AN1071

IrDA® Standard Stack for Microchip 16-bit and 32-bit Microcontrollers

Author: *Kim Otten*
Microchip Technology Inc.

INTRODUCTION

Infrared communication is a low-cost method of providing wireless, point-to-point communication between two devices. The Infrared Data Association, often referred to as IrDA, was formed in 1994 to develop standard methods for communicating over short-range infrared transmissions. These standards have continued to evolve and gain in popularity. Now, a wide variety of devices implement the IrDA standard specification, including computers, printers, PDAs, cell phones, watches and other instruments.

Microchip's 16-bit and 32-bit microcontrollers are a perfect fit for applications wanting to support IrDA standard communication. These low-cost microcontrollers, with their built-in IrDA standard support, provide an inexpensive solution with plenty of computing power.

IrDA® STANDARD

Overview

The IrDA standard specification is a half-duplex communication protocol with Serial Infrared (SIR) transmission speeds similar to those supported by an RS-232 port (9600 bps, 19.2 kbps, 38.4 kbps, 57.6 kbps and 115.2 kbps). Microchip currently supports only the SIR transmission speeds.

The half-duplex nature of the communications is due to the fact that the receiver is blinded by the light of its own transmitter. The infrared transceiver transmits pulses in a cone with a half-angle between 15 and 30 degrees (Figure 1). The pulses must be visible from a distance of one meter, but must not be so bright that the receiver is overwhelmed at close distances. In practice, optimal positioning for a receiver is usually a distance of 5 cm to 60 cm from the transmitter, in the center of the transmission cone.

Protocols

The initial specifications developed by the Infrared Data Association provided a mechanism for converting existing serial interfaces to infrared interfaces. These protocols closely mimic standard serial interfaces. As the infrared communication mechanism gained popularity, more protocols were created to tailor the communication format for different types of end applications.

The infrared communication support is designed as a Stack. Figure 2 shows the basic structure of the Stack.

FIGURE 1: OPTICAL PORT ANGLES

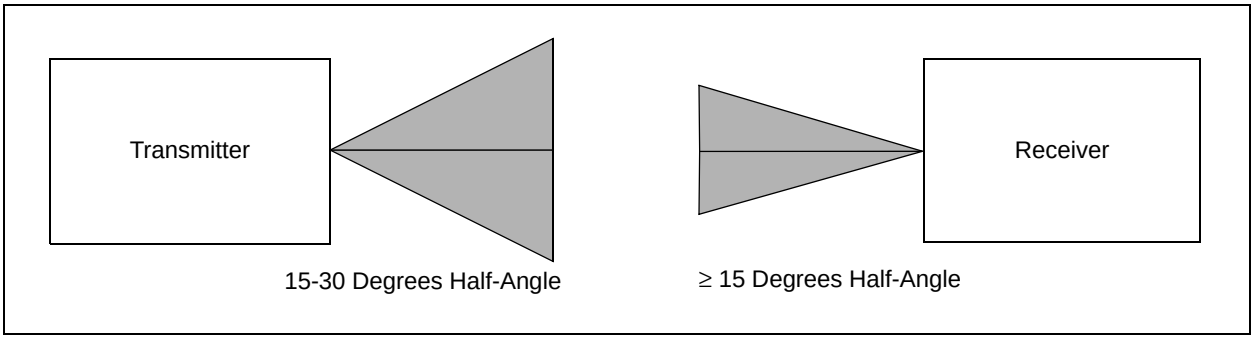
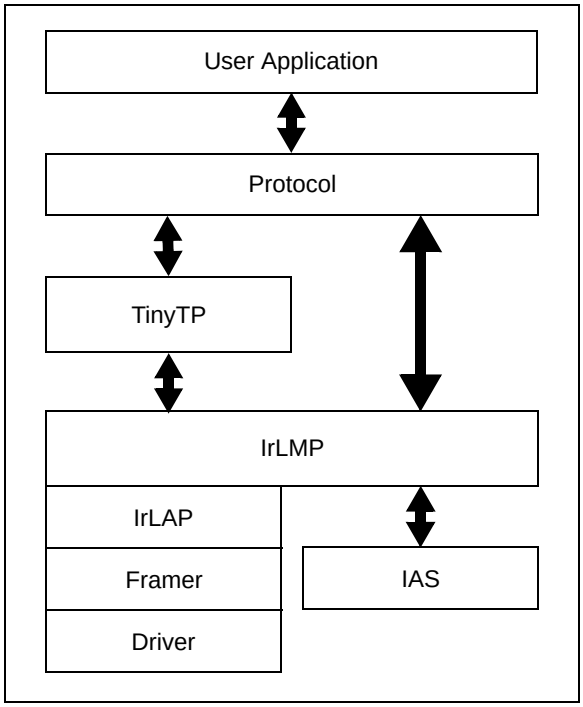


FIGURE 2: IrDA® STANDARD PROTOCOL STACK LAYERS



The Stack layers perform the following functions:

- Driver – Provides an interface between the Stack and the microcontroller.
- Framer – Prepares the IrLAP frame for transmission over the physical serial medium by wrapping it within a frame wrapper and encoding control characters in the data payload (with byte and bit stuffing) to make them transparent to the frame receiver. The framer receiver converts the encoded, transparent bytes back to their original values before validating and storing the frame in the receive queue.
- IrLAP (Infrared Link Access Protocol) – Provides a device-to-device connection for the reliable, ordered transfer of data. Also provides device discovery procedures.
- IrLMP (Link Management Protocol) – Provides fundamental discovery, multiplexing and link control operations between stations. It supports multiplexing of multiple applications over a single IrLAP link along with protocol and service discovery through the IAS.
- IAS (Information Access Service) – A mini database of the services provided by the device.
- TinyTP (Tiny Transport Protocol) – Provides flow control on IrLMP connections with an optional segmentation and reassembly service.

The current implementation of the Microchip IrDA Standard Stack allows access to the Stack through one of three different protocols:

- **IrCOMM 3-Wire Raw**

  This protocol is designed to emulate a simple serial interface consisting of two wires: a receive and a transmit line. (The third wire, ground, is not emulated). This protocol is also known as IrLPT, designed to emulate a PC parallel port interface.

- **IrCOMM 9-Wire Cooked**

  This protocol is designed to emulate a serial interface with either hardware or software handshaking.

- **OBEX**

  A higher level protocol, designed to simplify sending and receiving data objects.

These protocols and the application interfaces to them are described below.

Device Types

There are two basic types of devices:

- **Client** (or Primary)
  This device initiates the connection.
- **Server** (or Secondary)
  This device responds only when connected to.

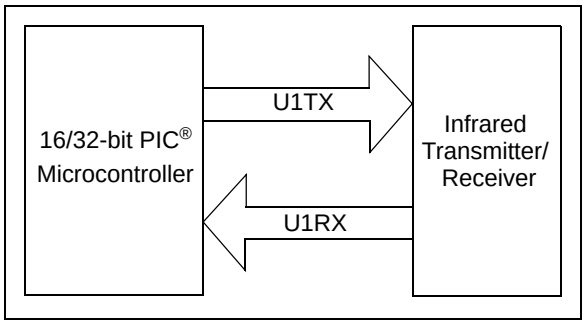
A third type of device, called a **Peer** device, can act as a Client or a Server. An example of a Peer device is a PDA, which can either beam information to another PDA or receive information from another PDA. Typically, IrCOMM applications are Clients or Servers.

HARDWARE DESIGN

Many members of Microchip's families of 16-bit and 32-bit microcontrollers provide native IrDA standard support through their UART modules. This greatly simplifies the hardware design (Figure 3).

For demonstration and prototyping purposes, Microchip has created the IrDA® PICtail™ Plus card (AC164124) for use with the Explorer 16 Development Board (DM240001).

FIGURE 3: BLOCK DIAGRAM



SOFTWARE DESIGN

Overview

The Microchip IrDA Standard Stack is distributed as a set of libraries, with source code provided for the lowest level drivers (see **Appendix A: "Source Code"**). This allows the Stack to be tailored to account for:

- Device family
- Device clock speed
- Protocol
- Device type

Due to the nature of the libraries, some operational parameters are fixed. These include the following parameters shown in Table 1.

TABLE 1: FIXED OPERATIONAL PARAMETERS

Item	Value	Effect
Internal Timer	`Timer 2`	Timer2 is unavailable to the application and Timer3 may be used only as a 16-bit timer.
Interrupts vs. Polling	`Interrupts`	The UART receive and transmit interrupts are used. Since these interrupts are vectored, this method provides the quickest, most reliable method of interfacing with the peripheral.
Window Size	`1`	Maximum number of information frames that can be transmitted before an Acknowledge is received. This parameter is set for minimum RAM usage.
Data Frame Size	`64`	Maximum LAP frame size. This parameter is set for minimum RAM usage.

Generic Stack API

The following function is supported for all Stack protocols and configurations.

