IMPART WinXTI Reference Manual

for Microsoft Windows 2000 and later systems

UM6210/00

BOLDON JAMES

IMPART WinXTI UM6210/00 Edition 1 June 2003

Product Support

All contact with Boldon James. regarding this product should be directed to:

Technical Support Unit
Boldon James
1 Westmere Court
Westmere Drive
Crewe
Cheshire
CW1 6ZE
England

Telephone +44 (0) 1270 507810

Fax +44 (0) 1270 507801

Email: support@boldonjames.com

WWW: http://www.boldonjames.com

The support services available are as defined in the current support agreement.

Contents

1 General

1.1 Purpose

This document describes an application interface for IMPART ISO 8073 Transport on Windows 2000 and later systems. WinXTI provides a network programming interface based on the X/Open XTI.

1.2 References

The current version of the XTI is documented in:

[1] *CAE Specification*. *Networking Services (XNS) Issue 5*.
The Open Group. ISBN 1 85912 165 9 (available from www.opengroup.org).

1.3 Terms and Abbreviations

Term	**Meaning**
API	Application Programming Interface.
DDK	Driver Development Kit.
OSI	Open Systems Interconnection.
SNA	Systems Network Architecture. A set of IBM networking standards.
TDI	Transport Driver Interface. A Windows NT in-kernel API specification for transport service providers.
WIN32	The 32 bit Microsoft Windows operating systems (such as Windows NT).
XTI	X/Open's Transport Interface definition.

2 Introduction

The WinXTI programming interface is designed to provide a 32 Bit Microsoft Windows programming interface based on the X/OPEN Transport Interface (XTI) that is independent of underlying transport providers.

WinXTI extends the Microsoft WIN32 API for Windows 2000 and later systems and can be used over IMPART ISO 8073 Transport.

3 THE WinXTI PROGRAMMING INTERFACE

The XTI specification (Ref [1]) is designed to be independent of any specific transport provider. Originally it was primarily concerned with supporting the OSI transport service definition, later being extended to support other transport providers such as TCP UDP and SNA. Additionally it can provide the services required of a higher level API such as NetBIOS. XTI therefore provides an interface suitable for numerous underlying providers and protocols.

XTI was designed for the UNIX operating system. WinXTI extends the XTI interface to provide an API for use under Microsoft Windows 2000 and later systems.

3.1 Deviations from XTI

Where possible, WinXTI follows the XTI specification (Ref [1]). This WinXTI specification defines only the changes and extensions to XTI. A programmer using WinXTI must also have access to the XTI specification.

The following sections describe the major deviations from XTI. The descriptions of the library interface functions in section 5 deal with function specific differences.

3.1.1 Transport endpoint data type

In UNIX, the data type representing a transport service endpoint (returned by **t_open()**) is a file descriptor, a signed integer that always has a positive value for a valid endpoint. In WinXTI, the unsigned data type, **TEP** defines a transport endpoint. A **TEP** can be any value between 1 and INVALID_TEP - 1. The value INVALID_TEP indicates an invalid endpoint.

The WIN32 **TEP** is a file handle, and can hence be used in file I/O operations such as **read()**, **write()**, **WriteFile()** and **ReadFile()**. However the endpoint **must** be closed by calling the appropriate WinXTI APIs completed by t_close(). Simply closing the file handle via **close()**, **CloseHandle()** or implicitly by process termination, will not correctly close the endpoint or free WinXTI resources.

A WIN32 WinXTI **TEP**, can be used by another process (following file handle duplication or inheritance) but the user must contend with synchronisation locking between processes sharing an endpoint. When porting UNIX applications that attempt to pass an endpoint to another process, it is recommended that the application is changed to use new thread rather than a new process.

3.1.2 poll()

UNIX XTI applications often make use of the UNIX system call, **poll()**, for multiplexing input and output over a set of file descriptors. There is no equivalent function in WIN32 or WinXTI. Applications should replace the use of poll() by threads performing blocking calls and utilise WIN32 event handles to signal completion of the calls if required.

3.1.3 Error values

Error codes are not made available via the external integers t_errno or errno. The functions **WtGetLastTerror()** and **WtGetLastError()** will return the last WinXTI error and system error respectively for the current thread.

