts;
    CommTimeouts.ReadIntervalTimeout = 0;
    CommTimeouts.ReadTotalTimeoutConstant = 300;
    CommTimeouts.ReadTotalTimeoutMultiplier = 1;
    m_port.SetTimeouts(CommTimeouts);

    // Signal sensor to return measurement data.
    m_port.PutStr("O\r");

    TCHAR strData[BUF_LEN];
    int iNumChars = m_port.GetStr(strData, BUF_LEN - 1);
    {
        // strData now contains measurement, process as required.
    }

    m_port.CloseComm();

    // <Buffer of output data not shown>
}

```

**Example of converting multi-byte strings to wide-character strings.**

```

// pMBS points to multi-byte string (e.g., char* pMBS = "string").
// iNumChars contains number of characters pointed to by pMBS.
TCHAR* psz = new TCHAR[iNumChars + 1];
mbstowcs(psz, (const char*)pMBS, iNumChars);
psz[iNumChars] = _T('\0');
// use psz;
delete[] psz;

```

## 6.9 Remote Communications using SSP

Example of using CSocketComm class and RemoteRequest() in order to dialout to a station over a modem and send an SSP alarm message. This sample can be built-in to and executed from an SLL created with the SDK. It will send an SSP Alarm message once a minute by dialing out the provided phone number once a minute. It will keep doing this until Xpert application is shutdown. The default phone number of 555-5555 and port of COM2: should be checked and/or changed.

The sample program automatically detects how the port was configured in Remote and will only dialout if an actual Modem or Voice modem was specified. If it's a radio or direct connect port the message is transmitted without taking the extra steps of dialing out and making a connection.

This demo requires v1.4 or later of Remote.exe and the SDK, as RemoteRequest, and ENABLESSPPARSER do not exist in earlier revisions.

```

#include "../Engine/Engine.h"
#include "../Utils/Report.h"
#include "../Utils/Power.h"

// Various parameters used by the program that may be changed for your h/w
CWinThread* ModemThread;
const TCHAR PhoneNumber[] = _T("555-5555");
const TCHAR ComPort[] = _T("COM2:");
const int PortIndex = 2; // COM2:

```

```

const TCHAR ToPath[] = _T("");           // Send message to any station listening
const int NumRetries = 3;                 // Try a message 3 times
const int AckDelay = 10000;               // Wait up to 10 seconds for an ACK
const int AlarmInterval = 60000;          // How often to send alarms

// Function to build up an SSP Alarm Message by iterating thru the Tag List
bool BuildAlarmMessage(TSSPData& Data)
{
    CString Name;
    CTag* Tag;
    bool Result = false;

    Data.Clear();
    Data.Write(OpAlarm);

    Engine.LockTags();
    for (POSITION P=Engine.TagList.GetStartPosition(); P != NULL; )
    {
        Engine.TagList.GetNextAssoc(P, Name, Tag);
        if (Tag->IsCurDataTag())
        {
            TValue Value;
            int Type;
            CSensorData::QualityType Quality;
            Data.Write(Name);                // Name of tag being sent
            Data.Write((UINT8) 2);           // Always just 2 values for Alarm or CurData
            Tag->GetTag(0, Type, Value);      // Retrieve the data value of the tag
            Data.Write((UINT8) 0);           // Indicate first value (1 of 2)
            Data.Write((UINT8) dt_real);     // Indicate data type
            Data.Write(Value.AsDouble());    // Add the data value to the packet
            Tag->GetTag(1, Type, Value, Quality); // Retrieve alarm status of the tag
            Data.Write((UINT8) 1);           // Indicate second value of the tag (2 of 2)
            Data.Write((UINT8) dt_alarm);    // Indicate data type
            Data.Write((DWORD) Value.AsInteger()); // Add the alarm value to the packet
            Result = true;
        }
    };
    Engine.UnlockTags();
    return Result;
};

// The top-level thread which manages dialing out, formatting, and sending the SSP alarm
UINT RemoteThread(LPVOID pParam)
{
    CSocketComm Port;
    Port.SetCommPort(ComPort);
    Port.SetConfiguration(CBR_9600, 8, NOPARITY, ONESTOPBIT, false);
    COMMTIMEOUTS CommTimeouts;
    CommTimeouts.ReadIntervalTimeout = 0;
    CommTimeouts.ReadTotalTimeoutMultiplier = 0;
    CommTimeouts.ReadTotalTimeoutConstant = 90000; // Allow 90 second timeout
    CommTimeouts.WriteTotalTimeoutMultiplier = 0;
    CommTimeouts.WriteTotalTimeoutConstant = 0;
    Port.SetTimeouts(CommTimeouts);