DWORD IrDA_GetVersion(void)

This function returns the version of the Stack in a four-byte value. The Most Significant Byte contains the major release number, followed by the minor release number, dot release and build number. For example, “v1.4.10.16” would be represented as the value “0x01040A10”.

Syntax

```
DWORD IrDA_GetVersion( void );
```

Inputs

None

Outputs

Stack version number in the form:

```
<major><minor><dot><build>.
```

IrCOMM 3-Wire Raw

This protocol was designed to allow simple conversion of existing serial interfaces. No emulated flow control is provided, just data paths for receiving and transmitting data.

This protocol is nearly identical to IrLPT with the exception of the connection process. The API allows the application to specify if it wants to connect using the IrLPT or the IrCOMM 3-wire raw protocol.

Basic client functionality should be implemented as shown in Example 1.

EXAMPLE 1: IrCOMM 3-WIRE RAW BASIC CLIENT FUNCTIONALITY

```
Initialize the stack
Establish communications with a server
While running
    Perform background stack processing
    Exchange data with the server
Endwhile
Close the communications link with the server
Terminate stack operation
```

Basic server functionality should be implemented as shown in Example 2.

EXAMPLE 2: IrCOMM 3-WIRE RAW BASIC SERVER FUNCTIONALITY

```
Initialize the stack
While running
    While client is connected
        Perform background stack processing
        Exchange data with the client
    Endwhile
Endwhile
Terminate stack operation
```

IrCOMM 3-Wire Raw API

The following function calls are provided for this protocol. Refer to **"Demo Applications"** section for typical usage examples.

`IrDA_CloseCommClient`

This function causes the client application to disconnect from the IrDA COMM server. This function automatically performs any necessary Stack operations while waiting for the time-out period.

Syntax

```
BYTE IrDA_CloseCommClient( WORD timeout );
```

Application Type

Client

Inputs

`timeout` – The number of milliseconds to wait for the Stack to complete any processing that is in progress

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_TIMEOUT` – Time-out

IrDA_CloseCommServer

This function causes the server application to disconnect from the IrDA COMM client. This function automatically performs any necessary Stack operations while waiting for the time-out period.

Syntax

```
BYTE IrDA_CloseCommServer( WORD timeout );
```

Application Type

Server

Inputs

`timeout` – The number of milliseconds to wait for the Stack to complete any processing that is in progress

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_TIMEOUT` – Time-out

IrDA_CommBackground

This function processes Stack events as long as the device is connected. It also monitors any time-outs that need to be checked. The return code indicates if the device is no longer connected.

Syntax

```
BYTE IrDA_CommBackground( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Device is still connected

`IRDA_ERROR` – Device is no longer connected

IrDA_CommInit

This function initializes the Stack and the device peripherals. It must be called before any other Stack functions. Once called, it does not need to be called again until `IrDA_CommTerminate()` has been called.

Syntax

```
BYTE IrDA_CommInit( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR` – Failure

IrDA_CommTerminate

This function terminates the Stack. It also turns off all microcontroller peripherals used by the Stack (timer and UART).

This function should not be called until `IrDA_CommBackground()` indicates that all Stack tasks are complete. After calling this function, no other Stack functions can be called until `IrDA_CommInit()` is called.

Syntax

```
BYTE IrDA_CommTerminate( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

IrDA_OpenCommClient

This function tries to establish a client connection with another device. This is the point where the application requests either an IrLPT or IrCOMM 3-wire raw connection. The only difference between the two is the class name used during the discovery process.

Syntax

```
BYTE IrDA_OpenCommClient( BYTE type );
```

Application Type

Client

Inputs

`type` – `COMM_LPT` or `COMM_THREE_WIRE_RAW`

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_NO_BUFFERS` – No buffers available, out of memory

`IRDA_ERROR_BAD_COMM_STATE` – Bad communication state, connection failed

IrDA_OpenCommServer

This function tries to establish a server connection with another device.

Syntax

```
BYTE IrDA_OpenCommServer( WORD timeout );
```

Application Type

Server

Inputs

`timeout` – The number of milliseconds to try to establish a connection

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_LINK_CONNECT` – Link connect failed

`IRDA_ERROR_APP_CONNECT` – Application connection failed

IrDA_ReadComm

This function reads data from the IrDA standard port and stores it at the indicated location. If the amount of data exceeds the maximum size, the remaining data is discarded. This function will terminate when either the maximum number of characters has been received or when the time-out expires.

Since each IrCOMM data packet may contain multiple data bytes, a single read operation can return multiple bytes of data. A read request with a time-out of 0 ms will return the data in a single received data packet.

Note: Since the data frame size is set to 64, the received data size will never exceed 64 bytes.

Syntax

```
BYTE IrDA_ReadComm( BYTE *dataArray, WORD maxSize, WORD timeout, WORD *dataLength );
```

Application Type

Client or Server

Inputs

`*dataArray` – Pointer to where to store the data

`maxSize` – The maximum number of bytes to store at `*dataArray`

`timeout` – Number of milliseconds to wait for the data

Outputs

`*dataLength` – The actual amount of data stored at `*dataArray`

Return values:

`IRDA_SUCCESS (0x00)` – Success, some data read

`IRDA_ERROR` – Not connected

`IRDA_ERROR_TIMEOUT` – Time-out, no data read

IrDA_ReadInitComm

This function is used if the application wants to perform other processing while waiting for data. This function initiates a read from the IrDA standard port. The actual read is performed in the background. While the read is in progress, `IrDA_CommServerBackground()` must be called to process the Stack events, and `IrDA_ReadResultComm()` should be called to monitor the status of the read operation. `IrDA_ReadResetComm()` should be called after the application Acknowledges that the read is complete.

Since each IrCOMM data packet may contain multiple data bytes, a single read operation can return multiple bytes of data. A read request with a time-out of 0 ms will return the data in a single received data packet.

Note: Since the data frame size is set to 64, the received data size will never exceed 64 bytes.

Syntax

```
BYTE IrDA_ReadInitComm( BYTE *dataArray, WORD maxSize, WORD timeout );
```

Application Type

Client or Server

Inputs

`*dataArray` – Pointer to where to store the data

`maxSize` – The maximum number of characters that can be stored

`timeout` – The number of milliseconds for the read to terminate

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR` – Not connected

IrDA_ReadResetComm

This function is used if the application wants to do other processing while waiting for data. This function resets the variables used to monitor a read operation. It should be called after IrDA_ReadResultComm() indicates the read operation is complete.

Syntax

```
BYTE IrDA_ReadResetComm( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00)

IrDA_ReadResultComm

This function is used if the application wants to do other processing while waiting for data. This function is called to check on the status of a read that was initiated by calling IrDA_ReadInitComm(). If the return code indicates that a read is not currently in progress, then the application can call IrDA_ReadComm() or IrDA_ReadInitComm() to perform a read. If the return code indicates that the read is not complete, then the application should continue to call IrDA_CommBackground() until the read is complete. If the return code indicates that the read is complete, then *dataLength will indicate the number of bytes that were read, and the application should call IrDA_ReadResetComm() to reset the read operation parameters.

Syntax

```
BYTE IrDA_ReadResultComm( WORD *dataLength );
```

Application Type

Client or Server

Inputs

None

Outputs

*dataLength – The actual amount of data stored at the location specified by the user in the call to IrDA_ReadInitComm()

Return values:

IRDA_COMM_READ_COMPLETE

IRDA_COMM_READ_NOT_IN_PROGRESS

IRDA_COMM_READ_NOT_COMPLETE

IrDA_StackIsActive

This function indicates whether or not the Stack is still processing frames.

Syntax

```
BYTE IrDA_StackIsActive( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

False – Stack is not active, all frames have been processed

True – Stack is active, frames are still being processed

IrDA_WriteComm

This function sends data out the IrDA standard port. The data is actually sent during background processing. This function does not lock the system while the write is in progress.

Note:	Since the data frame size is set to 64, each transfer is limited to a total of about 60 bytes. If the output buffer size exceeds that limit, an error will be returned.

Syntax