3.1.4 Blocking routines

In the current version, it is recommended that blocking operations should be used. Where asynchronous behaviour is required this can be implemented by the application launching a new thread to perform a blocking operation and using WIN32 event handles to signal completion.

3.1.5 Maximum number of transport endpoints

The maximum number of transport endpoints supported may be limited by the underlying transport provider. The number supported by WinXTI may be obtained by the API **WtStartup()**. Applications should make no assumptions about the availability of a particular number of endpoints.

3.1.6 Include files

The header file **winxti.h** is the only WinXTI specific header file required by applications.

For WIN32, the file **tdiaddr.h** is supplied. This contains extracts from NT DDK header files that define the transport address structures used by TDI transport providers. Applications that are not concerned with address structures, perhaps by using a name server DLL, will not require this header file. Applications that use DDK header files such as tdi.h should not include this file to avoid header file conflicts.

3.1.7 Return values on WinXTI function error

For future portability, the constant WXTI_ERROR is provided for checking failure of WinXTI calls. Rather than testing for -1 or < 0, the programmer should explicitly test for the return value WXTI_ERROR.

3.1.8 Multi-threaded Windows and WinXTI

In a multi-threaded environment, the author of a multi-threaded application must be aware that it is the responsibility of the application, not WinXTI, to synchronise access to an endpoint between threads. This is the same rule as applies to other form of I/O such as file I/O. Failure to synchronise calls to an endpoint leads to unpredictable results; for example if there are two simultaneous calls to **t_send()**, there is no guarantee as to the order the data will be sent.

Internally, WinXTI synchronises access to its internal data structures between threads and processes.

4 WinXTI LIBRARY

4.1 WinXTI Functions

XTI Function	Description
t_accept	Accepts a request for a transport connection.
t_alloc	Allocates XTI data structures.
t_bind	Binds a transport address to a transport endpoint.
t_close	Closes a transport endpoint.
t_connect [1]	Establishes a connection with the transport user at a specified destination.
t_error	A dummy function in WinXTI
t_free	Frees structures allocated using t_alloc.
t_getinfo	Returns a set of parameters associated with a transport provider.
t_getprotaddr	Gets protocol addresses associated with a transport endpoint.
t_getstate	Returns the state of a transport endpoint,
t_listen [1]	Retrieves an indication of a connect request from another transport user.
t_look	Returns the current event on a transport endpoint.
t_open	Establish a transport endpoint with a chosen transport provider.
t_optmgmt [2]	Negotiates protocol specific options with the transport provider.
t_rcv [1]	Retrieves data from a transport connection.
t_rcvconnect	Completes connection establishment if t_connect was called in asynchronous mode.
t_rcvdis	Returns an indication of an aborted connection, including an orderly release of a connection.

WinXTI Functions (cont.)

XTI Function	Description
t_rcvudata [1]	Retrieves a connectionless message.
t_rcvuderr	Retrieves error information associated with a previously sent connectionless message.
t_snd [1]	Sends data over an established transport connection.
t_snddis	Aborts a connection or rejects a connect request.
t_sndrel	Requests the orderly release of a connection.
t_sndudata [1]	Sends a connectionless message to the specified destination user.
t_strerror	Produces an error message string.
t_sync	Synchronises a transport endpoint with the transport provider.
t_unbind	Releases a transport endpoint, associated resources are freed.
WinXTI Extensions	
WtCleanup [3]	A dummy function.
WtFcntl	Gets or sets the mode of a transport endpoint.
WtGetLastError	Retrieves the last WinXTI system or transport provider error.
WtGetLastTerror	Retrieves the last WinXTI API error (t_error).
WtSetLastError	Sets the error to be returned by a subsequent **WtGetLastError()**.
WtSetLastTerror	Sets the error to be returned by a subsequent **WtGetLastTerror()**.
WtStartup	Retrieves WinXTI Version information.

[1] *These functions may block if acting on a blocking endpoint.*

[2] *Returns TNOTSUPPORT.*

[3] *Provided for symmetry with WtStartup(), has no functionality.*

4.2 Blocking and Non-Blocking Usage

The default behaviour of a transport endpoint under UNIX is to operate in blocking mode unless the programmer specifically requests the endpoint to act in non-blocking mode.

