



Intelligent Recording API for the Multiport Xtension Recorder

Authorized for release by:

.....
Philip D. Hill

Author	: Andrew Roberts
E-mail of author	: andrew.roberts@irl-recording.com
Issue of specification	: 5
Date of specification	: August 6 th 2010

This work is copyright © 2003-2010 Intelligent Recording Ltd. No part of it may be reproduced or used in any form without the express permission in writing of an authorized officer of Intelligent Recording.

These restrictions extend to all media in which this information may be contained.



REVISION HISTORY

Release	Date	Author	Comment
Alpha 1	13-xi-2003	Nigel Gaunt	First release
2	30-iii-2004	Nigel Gaunt	Updated with separate audio gain
3	08-vii-2010	Andrew Roberts	Changed Comvurgent to IRL and fixed table of contents
4	06-vii-2010	Andrew Roberts	Added MP_GetCallInfo and events
5	06-vii-2010	Andrew Roberts	Updated list of Sw types and codecs



Table Of Contents:

1. Overview	5
2. Deliverables	5
3. Interface Function List.....	6
Functions which affect the entire system.....	6
Functions to obtain a list of connected devices	6
Functions which affect a single connected port	6
4. Interface Function Description.....	7
void MP_Open(void).....	7
void MP_Close(void)	7
void MP_SetCallback((void *MyProc)(int SerialNumber, int Event, int Data))	7
LPCSTR MP_GetSwType(void)	7
LPCSTR MP_GetSwVersion(void)	9
int MP_GetDataFormat(void).....	9
short MP_ConvertData(const char Data)	9
int MP_GetMaxNumPorts (void).....	10
int MP_GetIfSerialNo(int Virtual)	10
int MP_GetStatus(int SerialNumber)	11
int MP_GetCallState(int SerialNumber).....	12
const char * MP_GetIfType(int SerialNumber)	12
void MP_AudioMonitor(int SerialNumber, int WhatToMonitor)	13
void MP_AutoRecord(int SerialNumber, int WhatToMonitor)	13
int MP_GetAudioData(int Port, int MaxLength, unsigned char *pPBXData, unsigned char *pPhoneData, unsigned char *pMergedData).....	13
void MP_SetAudioGain(int SerialNumber, BOOL Remote, int Gain)	14
void MP_ControlMonitor(int SerialNumber, BOOL OnOff)	14
int MP_GetControlDataPBX(int SerialNumber, int MaxLength, unsigned char *pPBXData).....	14
int MP_GetControlDataPhone(int SerialNumber, int MaxLength, unsigned char *pPhoneData).....	15
LPCSTR MP_GetCallerID(int SerialNumber)	15
LPCSTR MP_GetCallInfo(int SerialNumber, int Info) *	15
5. Callbacks.....	16
void MyProc(int SerialNumber, int Event, int Data)	16
MP_STATUS.....	16
MP_BCHANNELDATA	17
MP_DCHANNELDATAPBX.....	17
MP_DCHANNELDATAPHONE	17
MP_BCHANNELOVERFLOW	17
MP_DCHANNELOVERFLOWPBX.....	17
MP_DCHANNELOVERFLOWPHONE	17
MP_CALLSTATE	17
MP_CALLERIDUPDATE.....	17



MP_UIIUPDATE	18
MP_CALLEDPARTYIDUPDATE	18
MP_CONNECTEDNOUPDATE	18
6. Changes from the single port API.....	19



1. Overview

This document describes the programming API which Intelligent Recording will make available to allow the Intelligent Recording interface hardware to be used with your own applications.

This version is for the multiport recorder. Which supports numerous devices simultaneously on the same PC.

2. Deliverables

Intelligent Recording will provide a device driver for the Recorder units. A separate device driver is used for Windows 98/ME, and for Windows XP/2000. The Windows XP/2000 device driver is not currently Microsoft validated.