```
BYTE IrDA_WriteComm( BYTE *prt_buf, WORD buf_size );
```

Application Type

Client or Server

Inputs

`*prt_buf` – Pointer to the user data

`buf_size` – The number of characters to send

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_NO_BUFFERS` – No buffers, out of memory

`IRDA_ERROR_WRITE_MASK` – Bad communication state or `LM_Data_request` error if this bit is set

IrCOMM 9-Wire Cooked

This protocol is similar to the IrCOMM 3-wire raw protocol, except that hardware and software handshaking interfaces have been provided to mimic those used by a wired serial interface. Since there are no separate wires to carry these interface signals, the serial data stream is divided into two virtual channels, a control channel and a data channel. This slightly increases the complexity of this protocol.

Many devices that advertise or require the IrCOMM 9-wire cooked service do not actually utilize the control channel, since items like data rate and handshaking already are provided by the IrDA Standard Stack. Therefore, to reduce overhead, the Microchip IrDA Standard Stack provides a minimal interface to the emulated control signals.

Any required control channel handling must be performed by the application.

The Stack maintains the control parameter values that have been received from the remote device. Macros have been provided to simplify access to these values, as described in **Appendix C: "IrCOMM 9-Wire Cooked Control Channel Access Macros"**. If desired, the application may also utilize the control channel data structures to maintain its own control parameter values. These data structures are described in **Appendix B: "IrCOMM 9-Wire Cooked Data Structures"**.

Basic client and server functionality is identical to that of the IrCOMM 3-wire raw protocol. Data transfer is slightly more complicated, due to the control channel. When writing to the IrDA standard port, the control channel must be initialized. When reading from the IrDA standard port, the received control channel values are available for the user to check, as described in **Appendix C: "IrCOMM 9-Wire Cooked Control Channel Access Macros"**.

Note:	All raw data received is stored in the user buffer. XON/XOFF and ENQ/ACK characters are not filtered out, and must be processed by the application to emulate the required handshaking.

Data transmission is performed as shown in Example 3.

EXAMPLE 3: IrCOMM 9-WIRE COOKED DATA TRANSMISSION ALGORITHM

```
Initialize the data packet
If sending control parameters
    Initialize the control parameter list
    For each control parameter
        Add the control parameter
    Endfor
    Finish the control parameter list
Else
    Set the control parameter list to no
            parameters
Endif
Send the data
```

IrCOMM 9-Wire Cooked API

The following function calls are provided for this protocol. Refer to **"Demo Applications"** section for examples of typical usage.

IrDA_AddControlParam

Use this function to add a control parameter to a data packet being prepared for transmission.

Syntax

```
BYTE IrDA_AddControlParam( BYTE pi, DWORD pv );
```

Application Type

Client or Server

Inputs

`pi` – Parameter identifier

`pv` – Parameter value

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_PACKET_SIZE` – Max packet size exceeded

`IRDA_ERROR_UNKNOWN_PI` – Unknown parameter identifier

Valid values for `pi` are provided as constants in the file, `irdep.h`, as described in Table 2.

TABLE 2: IrCOMM 9-WIRE COOKED CONTROL PARAMETER IDENTIFIERS

Parameter	Identifier Constant	Value Size in Bytes	Value Structure
Service Type	SERVICE_TYPE	1	IRDA_SERVICE_TYPE
Data Rate (bps)	DATA RATE	4	DWORD
Data Format	DATA FORMAT	1	IRDA DATA FORMAT
Flow Control	FLOW CONTROL	1	IRDA FLOW CONTROL
XON/XOFF Characters	XON_XOFF	2	XON (in lower byte), XOFF (in upper byte)
ENQ/ACK Characters	ENQ_ACK	2	ENQ (in lower byte), ACK (in upper byte)
Line Status	LINE STATUS	1	IRDA_LINE_STATUS
Break	BREAK	1	None (0 = clear, 1 = set)
DTE Line Settings and Changes	DTE_LINE	1	IRDA_DTE_LINE_STATUS
DCE Line Settings and Changes	DCE_LINE	1	IRDA_DCE_LINE_STATUS
Poll for Line Settings	POLL_FOR_LINE	None	None

IrDA_CloseCommXClient

This function disconnects the IrDA Standard Stack IrCOMM 9-wire cooked client. This function automatically performs any necessary Stack operations while waiting for the time-out period.

Syntax

```
BYTE IrDA_CloseCommXClient( WORD timeout );
```

Application Type

Client

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_TIMEOUT – Time-out

IrDA_CloseCommXServer

This function disconnects the IrDA IrCOMM 9-wire cooked server. This function automatically performs any necessary Stack operations while waiting for the time-out period.

Syntax

```
BYTE IrDA_CloseCommXServer( WORD timeout );
```

Application Type

Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_TIMEOUT – Time-out

IrDA_CommXBackground

This function processes Stack events as long as the device is connected. It also monitors any time-outs that must be checked. The return code indicates whether the device is no longer connected.

Syntax

```
BYTE IrDA_CommXBackground( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Device is still connected

IRDA_ERROR – Device is no longer connected

IrDA_CommXInit

This function initializes the Stack and the device peripherals. It must be called before any other Stack functions. Once called, it does not need to be called again until IrDA_CommXTerminate() has been called.

Syntax

```
BYTE IrDA_CommXInit( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR – Failure

IrDA_CommXTerminate

This function terminates the Stack, turning off the clock and the UART.

This function should not be called until IrDA_CommXBackground() indicates that all Stack tasks are complete. After this function is called, IrDA_CommXInit() must be called to restart the Stack.

Syntax

```
BYTE IrDA_CommXTerminate( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IrDA_FinishControlParamList

Use this function to finalize a control parameter list. Call IrDA_StartControlParamList() to initialize the parameter list, IrDA_AddControlParam() to add each parameter, then call IrDA_FinishControlParamList() to finalize the list. If there are no control parameters, use IrDA_NoControlParameters() instead.

Syntax

```
BYTE IrDA_FinishControlParamList( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_PACKET_SIZE – Maximum packet size exceeded

IrDA_InitCommXDataPacket

Use this function to initialize a data packet for transmission.

Syntax

```
BYTE IrDA_InitCommXDataPacket( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_NO_BUFFERS – No buffers available

IrDA_NoControlParameters

Use this function to indicate that there are no control parameters in the data packet.

Syntax

```
BYTE IrDA_NoControlParameters( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_PACKET_SIZE – Maximum packet size exceeded

IrDA_OpenCommXClient

This function tries to establish a client connection with another device. Before calling this function, IrDA_CommXInit() must be called and returned with success.

Syntax

```
BYTE IrDA_OpenCommXClient( void );
```

Application Type

Client

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_NO_BUFFERS – No buffers available, out of memory

IRDA_ERROR_BAD_COMM_STATE – Bad communication state, connection failed

IRDA_ERROR_COMM_CONNECT – Communication service connect time-out

IRDA_ERROR_BAD_COMM_SERVICE – Communication service disconnected and is unsupported

IRDA_ERROR_SELECTOR_MASK – If these bits are set, the remainder indicates a get remote selector error

IRDA_ERROR_TTP_MASK – If these bits are set, the remainder indicates a TTP connect request error

IrDA_OpenCommXServer

This function tries to establish a server connection with another device. Before calling this function, IrDA_CommXInit() must be called and returned with success.

Syntax

```
BYTE IrDA_OpenCommXServer( WORD timeout );
```

Application Type

Server

Inputs

timeout – Number of milliseconds to wait for a connection

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_LINK_CONNECT – Link connect failed

IRDA_ERROR_APP_CONNECT – Application connection failed

IrDA_ReadCommX

This function reads data from the IrDA standard port, and stores it at the indicated location. If the amount of data exceeds the maximum size, the remaining data is discarded. This function will terminate when either the maximum number of characters has been received or the time-out expires.

Since each IrCOMM data packet may contain multiple data bytes, a single read operation can return multiple bytes of data. A read request with a time-out of 0 ms will return the data in a single received data packet.

Note: Since the data frame size is set to 64, the received data size will never exceed 64 bytes.

Syntax

```
BYTE IrDA_ReadCommX( BYTE *dataArray, WORD maxSize, WORD timeout, WORD *dataLength );
```

Application Type

Client or Server

Inputs

`*dataArray` – Pointer to the user's buffer

`maxSize` – Maximum number of characters that can be stored

`timeout` – Number of milliseconds to wait for the data

`*dataLength` – Pointer to return the number of bytes received

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success, some data read

`IRDA_ERROR` – Not connected

`IRDA_ERROR_TIMEOUT` – Time-out, no data read

IrDA_ReadInitCommX

This function is used if the application wants to do other processing while waiting for data. This function initiates a read from the IrDA standard port. The actual read is performed in the background. While the read is in progress, `IrDA_CommXServerBackground()` must be called to process the Stack events, and `IrDA_ReadResultCommX()` should be called to monitor the status of the read operation. `IrDA_ReadResetCommX()` should be called after the application Acknowledges that the read is complete.

Since each IrCOMM data packet may contain multiple data bytes, a single read operation can return multiple bytes of data. A read request with a time-out of 0 ms will return the data in a single received data packet.

Note: Since the data frame size is set to 64, the received data size will never exceed 64 bytes.

Syntax

```
BYTE IrDA_ReadInitCommX( BYTE *dataArray, WORD maxSize, WORD timeout );
```

Application Type

Client or Server

Inputs

`*dataArray` – Pointer to where to store the data

`maxSize` – Maximum number of characters that can be stored

`timeout` – Number of milliseconds for the read to terminate

Outputs

Return values

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR` – Not connected

IrDA_ReadResetCommX

This function is used if the application wants to do other processing while waiting for data. This function resets the variables used to monitor a read operation. It should be called after IrDA_ReadResultCommX() indicates the read operation is complete.

Syntax

```
BYTE IrDA_ReadResetCommX( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IrDA_ReadResultCommX

This function is used if the application wants to do other processing while waiting for data. This function is called to check on the status of a read that was initiated by calling IrDA_ReadInitCommX(). If the return code indicates that a read is not currently in progress, then the application can call IrDA_ReadCommX() or IrDA_ReadInitCommX() to perform a read. If the return code indicates that the read is not complete, then the application should continue to call IrDA_CommXBackground() until the read is complete. If the return code indicates that the read is complete, then *dataLength will indicate the number of bytes that were read, and the application should call IrDA_ReadResetCommX() to reset the read operation parameters.