The functions marked [1] above may block if the endpoint is operating in blocking mode.

If an application invokes an asynchronous or non-blocking operation that takes a pointer to a memory object (e.g. a buffer or a global variable) as an argument, it is the responsibility of the application to ensure the object is available to WinXTI throughout the operation. The application must not invoke any Windows function that might affect the mapping or addressability of the memory involved. In a multithreaded system, the application is also responsible for co-ordinating access to the object using appropriate synchronisation mechanisms.

4.3 Error Handling

For compatibility with thread-based environments, details of errors are obtained through **WtGetLastTerror()** and **WtGetLastError()**. The normal UNIX method of obtaining XTI errors via "t_error" and "errno" cannot be guaranteed to be reliable in a multi-threaded environment. These functions return the last WinXTI and system errors on a thread. **WtGetLastTerror()** returns the last WinXTI API error (t_error), if this is set to TSYSERR, then a system or transport provider error has occurred that is obtained by calling **WtGetLastError()**. Programmers porting UNIX XTI applications should replace uses of t_error and errno by calls to these APIs.

The errors returned by **WtGetLastTerror()** are defined in **winxti.h**. Errors returned by **WtGetLastError()** are either WIN32 system errors or errors returned by the transport provider.

4.4 Address Format and Options

The AddressType used is TDI_ADDRESS_TYPE_OSI_TSAP. The format of the Address is:

length (bytes):	1	0-20	1	0-32
value:	NSAP length	NSAP	TSEL length	TSEL

To use null CLNP, specify an NSAP of the form:

498nnneeeeeeeeeeeellss

where:

nnn is the subnetwork identifier;

eeeeeeeeeeee is the Ethernet address;

ll is the LSAP identifier.

ss is a network selector, conventionally 01.

The options supported in the isoco_options structure are:

mngmt.extform	extended formats

mngmt.checksum	checksums
expd	expedited data

5 WinXTI LIBRARY FUNCTIONS AND PARAMETERS

5.1 XTI functions.

The following sections describe only the interface for each function and changes from the XTI definition (Ref [1]). In most cases the only changes are in the data types. Changes that apply to all functions are:

- File descriptors are replaced by a transport end point (**TEP** datatype).
- Error return values should be tested against the constant WXTI_ERROR rather than -1.
- Error values are those defined in winxti.h.

5.1.1 t_accept()

NAME

t_accept - accept a connection request

SYNOPSIS

#include <winxti.h>

TEP WINAPI t_accept (TEP *fd*, TEP *resfd*, struct t_call FAR **call*)

PARAMETERS

fd

Local transport endpoint where the connection arrived.

resfd

Local transport endpoint where the connection is to be established.

call

Pointer to a **t_call** structure.

5.1.2 **t_alloc()**

NAME

t_alloc - allocate a library structure

SYNOPSIS

#include <winxti.h>

char WINAPI * t_alloc (TEP *fd*, int *struct_type*, int *fields*)

PARAMETERS

fd

Local transport endpoint.

struct_type

The type of structure to allocate.

fields

Specifies which buffers to allocate.

5.1.3 t_bind()

NAME

t_bind - bind an address to a transport endpoint

SYNOPSIS

#include <winxti.h>

char WINAPI * t_bind (TEP *fd*, struct t_bind FAR **req*,
struct t_bind FAR **ret*)

fd

Local transport endpoint.

req

Points to a **t_bind** structure used to request an address to be bound to the local endpoint.

ret

Point to a **t_bind** structure to contain the actual bound address on return.

5.1.4 **t_close()**

NAME

t_close - close a transport endpoint

SYNOPSIS

#include <winxti.h>

int WINAPI t_close (TEP *fd*)

PARAMETERS

fd

Local transport endpoint.

5.1.5 **t_connect()**

NAME

t_connect - establish a connection with another transport user

SYNOPSIS

#include <winxti.h>

int WINAPI t_connect (TEP *fd*, struct t_call FAR **sndcall*,
struct t_call FAR **rcvcall*)

PARAMETERS

fd

Local transport endpoint.

sndcall

Specifies information needed by the transport provider to establish a connection.

rcvcall

Contains information associated with the newly established connection on return.