The low level interface to the telephone will be provided as a Windows Dynamic Link Library. Intelligent Recording will provide a header file, a library definition file, and a DLL file which contain the code necessary to allow use of the Intelligent Recording Recorder hardware.

These files are guaranteed to be compatible with the latest version of Microsoft C++ Developer Studio. No guarantees are made for other systems.

The Header file will be MultiXtR.h, and will contain all the definitions in section 3 of this document.

The library file will be MultiXtR.lib. This contains definitions used for importing the DLL at compile time.

The Dynamic Link Library will be MultiXtR.DLL. This will contain the code necessary for driving the appropriate Intelligent Recording interface. Firmware for the Intelligent Recording interface unit is contained within the DLL file. It does not require a separate file.

There will be different versions of the DLL file for different telephone interfaces. The Lib and Header files for all interfaces will be identical.

3. Interface Function List

Functions which affect the entire system

MP_Open
MP_Close
MP_GetSwType
MP_GetSwVersion
MP_SetCallback
MP_GetDataFormat
MP_ConvertData

Functions to obtain a list of connected devices

MP_GetMaxNumPorts
MP_GetIfSerialNo

Functions which affect a single connected port

MP_GetStatus
MP_GetCallState
MP_GetIfType
[MP_SetAudioGain](#)
MP_GetAudioData
MP_AudioMonitor
MP_AutoRecord
MP_ControlMonitor
MP_GetControlDataPBX
MP_GetControlDataPhone
MP_GetCallerID

Optional functions not supported in all DLL's

MP_GetCallInfo



4. Interface Function Description

Notes

MP_Open and MP_Close should be called from the same thread as each other. This may be, but is not necessarily, the main windows thread. All other functions are thread safe and may be called from any thread at any time. Function calls are protected internally to prevent any data corruption between threads.

void MP_Open(void)

This function will attempt to open the link to the Intelligent Recording hardware. Since it cannot immediately determine success or failure, this function has no return value. The MP_GetStatus function should be called after MP_Open to determine whether required devices are correctly connected.

void MP_Close(void)

This function will close the link to the Intelligent Recording hardware. It will release the usb port and all allocated software devices, returning them to the Windows system.

void MP_SetCallback((void *MyProc)(int SerialNumber, int Event, int Data))

This function sets a callback which will be used by the Intelligent Recording library to identify significant events taking place on the telephone lines. The events provided to the callback are described in section 5 of this document. MyProc may be NULL, which disables the further sending of callbacks.

Note that callbacks will come from an independent thread, so should not try and draw to the Windows screen.

LPCSTR MP_GetSwType(void)

This function will return the version type of the current software library. This is a Intelligent Recording product code, which is a string containing manufacturer, hardware interface and audio encoding type. This string consists of 5 characters e.g. AVfUS, TOfUK. Currently supported types are as follows. There may be more in the future.



Digital Products

Position	Chars	Meaning
1 st and 2 nd	AV	Avaya Definity
1 st and 2 nd	AM	Avaya Magix (Merlin Magix)
1 st and 2 nd	IT	Intertel
1 st and 2 nd	LG	Lucky Goldstar (badged Vodavi in USA)
1 st and 2 nd	NE	NEC. Electra Elite, NEAX & Aspire
1 st and 2 nd	NM	Nortel Meridian and Nortel Option
1 st and 2 nd	NO	Nortel Norstar
1 st and 2 nd	PA	Panasonic DBS
1 st and 2 nd	PK	Panasonic KXTD and TDA
1 st and 2 nd	SA	Samsung DCS
1 st and 2 nd	SI	Siemens HiCom and HiPath (also Telrad + Tadiran)
1 st and 2 nd	TO	Toshiba DK and CTX
3 rd	f	Digital Phone System
4 th and 5 th	US	United States
4 th and 5 th	UK	United Kingdom