Syntax

```
BYTE IrDA_ReadResultCommX( WORD *dataLength );
```

Application Type

Client or Server

Inputs

None

Outputs

*dataLength – The actual amount of data stored at the location specified by the user in the call to IrDA_ReadInitCommX()

Return values:

IRDA_COMM_READ_COMPLETE

IRDA_COMM_READ_NOT_IN_PROGRESS

IRDA_COMM_READ_NOT_COMPLETE

IrDA_StackIsActive

This function indicates whether or not the Stack is still processing frames.

Syntax

```
BYTE IrDA_StackIsActive( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

False – Stack is not active, all frames have been processed

True – Stack is active, frames are still being processed

IrDA_StartControlParamList

Use this function to initialize a control parameter list. Call `IrDA_StartControlParamList()` to initialize the parameter list, call `IrDA_AddControlParam()` to add each parameter, then call `IrDA_FinishControlParamList()` when the parameter list is complete. If there are no control parameters, use `IrDA_NoControlParameters()` instead.

Syntax

```
BYTE IrDA_StartControlParamList( void );
```

Application Type

Client or Server

Inputs

None

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_PACKET_SIZE` – Maximum packet size exceeded

IrDA_WriteCommX

Use this function to transmit a data packet. The data to transmit is passed into this function. The control channel must be set up prior to calling this function. The data is actually sent during background processing. This function does not lock the system while the write is in progress.

Note:	Since the data frame size is set to 64, each transfer is limited to a total of about 60 bytes, including control parameters. If the output buffer size exceeds that limit, an error will be returned.

Syntax

```
BYTE IrDA_WriteCommX( BYTE *prt_buf, WORD buf_size );
```

Application Type

Client or Server

Inputs

`*prt_buf` – Pointer to user data

`buf_size` – Number of characters of user data

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR_PACKET_SIZE` – Data size too large, check control parameters

`IRDA_ERROR_APP_CONNECT` – Bad communication state

`IRDA_ERROR_TTP_DATA_MASK` – If these bits are set, the remainder indicates a TTP data request error

OBEX

Since the OBEX protocol is used to exchange complete objects, OBEX has the simplest user interface. A single client function is used to establish a connection, send data and terminate the connection. Server functionality is only slightly more complicated. Stack initialization and termination functions are provided to enable and disable the required peripherals.

OBEX API

The following function calls are provided for this protocol. Refer to **"Demo Applications"** section for examples of typical usage.

IrDA_InitServerOBEX

This function initializes the OBEX server.

Syntax

```
WORD IrDA_InitServerOBEX( void );
```

Application Type

Server or Peer

Inputs

None

Outputs

Return values:

`IRDA_SUCCESS (0x00)` – Success

`IRDA_ERROR` – LAP link failed

IrDA_ReceiveOBEX

This function receives an OBEX file from another device. The file can be stored either in RAM or in a user-defined memory area. If the file is to be stored in RAM, set the *fptrUserStore parameter to NULL. If the file requires application-specific code to store the bytes, such as writing to external memory, create a callback function with the following prototype:

```
void myDataStore( UINT32 index, UINT32 maxLength, UBYTE ch );
```

This function should take the byte, ch, and store it to the location index. The function should check that index has not exceeded maxLength before storing the data byte. When calling IrDA_ReceiveOBEX(), set the *fptrUserStore parameter to the callback function, and set *dataArray to NULL.

Syntax

```
BYTE IrDA_ReceiveOBEX( BYTE *fileDescription, BYTE *fileName, void *fptrUserStore,
       BYTE *dataArray, DWORD maxLength, DWORD *dataLength, WORD timeout );
```

Application Type

Server or Peer

Inputs

*fileDescription – Pointer to a text description of the file

*fileName – Name of the file to transfer

*fptrStore – Pointer to a user function to store received data; must be NULL if *dataArray is not NULL

*dataArray – Pointer to the characters to send; must be NULL if *fptrStore is not NULL

timeout – Operation time-out in milliseconds

maxLength – Maximum number of characters that can be stored

Outputs

*dataLength – Number of bytes received

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_USER_EXIT – Terminated by the user

IRDA_ERROR_LINK_TIMEOUT – Link connect time-out

IRDA_ERROR_TIMEOUT – OBEX connect time-out

IrDA_SendOBEX

This function sends an OBEX file to another device. This function contains the entire OBEX transfer, including initializing the Stack, establishing a connection to the other device, sending the data and terminating the connection and the Stack. The file can be located either in RAM or in a user-defined memory area. If the file is in RAM, set the *fptrUserRead parameter to NULL. If the file requires application-specific code to extract the bytes, such as reading from external memory, create a callback function with the following prototype:

```
void myDataRead( BYTE *destination, DWORD startIndex, WORD size );
```

This function should take size bytes starting with the byte at index startIndex, and copy them to the RAM location specified by *destination. When calling IrDA_SendOBEX, set the *fptrUserRead parameter to the callback function, and set *dataArray to NULL.

Syntax

```
WORD IrDA_SendOBEX( BYTE *fileDescription, BYTE *fileName, void *fptrUserRead, BYTE
      *dataArray, DWORD dataLength );
```

Application Type

Server or Peer

Inputs

*fileDescription – Pointer to a text description of the file

*fileName – Name of the file to transfer

*fptrUserRead – Pointer to a user function to obtain bytes of the file; should be NULL if *dataArray is not NULL

*dataArray – Pointer to the characters to send from RAM; should be NULL if *fptrUserRead is not NULL

dataLength – Number of characters in *dataArray to send

Outputs

Return values:

IRDA_SUCCESS (0x00) – Success

IRDA_ERROR_OBEX_MAKE – OBEX make failed

IRDA_ERROR_OBEX_SAR_TX – OBEX SAR TX failed

IRDA_ERROR_OBEX_SERVER_TO – Wait for server response time-out

IRDA_ERROR_OBEX_SERVER_RSP – Unknown server response

IRDA_ERROR_LAP_LINK – LAP link initialization failed

IRDA_ERROR_REMOTE_SEL – Get remote selector failed

IRDA_ERROR_NO_BUFFERS – No buffers

IRDA_ERROR_OBEX_CONNECT – OBEX connect failed

IRDA_ERROR_OBEX_TIMEOUT – Connection time-out

IRDA_ERROR_NO_BUF_DISCONN – No buffers available at disconnect

IRDA_ERROR_OBEX_SERVER – OBEX server connection failed

IrDA_TerminateOBEX

This function terminates the IrDA Standard Stack functioning.

Syntax