5.1.6 t_error()

NAME

t_error - a dummy function in WinXTI

SYNOPSIS

#include <winxti.h>

int WINAPI t_error (const char FAR **errmsg*)

No action is taken by this function. A value of zero is always returned.

5.1.7 **t_free()**

NAME

t_free - free a library structure

SYNOPSIS

#include <winxti.h>

int WINAPI t_free (char FAR **ptr*, int FAR *struct_type*)

PARAMETERS

ptr

Points to a structure type previously allocated by **t_alloc()**.

struct_type

Identifies the type of structure to be freed.

5.1.8 t_getinfo()

NAME

t_getinfo - get protocol-specific service information

SYNOPSIS

#include <winxti.h>

int WINAPI t_getinfo (TEP *fd*, struct t_info **info*)

PARAMETERS

fd

Local transport endpoint.

info

Points to a **t_info** structure that on return will contain characteristics of the underlying transport provider.

5.1.9 **t_getprotaddr()**

NAME

t_getprotaddr - get the protocol addresses

SYNOPSIS

#include <winxti.h>

int WINAPI t_getprotaddr (TEP *fd*, struct t_bind FAR **boundaddr*, struct t_bind FAR **peeraddr*)

PARAMETERS

fd

Local transport endpoint.

boundaddr

Points to a t_bind structure that on return will contain the local bound endpoint protocol address.

peeraddr

Points to a t_bind structure that on return will contain the remote peer entity protocol address.

5.1.10 **t_getstate()**

NAME

t_getstate - get the current state

SYNOPSIS

#include <winxti.h>

int WINAPI t_getstate (TEP *fd*)

PARAMETERS

fd

Local transport endpoint.

5.1.11 t_listen()

NAME

t_listen - listen for a connection indication

SYNOPSIS

#include <winxti.h>

int WINAPI t_listen (TEP *fd*, struct t_call **call*)

PARAMETERS

fd

Local transport endpoint.

call

Points to a **t_call** structure to contain information describing the connection indication on return.

5.1.12 **t_look()**

NAME

t_look - look at the current event on a transport endpoint

SYNOPSIS

#include <winxti.h>

int WINAPI t_look (TEP *fd*)

PARAMETERS

fd

Local transport endpoint.

5.1.13 t_open()

NAME

t_open - establish a transport endpoint

SYNOPSIS

#include <winxti.h>

TEP WINAPI t_open (char FAR **name*, int *oflag*, struct t_info FAR **info*)

PARAMETERS

name

Transport provider identifier, must be "\\\\.\\Tbh_Bjosi" (representing \\.\Tbh_Bjosi).

oflag

Mode flag for opening.

info

Points to a **t_info** structure to return various default characteristics of the underlying transport protocol.

5.1.14 **t_optmgmt()**

NAME

t_optmgmt - manage options for a transport endpoint

SYNOPSIS

#include <winxti.h>

int WINAPI t_optmgmt (TEP *fd*, struct t_optmgmt FAR **req*, struct t_optmgmt FAR **ret*)

PARAMETERS

fd

Local transport endpoint.

req

Points to a **t_optmgmt** structure containing fields to request certain actions of the provider.

ret

Points to a **t_optmgmt** structure containing fields to be returned to the user indicating options actioned.

DESCRIPTION

This function is not implemented and returns TNOTSUPPORT.

5.1.15 t_rcv()

NAME

t_rcv - receive data or expedited data sent over a connection

SYNOPSIS

#include <winxti.h>

int WINAPI t_rcv (TEP *fd*, const char FAR **buf*, int *nbytes*, int FAR **flags*)

PARAMETERS

fd

Local transport endpoint.

buf

Points to a buffer for the received data.

nbytes

The size of the receive buffer.

flags

On return, indicates if there is more data in the TSDU message, and/or it is expedited data.