    TSSPData SendData;
    TSSPData ReplyData;
    CString ReplyTo;
    CString ReplyFrom;
    DWORD ReplySeqNum;
    // Matches is an array of possible messages we expect to be sent back
    OPMATCH Matches[2] = {OPMATCH(0, OpAck, OpAlarm), OPMATCH(0, OpNak, OpAlarm)};

    bool DialupConnect = false;
    bool VoiceConnect = false;

    // Auto detect if we have a dial-up modem connected
    if (Port.OpenComm()) // Gain access to the port
    {
        int Options = 0;
        Port.GetComOptions(Options);
    }
}

```



```

DialupConnect = (Options & ISMODEM) != 0; // Detect Sutron or Hayes type Modem
VoiceConnect = (Options & ISVOICE) != 0; // Detect Sutron Voice Modem
Port.CloseComm();
};

// Dialout once a minute, exit when the application stops
while (WaitForSingleObject(Engine.GetAbortEvent(), AlarmInterval) == WAIT_TIMEOUT)
{
    if (DialupConnect)
    {
        if (Port.OpenComm()) // Gain access to the port
        {
            Report.Debug(_T("DEMO: Modem port %s opened"), ComPort);
            if (Port.LockComm(INFINITE)) // Request exclusive access
            {
                bool Connected = true;
                Report.Debug(_T("DEMO: Modem port locked"));
                // SSP Processing is off by default after locking a port.
                // We need to turn it on so we can send the alarm.
                Port.EscapeCommFunction(SETDTR); // Enable the modem
                Port.FlushInput();
                if (VoiceConnect)
                {
                    Report.Debug(_T("DEMO: Waiting for VOICE modem power up"));
                    Sleep(4000); // Voice modem requires a long power on delay
                }
                else
                {
                    Sleep(250);
                    // Enable result codes, disable command echo,
                    // enable short form, enable RLSD, enable DTR to hangup,
                    // and finally DIAL the Phone Number
                    // Short form response codes have no line feed in header
                    // and are ASCII numbers as opposed to strings, ("0" instead of "OK")
                    Report.Debug(_T("DEMO: Dialing phone number %s"), PhoneNumber);
                    Port.PutStr("ATQ0E0V0&D0&C1DT");
                    Port.PutStr(PhoneNumber);
                    Port.WaitForTxEmpty('\r');
                    TCHAR Str[32];
                    if (Port.GetStr(Str, 1))
                    {
                        Sleep(250); // Wait for more characters
                        int BytesToRead = Port.NumberBytesInputBuffer();
                        if (BytesToRead > (sizeof(Str) - 2))
                            BytesToRead = sizeof(Str) - 2;
                        if (BytesToRead)
                            Port.GetStr(Str+1, BytesToRead);
                        Report.Debug(_T("DEMO: Modem response: %s"), Str);
                        Sleep(1000);
                    }
                    else
                    {
                        Report.Debug(_T("DEMO: No response from modem"));
                        Connected = false;
                    }
                }
            };
            if (Str[0] == '3') // Check for NO CARRIER response from the modem
                Connected = false;
            DWORD Status = 0;
            Port.GetCommModemStatus(&Status);
            // Carrier Detect (ie RLSD) should be high if the modem connected
            if ((Status & MS_RLSD_ON) == 0)
                Connected = false; // no CD so we're obviously not connected
            if (Connected)
            {
                // Build up the Alarm Message to send
                if (BuildAlarmMessage(SendData))
                {
                    // SSP Processing and Parsing is turned off by default when use LockComm()
                    // but we need to enable them so RemoteRequest() can function
                    Port.SetComOptions(ENABLESSP | ENABLESSPPARSER, ENABLESSP | ENABLESSPPARSER);
                    Port.SetCapture(false); // No need to waste processing power capturing info we
                                           // don't need.
                }
            }
        }
    }
}

```