```
void IrDA_TerminateOBEX( void );
```

Application Type

Server or Peer

Inputs

None

Outputs

None

STACK INSTALLATION

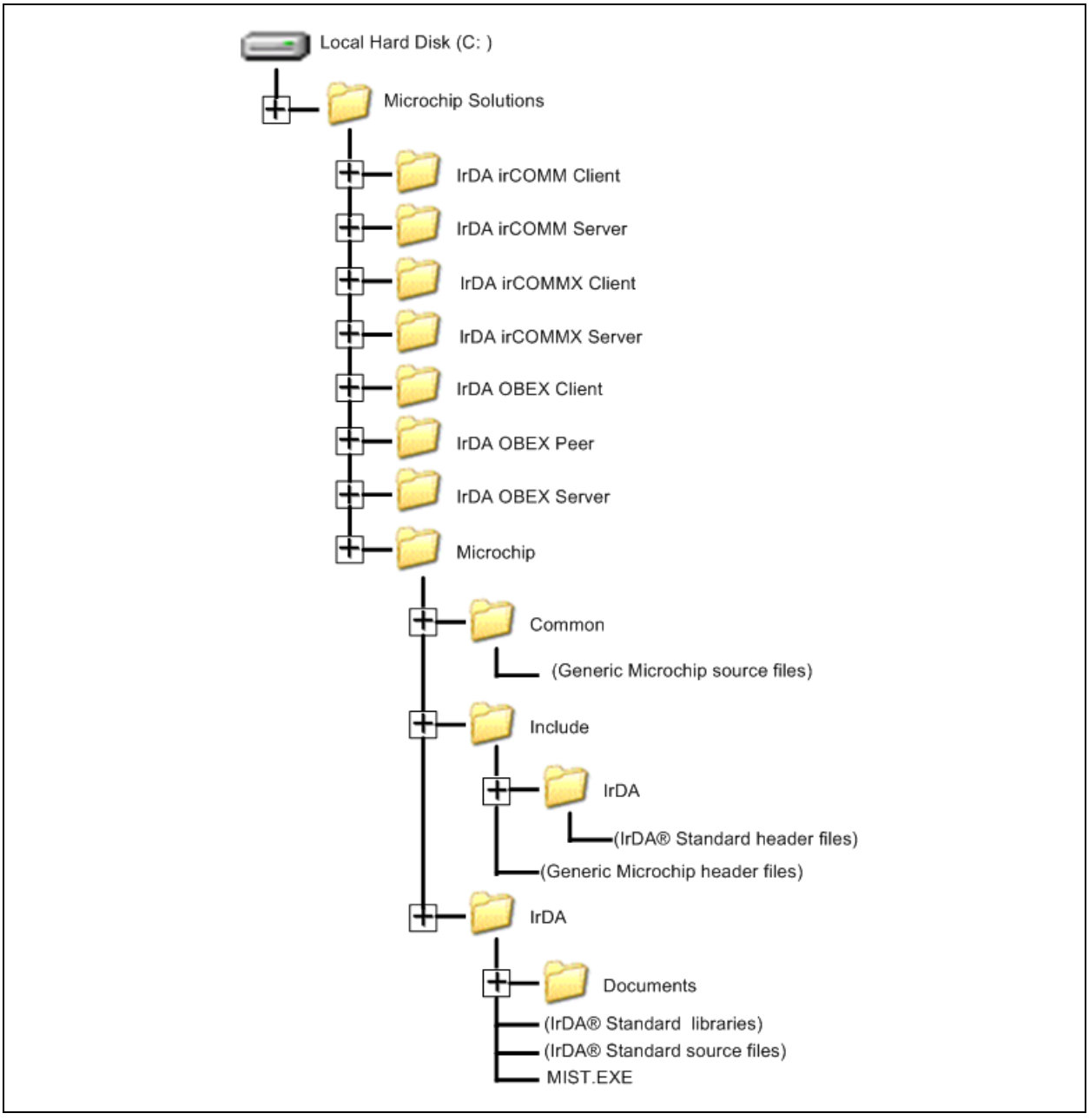
The Microchip IrDA Standard Stack libraries are available for download from the Microchip web site (see **Appendix A: "Source Code"**). Download and execute the installation file. Before the software is installed, you must accept the software license agreement.

By default, the libraries will be installed in the directory structure shown in Figure 4.

The name of the top-level folder, Microchip Solutions, may be changed during the installation process. The Microchip subfolder contains Microchip created libraries, source code, documentation and other support files. The other subfolders contain various demo projects.

When you create your own application, create a new subfolder for it at this level.

FIGURE 4: INSTALLATION DIRECTORY STRUCTURE



DEMO APPLICATIONS

16-bit Microcontrollers

The following items are required in order to fully utilize the demonstration projects for 16-bit microcontrollers:

- Explorer 16 Development Board with a PIC24FJ128GA010 PIM (two are recommended for the IrCOMM 3-wire raw and IrCOMM 9-wire cooked demonstrations)
- IrDA® PICtail™ Plus (two are recommended for the IrCOMM 3-wire raw and IrCOMM 9-wire cooked demonstrations)
- MPLAB® IDE, version 7.42 or newer
- MPLAB ICD 2 or MPLAB REAL ICE (device programmer)
- MPLAB C30 C Compiler, version 2.04 or newer

Each project can be built and programmed into the Explorer 16 by following these general steps:

1. Start MPLAB IDE.
2. Select *Project > Open…*.
3. Locate the `.mcp` project file in the desired demonstration directory. Select it and then click **Open**.
4. Select *Project > Build All* to build the project.
5. Select *Programmer > Select Programmer*. If the desired device programmer is not checked, select it.
6. Connect the device programmer to the PC using the USB cable.
7. Install the PIC24F PIM and the IrDA PICtail Plus into the Explorer 16.
8. Connect the device programmer to the Explorer 16. Then connect the power supply to the Explorer 16.

Note:	Do not attempt to use the device programmer to power the Explorer 16 Development Board.

9. If using MPLAB ICD 2 as the device programmer, select *Programmer > Connect* to connect to the MPLAB ICD 2.
10. Select *Programmer > Program* to program the Explorer 16.

32-bit Microcontrollers

The following items are required in order to fully utilize the demonstration projects for 32-bit microcontrollers:

- Explorer 16 Development Board with a PIC32MX460F512L PIM (two are recommended for the IrCOMM 3-wire raw and IrCOMM 9-wire cooked demonstrations) or PIC32MX795F512L PIM
- IrDA® PICtail™ Plus (two are recommended for the IrCOMM 3-wire raw and IrCOMM 9-wire cooked demonstrations)
- MPLAB® IDE, version 8.46 or newer
- MPLAB ICD 2 or MPLAB REAL ICE (device programmer)
- MPLAB C32 C Compiler, version 1.11 or newer

Each project can be built and programmed into the Explorer 16 by following these general steps:

1. Start MPLAB IDE.
2. Select *Project > Open…*.
3. Locate the `.mcp` project file in the desired demonstration directory. Select it and then click **Open**.
4. Select *Project > Build All* to build the project.
5. Select *Programmer > Select Programmer*. If the desired device programmer is not checked, select it.
6. Connect the device programmer to the PC using the USB cable.
7. Install the PIC32 PIM and the IrDA PICtail Plus into the Explorer 16.
8. Connect the device programmer to the Explorer 16. Then connect the power supply to the Explorer 16.

Note:	Do not attempt to use the device programmer to power the Explorer 16 Development Board.

9. If using MPLAB ICD 2 as the device programmer, select *Programmer > Connect* to connect to the MPLAB ICD 2.
10. Select *Programmer > Program* to program the Explorer 16.

The demonstration programs are designed to output information over the RS-232 connection so it can be displayed on a terminal program. Using a serial cable, connect the Explorer 16 Development Board's DB9 connector to a PC, and start a terminal emulation program, such as Microsoft® HyperTerminal, to monitor the output. Communication settings for the connection are: 57600 baud, 8 data bits, no parity, 1 Stop bit and no flow control.

The demonstration program on the Explorer 16 can now be executed. If using the MPLAB® REAL ICE™ in-circuit emulator, execution will begin as soon as programming is complete. If using the MPLAB ICD 2, begin execution by either removing the MPLAB ICD 2 cable from the Explorer 16 or by selecting *Programmer > Release from Reset*.

When running the demonstration projects, be sure that the infrared transceivers of the two communicating devices are properly aligned.

IrCOMM 3-Wire Raw

Two IrCOMM 3-wire raw demonstration projects are provided: a client demo and a server demo. The two projects are designed to work together utilizing two Explorer 16 Development Boards with IrDA PICtail Plus.

Follow the procedure described previously to set up one Explorer 16 Development Board using the project found in the `irCOMM Server Demo` directory. Allow the server application to execute. A brief banner will be displayed on the terminal emulation program.

The server will now wait until a client tries to establish a connection with it. The server will periodically print dots to the terminal, indicating that it is still waiting for a connection.

Next, set up a second Explorer 16 Development Board using the project found in the `irCOMM Client Demo` directory. Align the two boards so their infrared transceivers are pointed toward each other, and allow the client application to execute.

The client will establish a connection with the server, send the server a character string and disconnect from the server (Example 4 and Example 5). The server will display the received string and continue monitoring the client for more data until the client disconnects. Then, the server will shut down.

EXAMPLE 4: IrCOMM 3-WIRE RAW SERVER TERMINAL OUTPUT

```
irCOMM 3-wire Raw Server Demo

Waiting for client...
Receiving...
This is a test string!
This is a test string!
This is a test string!
This is a test string!
This is a test string!
Disconnected
Demonstration complete!
```

EXAMPLE 5: IrCOMM 3-WIRE RAW CLIENT TERMINAL OUTPUT

```
irCOMM 3-wire Raw Client Demo

Sending the test string...
Sending the test string...
Sending the test string...
Sending the test string...
Sending the test string...
Demonstration complete!
```

Note that the server has two methods of reading data from the client. Switch between the two methods by either defining or not defining USE_SINGLE_STEP_READ at the top of the server source file. The affects of these methods are displayed on the terminal. When a single step read utilizing the `IrDA_ReadComm()` is used, the program is simpler, but execution is locked until either the read is complete or the read times out. A read that utilizes the `IrDA_ReadInitComm()`, `IrDA_ReadResultComm()`, `IrDA_ReadResetComm()` and `IrDA_CommBackground()` functions is more complicated to implement, but gives the user more control over how the read is performed. To simply read the contents of a single data transfer, either method with a time-out of '0' could be used.

IrCOMM 9-Wire Cooked

Two IrCOMM 9-wire cooked demonstration projects are provided: a client demo and a server demo. The two projects are designed to work together utilizing two Explorer 16 Development Boards with IrDA PICtail Plus.

Follow the procedure described on the previous page to set up one Explorer 16 Development Board using the project found in the `irCOMMX Server Demo` directory. Allow the server application to execute. A brief banner will be displayed on the terminal emulation program.

The server will now wait until a client tries to establish a connection with it. The server will periodically print dots to the terminal, indicating that it is still waiting for a connection.

Next, set up a second Explorer 16 Development Board using the project found in the `irCOMMX Client Demo` directory. Align the two boards so their infrared transceivers are pointed toward each other, and allow the client application to execute.

The client will establish a connection with the server, send the server a character string and disconnect from the server. The server will display the received string and continue monitoring the client for more data until the client disconnects. Then, the server will shut down (Example 6 and Example 7).

EXAMPLE 6: IrCOMM 9-WIRE COOKED SERVER TERMINAL OUTPUT