5.1.16 t_rcvconnect()

NAME

t_rcvconnect - receive the confirmation from a connection request

SYNOPSIS

#include <winxti.h>

int WINAPI t_rcvconnect (TEP *fd*, struct t_call FAR **call*)

PARAMETERS

fd

Local transport endpoint.

call

Points to a **t_call** structure to contain, on return, the protocol address, options and user data associated with the newly established remote endpoint.

5.1.17 t_rcvdis()

NAME

t_rcvdis - retrieve information from disconnection

SYNOPSIS

#include <winxti.h>

int WINAPI t_rcvdis (TEP *fd*, struct t_discon FAR **discon*)

PARAMETERS

fd

Local transport endpoint.

discon

Points to a **t_discon** structure to retrieve the cause and user data sent with a disconnection.

5.1.18 **t_rcvudata()**

NAME

t_rcvudata - receive a data unit

SYNOPSIS

#include <winxti.h>

int WINAPI t_rcvudata (TEP *fd*, struct t_unitdata FAR **unitdata*, int FAR **flags*)

PARAMETERS

fd

Local transport endpoint.

unitdata

Points to a t_unitdata structure to contain the protocol address, data and options associated with the received data.

flags

Set on return to indicate if more of the unit data message remains to be read.

5.1.19 t_rcvuderr()

NAME

t_rcvuderr - receive a unit data error indication

SYNOPSIS

#include <winxti.h>

int WINAPI t_rcvuderr (TEP *fd*, struct t_uderr FAR **uderr*)

PARAMETERS

fd

Local transport endpoint.

uderr

Points to a **t_uderr** structure to receive the information concerning an error on a previously sent data unit.

5.1.20 **t_snd()**

NAME

t_snd - send data or expedited data over a connection

SYNOPSIS

#include <winxti.h>

int WINAPI t_snd (TEP *fd*, const char FAR **buf*, unsigned int *nbytes*, int *flags*)

PARAMETERS

fd

Local transport endpoint.

buf

Points to a buffer containing the data to be sent.

nbytes

The number of bytes of user data to be sent.

flags

Indicates if there is more data in the TSDU message, and/or it is expedited data.

5.1.21 t_snddis()

NAME

t_snddis - send user-initiated disconnection request

SYNOPSIS

#include <winxti.h>

int WINAPI t_snddis (TEP *fd*, struct t_call FAR **call*)

PARAMETERS

fd

Local transport endpoint.

call

Points to a t_call structure containing information associated with the disconnection.

5.1.22 t_sndrel()

NAME

t_sndrel - initiate an orderly release

SYNOPSIS

#include <winxti.h>

int WINAPI   t_sndrel (TEP *fd*)

PARAMETERS

fd

Local transport endpoint.

5.1.23 **t_sndudata()**

NAME

t_sndudata - send a data unit

SYNOPSIS

#include <winxti.h>

int WINAPI t_sndudata (TEP *fd*, struct t_unitdata FAR **unitdata*)

PARAMETERS

fd

Local transport endpoint.

unitdata

Points to a **t_unitdata** structure specifying the remote protocol address, user data and options.

5.1.24 t_strerror()

NAME

t_strerror - produce an error message string

SYNOPSIS

#include <winxti.h>

char WINAPI * t_strerror (int *errnum*)

PARAMETERS

errnum

The WinXTI error value.

5.1.25 t_sync()

NAME

t_sync - synchronise transport library

SYNOPSIS

#include <winxti.h>

int WINAPI t_sync (TEP *fd*)

PARAMETERS

fd

Local transport endpoint.

DESCRIPTION

In WIN32, **t_sync()** is provided for compatibility only, and has no functionality.

5.1.26 **t_unbind()**

NAME

t_unbind - disable a transport endpoint

SYNOPSIS

#include <winxti.h>

int WINAPI t_unbind (TEP *fd*)

PARAMETERS

fd

Local transport endpoint.

5.2 Microsoft Windows Specific Extensions

5.2.1 WtCleanup()

NAME

WtCleanup - Terminate use of WinXTI DLL.

SYNOPSIS

```
#include <winxti.h>

int WINAPI          WtCleanup (void)
```

DESCRIPTION

WtCleanup() is provided for symmetry with WtStartup() and contains no functionality.