```

        // Command Remote to send the SSP Request and wait for the response
        int Result = RemoteRequest(PortIndex, ToPath, Engine.StationName,
            GetFlagSeq(), SendData, 2, Matches, ReplyTo, ReplyFrom, ReplySeqNum,
            ReplyData, NumRetries, AckDelay);
        if (Result == 0) // See if it was the first match ie OpAck
        {
            Report.Status(_T("DEMO: Modem alarm message sent, ack received, and ")
                _T("alerts cleared."));
            Engine.ClearAlert();
        }
        else if (Result < 0)
            Report.Warning(_T("DEMO: Timed out sending a Modem alarm ")
                _T("message (code=%d)."), Result);
        else
            Report.Warning(_T("DEMO: Tried to send a Modem alarm message, ")
                _T("but it was rejected (NAKed)."));
    }
    else
        Report.Warning(_T("DEMO: No ComTags to send, please add some to the setup."));
}
else
    Report.Debug(_T("DEMO: Modem Connect failed"));

// We might do the following commented out commands if we were intending to
// communicate directly with the modem some more, but since we're all done, we
// can just UnLock() and Remote will take care of hanging up and restoring the
// settings:

// SetCapture(true);
// SetComOptions(0, ENABLESSP | ENABLESSPPARSER);
// Port.EscapeCommFunction(CLRDTR); // Hangup the modem

Port.UnLockComm();
Report.Debug(_T("DEMO: Modem port unlocked"));
};
Port.CloseComm();
Report.Debug(_T("DEMO: Modem port closed"));
}
}
else // Radio or Direct Connect - No need to even open the com port for this case
{
    Report.Debug(_T("DEMO: Sending alarm to %s"), ComPort);
    // Build up the Alarm Message to send
    if (BuildAlarmMessage(SendData))
    {
        // Command Remote to send the SSP Request and wait for the response
        int Result = RemoteRequest(PortIndex, ToPath, Engine.StationName,
            GetFlagSeq(), SendData, 2, Matches, ReplyTo, ReplyFrom, ReplySeqNum,
            ReplyData, NumRetries, AckDelay);
        if (Result == 0) // See if it was the first match ie OpAck
        {
            Report.Status(_T("DEMO: Alarm message sent, ack received, and alerts cleared."));
            Engine.ClearAlert();
        }
        else if (Result < 0)
            Report.Warning(_T("DEMO: Timed out sending an Alarm message (code=%d)."), Result);
        else
            Report.Warning(_T("DEMO: Tried to send an Alarm message, but it was ")
                _T("rejected (NAKed)."));
    }
    else
        Report.Warning(_T("DEMO: No ComTags to send, please add some to the setup."));
};
};
return 0;
}

////////////////////////////////////
// App Init and Exit Callbacks

```

```
// Modifications to AppInit() necessary to start the thread.
extern "C" __declspec(dllexport) void AppInit()
{
    ModemThread = AfxBeginThread(RemoteThread, NULL);
}

// Modifications to AppExit() necessary to shutdown the thread.
extern "C" __declspec(dllexport) void AppExit()
{
    Report.Debug(_T("DEMO: Stopping Alarm thread"));
    WaitForSingleObject(ModemThread->m_hThread, INFINITE);
    Report.Debug(_T("DEMO: Finished"));
}
```

## 6.10 Log API

1. CLogDesc class is used to create, open, name, and write (append) to a log.
2. LOGCURSOR and LOG classes are used to read from a log
3. Example: The following example searches for a log named “MyLog”. If the log is not found, it is created and a note is appended to it. Once the log is either found or created, its contents are printed as status messages.

```
void FindAndDumpLog(const CString& strName)
{
    // Find log named strName
    CLogDesc* pLog = NULL;
    bool bFound = false;
    POSITION pos;
    for (int i = 0; i < LogList.GetCount(); i++)
    {
        pos = LogList.FindIndex(i);
        pLog = LogList.GetAt(pos);
        if (!strName.Compare(pLog->GetName()))
        {
            bFound = true;
            break;
        }
    }

    // Create the log if it did not exist.
    if (!bFound)
    {
        // Create a log named strName 4k size, wrap on, don't ignore bad data.
        // Log is closed and opened after creating to commit to flash.
        pLog = (CLogDesc*)new CLogDesc(strName, 4096, true, false);
        pLog->Create();
        pLog->log.Close();
        pLog->Open();

        // Add log to list maintained by system.
        LogList.AddHead(pLog);
    }

    // Get cursor into log for reading contents.
    LOGCURSOR Cursor(pLog->log);

    // Data used during read.
    CString strLine;
    INT64 Time;
    TCHAR Note[128];
    UINT8 Quality = 0;
    LVDATAITEM di;
    int nDataLineCount = 0;
    int MaxLines = pLog->log.GetLineCount() + 100;
```