```
irCOMM 9-wire Cooked Server Demo

Waiting for client...
Receiving...
This is a test string!
This is a test string!
This is a test string!
This is a test string!
This is a test string!
Disconnected
Demonstration complete!
```

EXAMPLE 7: IrCOMM 9-WIRE COOKED CLIENT TERMINAL OUTPUT

```
irCOMM 9-wire Cooked Client Demo

Sending the test string...
Sending the test string...
Sending the test string...
Sending the test string...
Sending the test string...
Demonstration complete!
```

Note that the server has two methods of reading data from the client. Switch between the two methods by either defining or not defining `USE_SINGLE_STEP_READ` at the top of the server source file. The affects of these methods are displayed on the terminal. When a single step read, utilizing the `IrDA_ReadCommX()` is used, the program is simpler, but execution is locked until either the read is complete or the read times out. A read that utilizes the `IrDA_ReadInitCommX()`, `IrDA_ReadResultCommX()`, `IrDA_ReadResetCommX()` and `IrDA_CommXBackground()` functions is more complicated to implement, but gives the user more control over how the read is performed. To simply read the contents of a single data transfer, either method with a time-out of '`0`' could be used.

Also, note that the procedure for the client to send data is different than for the IrCOMM 3-wire raw example. Before the data is written, the control channel must be initialized. The general procedure to send a packet is shown in Example 8.

EXAMPLE 8: IrCOMM 9-WIRE COOKED DATA TRANSMISSION PROCEDURE

```
IrDA_InitCommXDataPacket()
if sending control parameters
    IrDA_StartControlParamList()
    for each control parameter
        IrDA_AddControlParam()
    endfor
    IrDA_FinishControlParamList()
else
    IrDA_NoControlParameters()
endif
IrDA_WriteCommX()
```

OBEX

Three OBEX demonstration projects are provided. The client demo and server demo projects are designed to either work together utilizing two Explorer 16 Development Boards with IrDA PICtail Plus or work with another OBEX device, such as a PDA or cell phone.

The following descriptions are for using a single Explorer 16 with a PDA. To perform the demonstration with two Explorer 16 boards, simply program one as the client and one as the server, align the two boards and allow them to execute.

To experiment with receiving OBEX data, set up the Explorer 16 Development Board using the project found in the `OBEX Server Demo` directory. Allow the server application to execute. A brief banner will be displayed on the terminal emulation program.

The server will now wait until a client tries to establish a connection with it. Align the PDA's infrared transceiver with the transceiver of the IrDA PICtail Plus, select a contact from the PDA's address book and beam it to the Explorer 16. The PDA will transmit the contact information as a vCard, and the Explorer 16 will send the raw data contained in the OBEX transfer to the terminal for display (Example 9 and Example 10).

To experiment with sending OBEX data, set up the Explorer 16 Development Board using the project found in the `OBEX Client Demo` directory. Align the PDA's infrared transceiver with the IrDA PICtail Plus's transceiver, and allow the client application to execute.

The client will establish a connection with the PDA, send a vCard to the PDA, and disconnect from the PDA. To view the vCard, allow the PDA to accept the transfer, and then view the entry in the PDA's address book (Example 11).

The third demonstration project is an OBEX peer, designed to mimic a PDA. While powered, the Explorer 16 will enter Server mode and periodically see if any devices are trying to establish a connection with it. If a device does establish a connection with it and sends it information, the received information will be displayed on the terminal emulation program. If the user pressed the RD6 button on the Explorer 16 Development Board, the application will switch to client mode, try to establish a connection with a server and send a vCard to the server. It will then return to Server mode.

EXAMPLE 9: OBEX SERVER TERMINAL OUTPUT FOR 16-BIT MICROCONTROLLERS

```
OBEX Server.
Receiving... ----
BEGIN:VCARD
N:Microcontroller;PIC24F
ADR;DOM;WORK:;;2355 W. Chandler Blvd.;Chandler;AZ;85224;USA
ORG:Microchip Technology, Inc.
TITLE:16-bit Microcontroller
TEL;PREF;WORK;VOICE:(480) 792-7200
URL;WORK:www.microchip.com/16bit
BDAY:20060418
```

EXAMPLE 10: OBEX SERVER TERMINAL OUTPUT FOR 32-BIT MICROCONTROLLERS

```
OBEX Server.
Receiving... ----
BEGIN:VCARD
N:Microcontroller;PIC32MX
ADR;DOM;WORK:;;2355 W. Chandler Blvd.;Chandler;AZ;85224;USA
ORG:Microchip Technology, Inc.
TITLE:32-bit Microcontroller
TEL;PREF;WORK;VOICE:(480) 792-7200
URL;WORK:www.microchip.com/32bit
BDAY:20060418
```

EXAMPLE 11: OBEX CLIENT TERMINAL OUTPUT