RETURN VALUE

Upon successful completion, a value of 0 is returned, otherwise WXTI_ERROR is returned.

5.2.2 **WtFcntl()**

NAME

WtFcntl - control the mode of a transport endpoint

SYNOPSIS

#include <winxti.h>

int WINAPI WtFcntl (TEP *fd*, long *cmd*, u_long FAR **argp*)

PARAMETERS

fd

Local transport endpoint.

cmd

The command to perform on the endpoint.

argp

Points to the parameter for cmd.

DESCRIPTION

This routine may be used on a transport endpoint in any state. It is used to alter the blocking mode of the endpoint. It is a subset of the UNIX **fcntl()** system call.

Command	**Semantics**
F_GETFL	Get the mode of the endpoint *fd*. *argp* points to a bit mask of flags indicating the mode of the endpoint.
F_SETFL	Set the blocking mode of the endpoint *fd*. *argp* points to the flags that set the blocking mode of the endpoint. Note that only the blocking mode can be set by this call.

The flags pointed to by argp can have the following (hexadecimal) values:

Flag Name	**Value**	**Meaning**
O_RDWR	2	Read and write
O_NDELAY	4	Non-blocking I/O
O_NONBLOCK	80	Non-blocking I/O

No other values are applicable to WinXTI.

RETURN VALUE

The value 0 is returned if the operation was successful, otherwise the value WXTI_ERROR is returned.

5.2.3 **WtGetLastError()**

NAME

WtGetLastError() - get the system error value for the last failed operation

SYNOPSIS

```
#include <winxti.h>

int WINAPI          WtGetLastError (void)
```

DESCRIPTION

This function returns the last system error that occurred for the calling thread. When a WinXTI library function fails, and the XTI error is **TSYSERR,** indicating a system error, then **WtGetLastError()** should be called immediately to obtain the system error value. This function simply returns the result of the WIN32 API **GetLastError()**.

A system error may be either a WIN32 system error, or a transport provider specific error.

When porting UNIX XTI applications, all references to **errno** as a result of the XTI error **T_SYSERROR** should be replaced by a call to **WtGetLastError()**.

RETURN VALUE

The error value for the last system error on this thread.

5.2.4 WtGetLastTerror()

NAME

WtGetLastError() - get the WinXTI error value for the last failed operation

SYNOPSIS

```
#include <winxti.h>

int WINAPI          WtGetLastTerror (void)
```

DESCRIPTION

This function returns the last WinXTI API error that occurred for the calling thread.

RETURN VALUE

The error value for the last WinXTI API error on this thread.

5.2.5 **WtSetLastError()**

NAME

WtSetLastError - set the error code to be retrieved by **WtGetLastError()**

SYNOPSIS

#include <winxti.h>

void WINAPI WtSetLastError (int *iError*)

PARAMETER

iError

The error code to be returned by a subsequent **WtGetLastError()**.

DESCRIPTION

In a WIN32 environment this function will call **SetLastError()**. This sets the error to be returned by a subsequent call to **WtGetLastError()** for the current thread. Note that any subsequent WinXTI routine called by the thread for the same endpoint may re-set the error set by **WtSetLastError()**.

RETURN VALUE

None.

5.2.6 WtSetLastTerror()

NAME

WtSetLastTerror - set the error code to be retrieved by **WtGetLastTerror()**

SYNOPSIS

#include <winxti.h>

void WINAPI WtSetLastTerror (int *iError*)

PARAMETER

iError

The error code to be returned by a subsequent **WtGetLastTerror()**.

DESCRIPTION

Sets the error to be returned by a subsequent call to **WtGetLastTerror()** for the current thread. Note that any subsequent WinXTI routine called by the thread for the same endpoint may re-set the error set by **WtSetLastTerror()**.

RETURN VALUE

None.

5.2.7 **WtStartup()**

NAME

WtStartup - start a WinXTI session.

SYNOPSIS

#include <winxti.h>

int WINAPI WtStartup (WORD *wVersionRequested*,
WXTIData * *lpWXTIData*)

PARAMETERS

wVersionRequested

The highest version of WinXTI that the caller can use.

lpWXTIData

Points to the WXTIData structure to receive details of the WinXTI implementation.