ISDN Products

Position	Chars	Meaning
1 st to 5 th	BRIUK	Basic Rate ISDN
1 st to 5 th	PRIUK	Primary Rate ISDN

IP Products

Position	Chars	Meaning
1 st and 2 nd	NE	NEC
1 st and 2 nd	CI	Cisco
1 st and 2 nd	AV	Avaya
1 st and 2 nd	NO	Nortel
1 st and 2 nd	GE	Generic
3 rd	I	IP Phone System
4 th and 5 th	SP	SIP Protocol

Analog Products

Position	Chars	Meaning
1 st to 3 rd	UA2	Universal Adaptor 2
1 st to 2 nd	UK	Universal Adaptor 3

**LPCSTR MP_GetSwVersion(void)**

This function will return the version number of the current software library. This is an ASCII string of the format “nn.nn.nn”

int MP_GetDataFormat(void)

This function will return the data format being used by the attached telephone system. Current possible values are:

Format	Description
WAVE_FORMAT_MULAW	ITU-T G.711 μ -law (USA/Japan)
WAVE_FORMAT_ALAW	ITU-T G.711 A-law (Europe/RoW)
WAVE_FORMAT_G729A	ITU-T G.729 Annex A
WAVE_FORMAT_G722_ADPCM	ITU-T G.722 (7Khz)
WAVE_FORMAT_ADPCM	ADPCM (16KHz)

These values are defined in the Windows system file <MMREG.H>.

short MP_ConvertData(const char Data)

This function converts received telephone data into 16 bit data suitable for use in Windows WAV files, or by a soundcard. This function works correctly regardless of whether the data received from the telephone is in European A-law or American μ -law.

**int MP_GetMaxNumPorts (void)**

This function will return the maximum number of connected devices connected to the PC. Not necessarily the number which currently are connected.

int MP_GetIfSerialNo(int Virtual)

This function will return the serial number of the connected virtual device. If no device is present in this slot it returns 0. Even if the device has a serial number, it might still be faulty. You will need to check the status to ensure that the device is working.

Please note that this information will NOT be available at program startup, due to the time delay necessary to initialise the USB devices.

e.g.

```
// Check all connected devices
for( int I=0 ; I<MP_GetMaxNumPorts() ; I++ )
{
    int Serial = MP_GetIfSerialNo( I );
    if( Serial!=0 )
    {
        // Found a device. Can now use it.
        ...
    }
    // else... No device here
}
```


int MP_GetStatus(int SerialNumber)

This function will return the current status of the requested device, and the status of the connected exchange and telephone. Possible return values from this function are as follows

Value	Meaning
STATUS_GOOD	The Intelligent Recording device, Exchange and Telephone are all functioning correctly
+1 to +100	Intelligent Recording device initialising, percentage complete
STATUS_NOTPRESENT	Unable to open the USB link to the Intelligent Recording device
STATUS_UNLICENSED	Incorrect Intelligent Recording device type. i.e. MP_GetIfType not compatible with MP_GetSwType
STATUS_NOEXCHANGE	Intelligent Recording device ok, Exchange not connected.
STATUS_NOPHONE	Intelligent Recording device ok, Exchange ok, Keypad not connected
STATUS_UNKNOWNERROR	Intelligent Recording device ok. Keypad OR Exchange not connected or not communicating. Unable to determine which is at fault.
STATUS_NOTINITIALISED	Interface not initialised by Open, or has been Close'd.
STATUS_INTERNALERROR	Internal error. Some error has occurred which has stopped the device functioning. May have just been disconnected
STATUS_TOOMANY	Too many devices are connected to this PC. So unable to initialise this one.

NOTE: Unless MP_GetStatus shows the device is connected and initialised, most of the following functions are likely to return useless defaults. Note that the status can change to faulty after being connected ok (if plugs are pulled out), so software should monitor the status regularly, either by polling or via callbacks.



int MP_GetCallState(int SerialNumber)

This function will return the state of the keyset connected to the device. Some systems can return more information on the connected telephone than others. So you should ensure you cope with all the possible return values from this function.