```
OBEX Client.
Sending vCard...
vCard sent.
```

MICROCHIP'S IrDA® STACK TOOL

The simplest way to start a new project using an IrDA standard protocol is to use Microchip's IrDA Standard Stack tool, which is installed with the libraries and source files associated with this application note (see **Appendix A: "Source Code"**). Refer to the README file installed with the Stack for information regarding any updates to the Stack tool.

1. In the installation subdirectory, locate and start the IrDA Standard Stack tool (`MIST.EXE`).

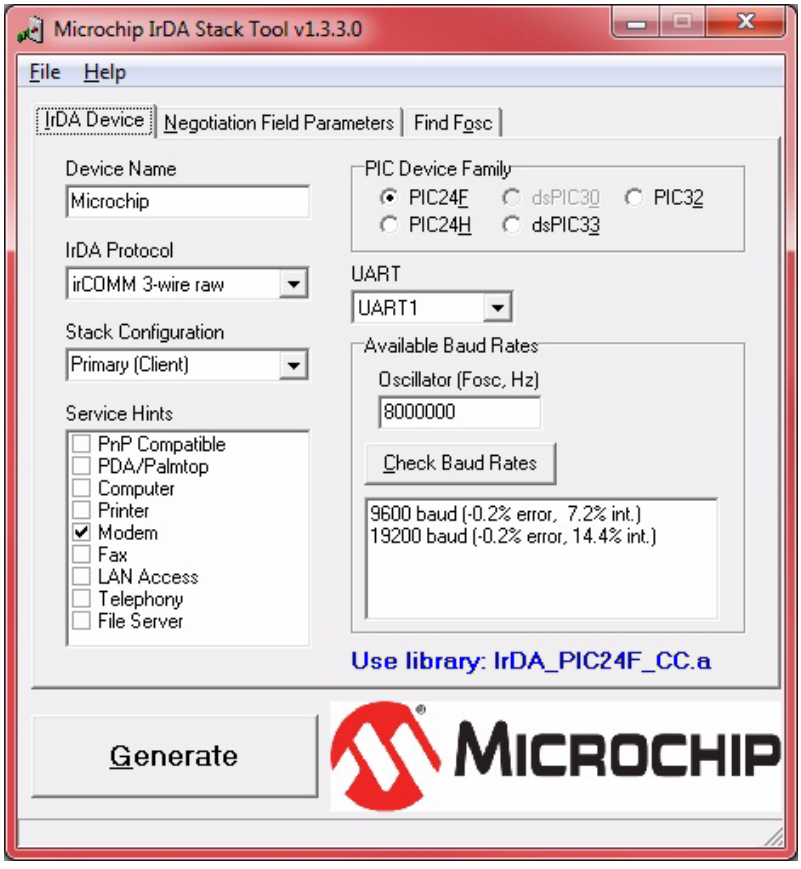
   This tool will create project files required to use the Microchip IrDA Standard Stack. It will also indicate which library file to include in your project, based on the selected protocol and device type.
2. Select the **IrDA Device** tab and complete the following fields:
   a) **Device Name**: The string that your device will report as its identifier during the discovery process. Maximum of 23 characters.
   b) **IrDA Protocol**: The IrDA standard protocol that your application will use.
   c) **Stack Configuration**: Your application's device type.
   d) **Service Hints**: Service hints for your application.
   e) **PIC Device Family**: Select the device family for your application's target PIC microcontroller device
   f) **PIC Device Header File**: Enter the header file for your application's target PIC microcontroller device.
   g) **UART**: Select the UART that will be used for IrDA communication. Confirm that the selected UART exists on the target device.
   h) **Oscillator Frequency (Hz)**: Enter your application's oscillator frequency in hertz, and then click **Select**.

   The baud rates that your application can support are displayed. Use these baud rates for reference when selecting supported baud rates on the **Negotiation Field Parameters** tab.

> **Note:** If PIC32 is selected as the PIC Device Family, **Peripheral Bus (Fpb)** is displayed as a selection within the Available Baud Rates, which can be used to select the peripheral bus speed divider.

3. Remove the PIC device header file.

FIGURE 5: SELECTING AND CONFIGURING THE IrDA® DEVICE



4. Select the **Negotiation Field Parameters** tab.

   Use this tab to enter the desired connection parameters to be used during the negotiation process. Common default settings are provided.

   a) **Supported Baud Rates**: Select the baud rates that your application will support.

   b) **Additional BOF's**: Select the number of additional flags needed at the beginning of each frame (BOF = Beginning of Frame).

   c) **Minimum Turnaround Time**: Select the minimum communication turnaround time.

   d) **Maximum Turnaround Time**: Select the maximum communication turnaround time.

   e) **Window Size**: Select the application window size. The maximum window size is fixed by the library.

   f) **Data Size**: Select the data frame size. The maximum data size is fixed by the library.

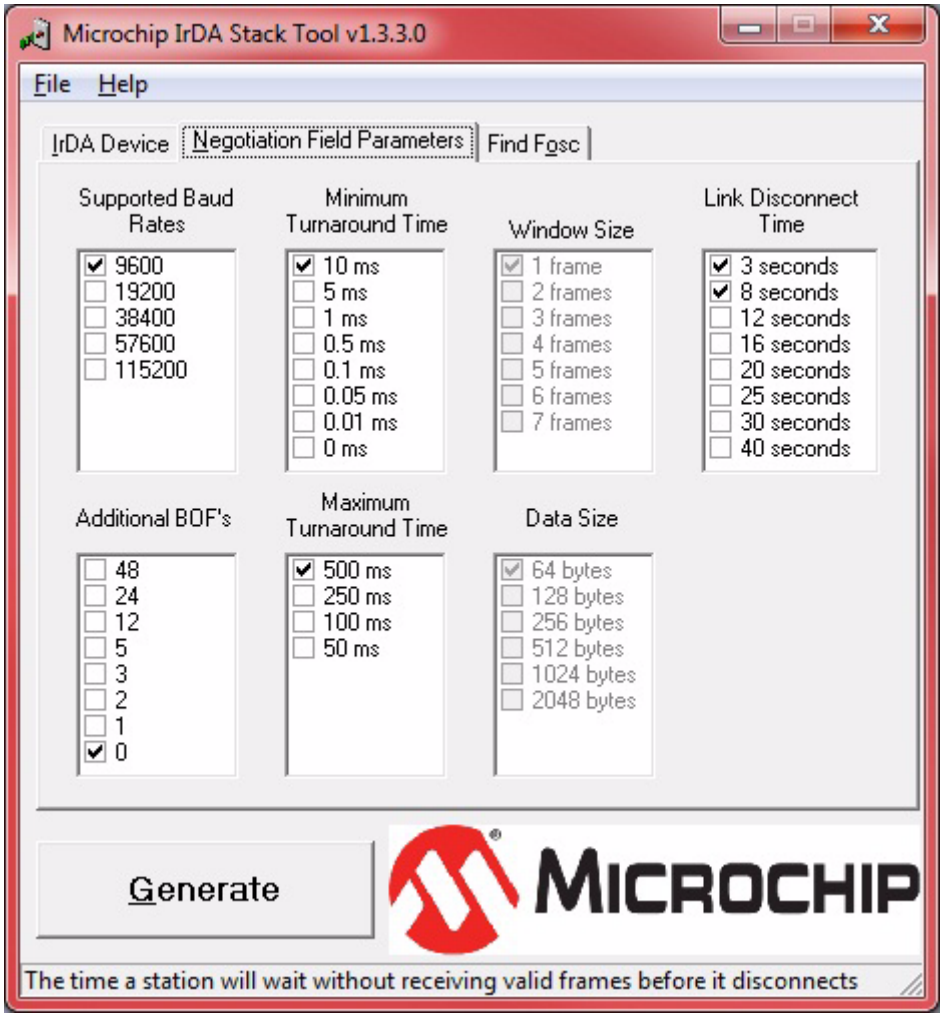
   g) **Link Disconnect Time**: Select the supported link disconnect times.

5. After entering all information, click **Generate** to create the project files, `IrDA_def.h` and `myIrDA.c`.

   If the information contains any errors, a message will be displayed and the files will not be created. Otherwise, you will be prompted for the project directory.

6. Select the project directory for the files, and then click **OK**.

FIGURE 6: CONFIGURING THE NEGOTIATION FIELD PARAMETERS

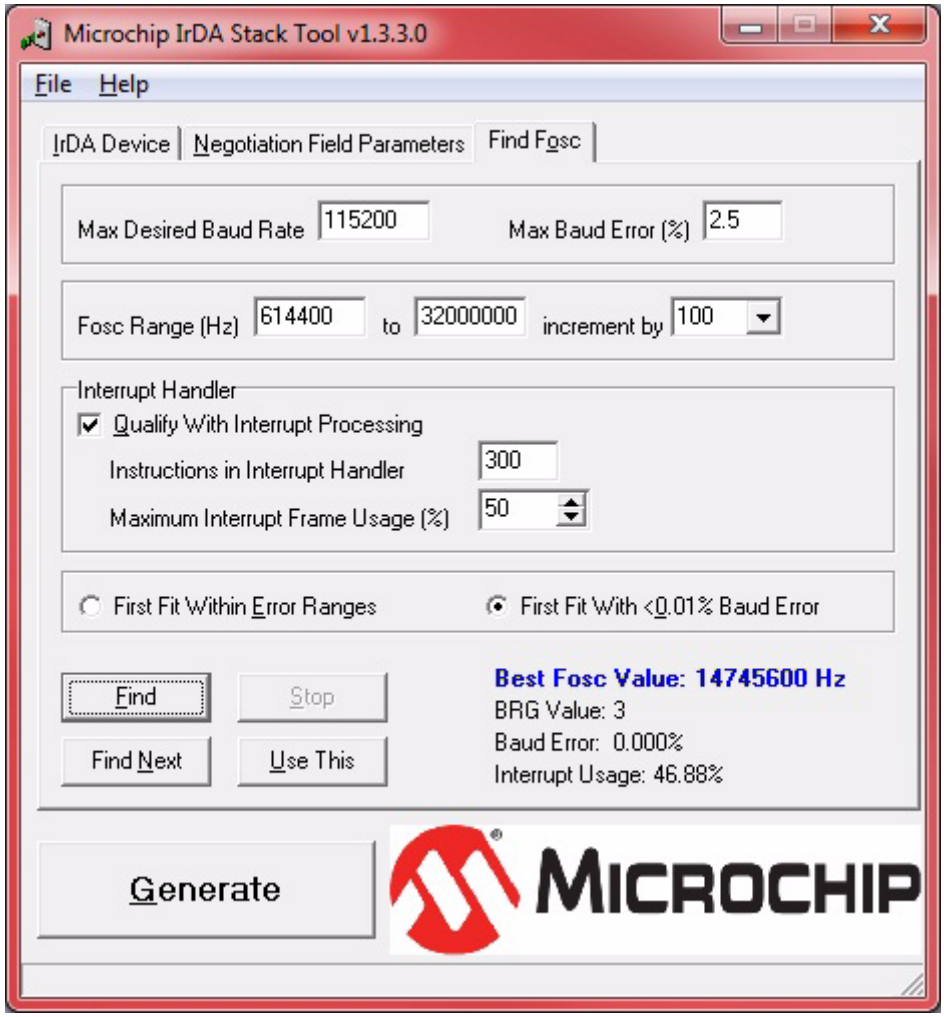


7. Select the **Find Fosc** tab.

   Use this optional tab to find the best oscillator frequency value. Common default settings are provided.

   a) **Max Desired Baud Rate:** Specify the maximum desired baud rate.
   b) **Max Baud Error(%):** Specify the maximum baud error percentage.
   c) **Fosc Range (Hz):** Specify the frequency range of the oscillator.
   d) **Increment by:** Specify the increment value.
   e) **Interrupt Handler:** Select **Qualify with Interrupt Processing** to input the maximum instructions inside the interrupt handler and the maximum interrupt frame usage in percentage.
   f) **Instructions in Interrupt Handler:** Specify the maximum instructions inside the interrupt handler.
   g) **Maximum Interrupt Frame Usage (%):** Specify the maximum interrupt frame usage percentage.
   h) **First Fit:** Select the desired first fit check to be performed.
   i) **Find:** Click **Find** to calculate the value.
   j) **Stop:** Select **Stop** to stop searching for the next value.
   k) **Find Next:** Click **Find** to locate the next nearest value.
   l) **Use This:** Click **Use This** to apply these values into the generated header files.

FIGURE 7: CONFIGURING THE FIND FOSC PARAMETERS



CONCLUSION

The Microchip IrDA Standard Stack provides a modular, easy-to-use set of libraries to add support for an IrDA standard protocol to your application. The low-level drivers allow the Stack to be tailored to the target hardware, while the libraries keep the Stack interface simple. The Microchip IrDA Standard Stack will allow you to add a valuable connectivity aspect to your embedded design.

REFERENCES

- Infrared Data Association web site:
  http://www.irda.org
- Microchip Technology, Inc. web site:
  http://www.microchip.com

APPENDIX A: SOURCE CODE

Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the "Company") is intended and supplied to you, the Company's customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

The libraries and source code files associated with this application note are available for download as a single archive file from the Microchip corporate web site, at:

www.microchip.com

APPENDIX B: IrCOMM 9-WIRE COOKED DATA STRUCTURES

The following data structures are used to store control parameter values that have been received from the remote device. It is recommended that the application utilize the macros described in **Appendix C: "IrCOMM 9-Wire Cooked Control Channel Access Macros"** to access the value of these parameters rather than accessing the variables directly.

The application may also use these data structures to maintain its own control parameters. Refer to the file, `irdep.h`, for constants that can be used when utilizing these structures.

EXAMPLE B-1: SERVICE TYPE PARAMETER STRUCTURE