DESCRIPTION

On WIN32, this function and its complementary function **WtCancel()** are optional.

The *lpWXTIData* structure may be a NULL pointer.

6 IMPART Tracing

If you wish to include tracing into your application you can make use of IMPART's in-built tracing facility called IMPART Tracing. This facility supports the logging of messages with varying “trace levels” and, under the control of IMPART Tracing, you can specify the trace level you wish to monitor and that all messages at or above that level will be output to a log file or window. You do not have to log any messages yourself as IMPART Tracing can be used to log IMPART’s own messages which include calls on the WinXTI API. For more information on IMPART Tracing see its on-line Help.

If you plan to use tracing, you should link the file bjlog.lib.

In order to register your application with IMPART Tracing you need to call the BJLogInitialise function early on in your application, and BJLogTerminate at the end. Messages can be logged using BJLogMessage. These functions are specified below.

The messages logged by IMPART Tracing are controlled by settings in the registry. When you run your application for the first time, the registry entries are created for you with default values. You can then use the IMPART Tracing utility to choose the settings you require. The registry key used is:

> HKEY_LOCAL_MACHINE\SOFTWARE\Boldon James\<application name>\Diagnostics

with the following String entries (in the format "Value name"="Value data"):

> "Date"="OFF"
>
> "DeBuggingWindow"="OFF"
>
> "FileInfo"="OFF"
>
> "LogFile"="c:\temp\foo.log"
>
> "LoggingLevel"="INFO"
>
> "LogToFile"="OFF"
>
> "MonitorWindow"="<title>"

Where <application name> is the name you passed to BJLogInitialise (see below). IMPART Tracing will use (and update) these registry settings.

6.1 BJLogInitialise()

NAME

BJLogInitialise - start logging

SYNOPSIS

#include <bjlog.h>

BOOL WINAPI BJLogInitialise (LPCSTR *lpAppId*, LPCSTR *loc*, LPCSTR *sect*, BOOL *Reg*)

PARAMETERS

lpAppID

Defines the name for your application, this name will be displayed by IMPART Tracing. A suggested convention is to supply the name of your application. N.B.: You should only pass one word (no spaces) for this parameter, e.g. "xtitest" and this should also be the same as the name used in loc below.

loc

Defines the Registry Key under HKEY_LOCAL_MACHINE that holds the root of the logging information for the above named application. You must use the IMPART convention, e.g. “SOFTWARE\\Boldon James\\xtitest".

sect

Specifies a registry sub-key name that holds the logging information. You must use the name 'Diagnostics' for compatibility with IMPART Tracing.

Reg

This BOOL value should be set to TRUE.

DESCRIPTION

If your application wishes to use the IMPART Tracing facility then it must be registered by this function. IMPART Tracing enables you to control the level of WinXTI tracing and whether information is logged to a file, a window, or both.

RETURN VALUE

TRUE if successful, FALSE otherwise.

6.2 BJLogMessage()

NAME

BJLogMessage - log a trace message

SYNOPSIS

#include <bjlog.h>

BOOL WINAPI BJLogMessage (LPCSTR *lpAppId*, LPCSTR *lpFile*, WORD *wLineNo*, WORD *wMsgType*, LPCSTR *lpErrorStr* [,*argument*]...);

PARAMETERS

lpAppid

See above.

lpFile

Conventionally the name of your source file.

wLineNo

Conventionally the line number within your source file.

wMsgType

One of: FATAL, NONFATAL, WARNING, WARNING, INFO, LEVEL1, LEVEL2, … , LEVEL11, in decreasing severity.

lpErrorStr

The actual message.

argument

You can use printf-style formatting in the message.

DESCRIPTION

Your application must call this function to log a trace message.

RETURN VALUE

TRUE if successful, FALSE otherwise.

6.3 BJLogTerminate()

NAME

BJLogTerminate - terminate logging

SYNOPSIS

```
#include <bjlog.h>

BOOL WINAPI        BJLogTerminate(void)
```

DESCRIPTION

Your application should call this function before exiting.

RETURN VALUE

TRUE if successful, FALSE otherwise.