Value	Meaning
CALL_IDLE	The keyset is at rest, and not in use.
CALL_ACTIVE	The keyset is in use. The DLL is unable to determine the type of activity in progress
CALL_INCOMING	The keyset is in use on an incoming call
CALL_OUTGOING	The keyset is in use on an outgoing call
CALL_INCOMING_INTERNAL	The keyset is in use on an incoming call from an internal extension
CALL_INCOMING_EXTERNAL	The keyset is in use on an incoming call from an external line
CALL_OUTGOING_INTERNAL	The keyset is in use on an outgoing call to an internal extension
CALL_OUTGOING_EXTERNAL	The keyset is in use on an outgoing call to an external line
CALL_PAGE	The keyset is receiving a paging message

const char * MP_GetIfType(int SerialNumber)

Get Interface Type. This function will return a pointer to a string describing the currently connected Intelligent Recording device type. i.e. What manufacturer or phone types it is compatible with. Return values exactly the format as MP_GetSwType.

Note that several manufacturers use interfaces which are very similar, and therefore some software will run different interfaces to its own type. Therefore the interface type may not be the same as the software type. Probably best not to display this information to your customers.

To date: Both Avaya versions are compatible. Both Nortel versions are compatible. NEC and Toshiba are compatible.

If you try and run software on a platform it is not compatible with, you will get a reported status of STATUS_UNLICENSED.

**void MP_AudioMonitor(int SerialNumber, int WhatToMonitor)**

Use this function to manually start and stop monitoring of Bearer Channel (audio) data. No B Channel Data or events can be received until recording is started. Once started, audio data will come in to your system at the standard telephone data rate of 8kHz.

You can use the WhatToMonitor parameter to tell the DLL which information you will require. Use AUDIO_MONO if you are only recording merged data, AUDIO_STEREO if you are only recording individual channels, or AUDIO_ON if you might want both. Asking for only partial data can save USB bandwidth.

void MP_AutoRecord(int SerialNumber, int WhatToMonitor)

Auto record will automatically start recording when data is available. As soon as the telephone is in use, the system will automatically start streaming data. When the line again becomes idle the streaming of audio data will stop.

The purpose of providing this facility is to remove the need for continual monitoring when the telephone is idle. This reduces USB usage and PC loading.

The WhatToMonitor parameter is exactly as described for MP_AudioMonitor above.

int MP_GetAudioData(int Port, int MaxLength, unsigned char *pPBXData, unsigned char *pPhoneData, unsigned char *pMergedData)

Use this function to recover monitored Bearer (audio) channel data. This method delivers audio data sent in either/both directions to and from the telephone and/or a merged data channel. The amount of available data for all three channels is always identical. This function returns the amount of data available (the same for each channel). This may be 0, and will be if you have not enabled Bearer Channel monitoring with MP_AudioMonitor. This function erases all returned data from the low level interface buffer.

Any or all of the three pointers may be NULL, in which case MaxLength bytes are deleted from all received channels anyway, in order that the amount of data available from each channel remains identical.

You can check the amount of data available by calling this function with MaxLength=0. This returns the amount of data available from each separate buffer. This does not erase any data from any channel.



e.g.

```
// See how much data is available
int DataLength = MP_GetAudioData( Me, 0, NULL, NULL, NULL );
// Create buffers in the heap to hold the data
char *pPhone = new char[ DataLength ];
char *pExchg = new char[ DataLength ];
// Recover the data. Not interested in merged channel
MP_GetAudioData( Me, DataLength, pPhone, pExchg, NULL );
// Process the recovered data
ProcessAudioPBX( Me, DataLength, pPhone );
ProcessAudioToExchg( Me, DataLength, pExchg );
// Free the heap space
delete [] pPhone;
delete [] pExchg;
```

void MP_SetAudioGain(int SerialNumber, BOOL Remote, int Gain)

This function boosts or reduces the signal from the recorder.