```
typedef union _IRDA_SERVICE_TYPE
{
    BYTE Val;
    struct _IRDA_SERVICE_TYPE_bits
    {
        unsigned int              : 1;
        unsigned int b3Wire       : 1;
        unsigned int b9Wire       : 1;
        unsigned int bCentronics  : 1;
    } bits;
} IRDA_SERVICE_TYPE;
```

EXAMPLE B-2: DATA FORMAT PARAMETER STRUCTURE

```
typedef union _IRDA_DATA_FORMAT
{
    BYTE Val;
    struct_IRDA_DATA_FORMAT_bits
    {
        unsigned int characterLength  : 2;
        unsigned int stopBits         : 1;
        unsigned int parity           : 2;
    } bits;
} IRDA_DATA_FORMAT;
```

EXAMPLE B-3: CONTROL INDICATIONS PARAMETER STRUCTURE

```
typedef union _IRDA_CONTROL_INDICATIONS
{
    BYTE Val;
    struct _IRDA_CONTROL_INDICATIONS_bits
    {
        unsigned int breakIndication  : 1;
        unsigned int pollLineSettings : 1;
    } bits;
} IRDA_CONTROL_INDICATIONS;
```

EXAMPLE B-4: FLOW CONTROL PARAMETER STRUCTURE

```
typedef union _IRDA_FLOW_CONTROL
{
    BYTE Val;
    struct _IRDA_FLOW_CONTROL_bits
    {
        unsigned int XON_XOFF_input   : 1;
        unsigned int XON_XOFF_output  : 1;
        unsigned int RTS_CTS_input    : 1;
        unsigned int RTS_CTS_output   : 1;
        unsigned int DSR_DTR_input    : 1;
        unsigned int DSR_DTR_output   : 1;
        unsigned int ENQ_ACK_input    : 1;
        unsigned int ENQ_ACK_output   : 1;
    } bits;
} IRDA_FLOW_CONTROL;
```

EXAMPLE B-5: LINE STATUS PARAMETER STRUCTURE

```
typedef union _IRDA_LINE_STATUS
{
    BYTE Val;
    struct _IRDA_LINE_STATUS_bits
    {
        unsigned int              : 1;
        unsigned int OverrunError : 1;
        unsigned int ParityError  : 1;
        unsigned int FramingError : 1;
    } bits;
} IRDA_LINE_STATUS;
```

EXAMPLE B-6: DTE LINE STATUS STRUCTURE

```
typedef union _IRDA_DTE_LINE_STATUS
{
    BYTE Val;
    struct _IRDA_DTE_LINE_STATUS_bits
    {
        unsigned int deltaDTR : 1;
        unsigned int deltaRTS : 1;
        unsigned int DTR      : 1;
        unsigned int RTS      : 1;
    } bits;
} IRDA_DTE_LINE_STATUS;
```

EXAMPLE B-7: DCE LINE STATUS STRUCTURE

```
typedef union _IRDA_DCE_LINE_STATUS
{
    BYTE Val;
    struct _IRDA_DCE_LINE_STATUS_bits
    {
        unsigned int deltaCTS : 1;
        unsigned int deltaDSR : 1;
        unsigned int deltaRI  : 1;
        unsigned int deltaCD  : 1;
        unsigned int CTS      : 1;
        unsigned int DSR      : 1;
        unsigned int RI       : 1;
        unsigned int CD       : 1;
    } bits;
} IRDA_DCE_LINE_STATUS;
```

APPENDIX C: IrCOMM 9-WIRE COOKED CONTROL CHANNEL ACCESS MACROS

The following macros are available to access the control parameters that are received from the remote device. Note that they cannot be used to access control channel variables declared by the application.

TABLE C-1: FLOW CONTROL VALUE MACROS

Macro Name	Description
`IrDA_GetCommStatus_DataRate()`	Data rate of the remote device as an unsigned 32-bit value
`IrDA_GetCommStatus_DataSize()`	Character Length: 5 bits = `0x00` 6 bits = `0x01` 7 bits = `0x02` 8 bits = `0x03`
`IrDA_GetCommStatus_StopBits()`	Stop bits: 1 stop bit = `0` 2 stop bits = `1`
`IrDA_GetCommStatus_Parity()`	Parity Enable and Type: No parity = `0x00` Odd parity = `0x01` Even parity = `0x03` Mark parity = `0x05` Space parity = `0x07`
`IrDA_GetCommStatus_XON()`	XON character
`IrDA_GetCommStatus_XOFF()`	XOFF character
`IrDA_GetCommStatus_ENQ()`	ENQ character
`IrDA_GetCommStatus_ACK()`	ACK character
`IrDA_GetCommStatus_Break()`	Break: Clear break = `0` Set break = `1`
`IrDA_GetCommStatus_PollLineSettings()`	Sender requests line settings and changes = `1` No request = `0`
`IrDA_ClearPollLineSettings()`	Clear poll line settings state; must be done after responding to the request

TABLE C-2: FLOW CONTROL SIGNAL MACROS

Macro Name	Description
IrDA_GetCommStatus_XON_XOFF_input()	XON/XOFF on input
IrDA_GetCommStatus_XON_XOFF_output()	XON/XOFF on output
IrDA_GetCommStatus_RTS_CTS_input()	RTS/CTS on input
IrDA_GetCommStatus_RTS_CTS_output()	RTS/CTS on output
IrDA_GetCommStatus_DSR_DTR_input()	DSR/DTR on input
IrDA_GetCommStatus_DSR_DTR_output()	DSR/DTR on output
IrDA_GetCommStatus_ENQ_ACK_input()	ENQ/ACK on input
IrDA_GetCommStatus_ENQ_ACK_output()	ENQ/ACK on output

TABLE C-3: DTE LINE SETTINGS AND CHANGES MACROS

Macro Name	Description
IrDA_GetCommStatus_deltaDTR()	DTR has not changed = 0 DTR has changed = 1
IrDA_GetCommStatus_deltaRTS()	RTS has not changed = 0 RTS has changed = 1
IrDA_GetCommStatus_DTR()	DTR state
IrDA_GetCommStatus_RTS()	RTS state

TABLE C-4: DCE LINE SETTINGS AND CHANGES MACROS

Macro Name	Description
IrDA_GetCommStatus_deltaCTS()	CTS has not changed = 0 CTS has changed = 1
IrDA_GetCommStatus_deltaDSR()	DSR has not changed = 0 DSR has changed = 1
IrDA_GetCommStatus_deltaRI()	RI has not changed = 0 RI has changed = 1
IrDA_GetCommStatus_deltaCD()	CD has not changed = 0 CD has changed = 1
IrDA_GetCommStatus_CTS()	CTS state
IrDA_GetCommStatus_DSR()	DSR state
IrDA_GetCommStatus_RI()	RI state
IrDA_GetCommStatus_CD()	CD state

APPENDIX D: REVISION HISTORY

Revision A (March 2007)

This is the initial released version of this document.

Revision B (December 2010)

This revision includes the following updated:

- References to 32-bit microcontrollers were added throughout the document
- Added **"32-bit Microcontrollers"**
- Added Example 10
- Updated Figure 5 and Figure 6
- Added an additional step that details usage of the optional FOSC tab in the IrDA Stack tool (see step 7 in **"Microchip's IrDA® Stack Tool"**)

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

Printed on recycled paper.

ISBN: 978-1-60932-736-1

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-6578-300
Fax: 886-3-6578-370

Taiwan - Kaohsiung
Tel: 886-7-213-7830
Fax: 886-7-330-9305

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

08/04/10