```

// Start reading from top of log (oldest data).
if (Cursor.GotoTop())
{
    // Do until number of lines read test exceeds available, or until
    // MoveNext fails (see below).
    for(;;)
    {
        // Function called to read is different for notes versus data.
        if (Cursor.IsNote())
        {
            Cursor.ReadNote(Time, Note);
            if (Time < 0) Time = 0;
            TTime timestamp(Time);
            wsprintf(di.Time, _T("%02d:%02d:%02d"), timestamp.GetHour(),
                timestamp.GetMinute(), timestamp.GetSecond());
            wsprintf(di.Date, _T("%02d/%02d/%4d"), timestamp.GetMonth(),
                timestamp.GetDay(), timestamp.GetYear());
            strLine.Format(_T("%s,%s,%s,,\r\n"), di.Date, di.Time, Note);
        }
        else if (Cursor.IsRecord()) // Record type of data
        {
            // read data
            Cursor.ReadRecord(Time, di.Sensor, di.Data);

            // format data
            if (Time < 0) Time = 0;
            TTime timestamp(Time);
            wsprintf(di.Time, _T("%02d:%02d:%02d"), timestamp.GetHour(),
                timestamp.GetMinute(), timestamp.GetSecond());
            wsprintf(di.Date, _T("%02d/%02d/%4d"), timestamp.GetMonth(),
                timestamp.GetDay(), timestamp.GetYear());
            strLine.Format(_T("%s,%s,%s,%s\r\n"), di.Date, di.Time, di.Sensor, di.Data);
        }
        else
        {
            Cursor.ReadSensor(Time, di.Sensor, di.Units, Quality, di.Data);
            if (Time < 0) Time = 0;
            TTime timestamp(Time);
            wsprintf(di.Time, _T("%02d:%02d:%02d"), timestamp.GetHour(),
                timestamp.GetMinute(), timestamp.GetSecond());
            wsprintf(di.Date, _T("%02d/%02d/%4d"), timestamp.GetMonth(),
                timestamp.GetDay(), timestamp.GetYear());
            if (Quality == CSensorData::GOOD)
                strcpy(di.Quality, _T("G"));
            else if (Quality == CSensorData::BAD)
                strcpy(di.Quality, _T("B"));
            else if (Quality == CSensorData::UNDEFINED)
                strcpy(di.Quality, _T("U"));

            strLine.Format(_T("%s,%s,%s,%s,%s,%s\r\n"), di.Date, di.Time,
                di.Sensor, di.Data, di.Units, di.Quality);
            Report.Status(strLine);
        }
    }

    // Do until no more to read.
    if (!Cursor.MoveNext() || nDataLineCount++ > MaxLines)
        break;
}
}
}

```

## 6.11 Xlite Display IO

This example shows how an SLL can hook into the Xlite display in order to interact with the user. The Xpert (Xlite) looks for the functions below at startup so that they can be called when necessary as the user manipulates the display. All three functions must be defined.

```
extern "C" __declspec(dllexport) bool IsMainMenu()
```

```

{
    // Return true to indicate that we want this menu to serve as the Xlite's
    // main menu.
    return true;
}

extern "C" _declspec(dllexport) LPCTSTR GetMenuEntry()
{
    // This menu entry will be used if user finds his way into the standard Xlite menu.
    return _T("My Menu (*)");
}

extern "C" _declspec(dllexport) bool DisplayProc(DisplayIO* pDisp)
{
    // The display procedure.
    // Returning true tells the Xlite to continue showing the standard menu.
    // Returning false tells the Xlite to exit the menu entirely.
    if (!pDisp)
    {
        // Warn user through log entry, and don't cause exit of standard Xlite menu.
        Report.Warning(_T("My display procedure recieved NULL display pointer."));
        return true;
    }

    // Look to see if storage card exists.
    DWORD dwAttrib = GetFileAttributes(_T("\\Storage Card"));
    if (dwAttrib & FILE_ATTRIBUTE_DIRECTORY && dwAttrib != 0xFFFFFFFF)
    {
        // Card exists so do something with it...
        ...
    }
    else
    {
        // No storage card is inserted is inserted so tell user to insert it.
        pDisp->Write(_T("Insert storage card!"));
        Sleep(2000);

        // Prompt to exit. If user confirms exit, cause exit of entire menu.
        pDisp->Write(_T("Exit?"));
        if (pDisp->OKCancel())
            return false;
        else
            return true;
    }
}

```