There are two separate gains for each port. Remote=TRUE sets the gain on the speaker i.e. The remote caller. Remote=FALSE sets the gain from the local microphone.

Gain is a fixed point modifier, and will be divided by 256. A value of 256 gives no boost, i.e. exactly what the recorder is seeing on the line. A value >256 will amplify the signal. A value <256 will attenuate the signal. Beware that large gain values can result in the audio becoming heavily corrupted.

Both values default to unmodified. i.e. 256.

void MP_ControlMonitor(int SerialNumber, BOOL OnOff)

Use this function to manually start and stop monitoring of Data Channel (control) data.

int MP_GetControlDataPBX(int SerialNumber, int MaxLength, unsigned char *pPBXData)

Use this function to recover monitored D channel data sent to the telephone from the exchange. The low level interface can buffer approx 8kbytes of data. You provide a buffer and the maximum amount of data you require. This function returns the amount of data available, which may be 0, and will be if you have not enabled monitoring with MP_ControlMonitor. This method erases any returned data from the low level interface buffer.

If pPBXData is NULL, calling this function will erase MaxLength bytes of data without returning them.



You can check the amount of data available by calling this function with MaxLength=0. This does not erase any data from the buffer.

e.g.

```
// See how much data is available and process it all
int DataLength = MP_GetControlDataPBX( 0, NULL );
char *pData = new char[ DataLength ]
MP_GetControlDataPBX( DataLength, pData );
ProcessDataPBX( DataLength, pData );
delete [] pData;
```

int MP_GetControlDataPhone(int SerialNumber, int MaxLength, unsigned char *pPhoneData)

Use this function to recover monitored D channel data sent from the telephone to the exchange. All other details exactly as for GetControlDataPBX above.

LPCSTR MP_GetCallerID(int SerialNumber)

This function returns the caller ID, of the current active telephone call. This may be an empty string if no call is currently in progress. Note that caller id changes do not necessarily occur at the beginning and end of calls. They may be delayed, or on some systems may even precede active call data.

LPCSTR MP_GetCallInfo(int SerialNumber, int Info) *

This function returns information for the current active telephone call. This may be an empty string if no call is currently in progress. The value of Info determines the information returned. Info can be one of:

Value	Meaning
INFO_CALLERID	Return the caller id (as MP_GetCallerID)
INFO_UUI	Return UUI information if supported
INFO_CALLEDPARTY	Return the called party number if supported
INFO_CONNECTEDNO	Return the connected number if supported

These values should be queried after the corresponding event indicates information is available.

* Not all interface DLL's support MP_GetCallInfo. It should not be called directly, but a pointer to it should be obtained using the Windows function GetProcAddress. If this returns NULL then the function is not available.



5. Callbacks

void MyProc(int SerialNumber, int Event, int Data)

This is the method which you provide to allow the low level interface to tell you about received data events. Provided you have sent a pointer to this function to MP_SetCallback, it will be called whenever any significant events occur. You are of course, free to ignore any of the returned events which your application does not require.

Note that the Callback function will be called from an independent child thread, not from the Application's Window thread. It should not therefore be used to draw data directly to a Windows screen. The safest use of the callback is to use a threadsafe interface (such as Windows messaging) to pass the message to Windows for handling by the Windows thread.

```
void MyProc( int SerialNumber, int Event, int Data )
{
    // Send the message to Windows thread for processing
    PostMessage( NULL, WM_USER_MULTIPORTEVENT, Event,
                SerialNumber );
    // Note: Data ignored. We can poll for an up-to-date
    //       value when this message is handled.
}
```

All the functions provided by this library are threadsafe (except MP_Start and MP_Stop) and may be called from the callback's thread, or from the main Windows thread (or indeed from any other thread). So it is possible to recover data, manipulate and even record it from the callback, if desired. However, you should avoid doing any time consuming operations, particularly disk access, within the callback function.

The constants for the Event parameter are defined in MultiXtR.h. Possible values and their meanings are discussed in the following sections.

Note that callbacks have been slowed considerably, compared to the single port version of this API. Therefore the hint information contained in the Data parameter is quite likely to be out of date by the time you process it. It would always be better to use the appropriate poll function to obtain an up-to-date value.

MP_STATUS

The callback function will be called with this event whenever a Intelligent Recording interface device changes status. The Data parameter is the new status, and the possible values and meanings of these values are described in MP_GetStatus.



MP_BCHANNELDATA

The Callback function will be called with this event when B channel data recording is enabled and data is available from this device. The Data parameter gives the number of available bytes.

MP_DCHANNELDATAPBX

The Callback function will be called with this event when a D channel data packet is sent from the exchange to the telephone. The Data parameter will give the number of bytes available. Note that due to packetization of data from the usb device, packets received here will not necessarily correspond to packets of data in the telephone->exchange protocol.

MP_DCHANNELDATAPHONE

Same as above, except it reports data from the phone to the exchange.

MP_BCHANNELOVERFLOW

The B channel buffers have overflowed due to data not being read quickly enough from the buffer. Data recording will continue, but will overwrite old data, leading to missing audio and a discontinuity.

MP_DCHANNELOVERFLOWPBX

The buffer of D channel data passing to the phone has overflowed. Some data has been lost.

MP_DCHANNELOVERFLOWPHONE

The buffer of D channel data passing from the phone has overflowed. Some data has been lost.

MP_CALLSTATE

This event is sent whenever the call state changes on an interface. Note that you may receive multiple call state messages for a single call, as the interface determines what the user is doing with his telephone. Possible Data values are any of the return values from MP_GetCallState.

MP_CALLERIDUPDATE

This event is sent whenever the caller id string returned by MP_GetCallerID (or MP_GetCallInfo(... , INFO_CALLERID)) changes. Note that caller id changes do not necessarily occur at the beginning and end of calls. They may be delayed, or on some systems may even precede active call data.



MP_UUIUPDATE

This event is sent whenever the User to User Information string returned by MP_GetCallInfo(..., INFO_UUI) changes.

MP_CALLEDPARTYIDUPDATE

This event is sent whenever the called party number string returned by MP_GetCallInfo(..., INFO_CALLEDPARTY) changes. Note that called party changes do not necessarily occur at the beginning and end of calls. They may be delayed, or on some systems may even precede active call data.

MP_CONNECTEDNOUPDATE

This event is sent whenever the connected number string returned by MP_GetCallInfo(..., INFO_CONNECTEDNO) changes. Note that connected number changes do not necessarily occur at the beginning and end of calls. They may be delayed, or on some systems may even precede active call data.



6. Changes from the single port API

We have added a serial number to all functions which affect telephones. This is the same serial number which is present in the interface device. For single port devices, the port serial number is the same as the device number, and should be written on the label. Because these are hardwired, the same as the telephone connections, your software can map Intelligent Recording serial numbers to fixed keyset extension numbers or usernames.

In future, we expect there to be an interface device with up to 60 telephone channels. This API has been designed to support those devices too. So you will be able to use the new hardware with no changes to your software. Multiport devices will report each telephone channel with a different serial number.

Because there are now many devices to control, initialisation can be much slower.

We have greatly slowed the callback generation from the single port version. Under some conditions, the single port device could flood a Windows98 message queue on its own (Later Windows versions seem much better). Clearly, with multiple devices, this problem would be more significant. The DLL now has much larger internal buffers. It can hold approx 10 seconds worth of data. You should therefore expect that both audio and Control data will be presented in much larger chunks than before. How much depends on the speed of your system, and the number of connected devices, but typically 4k at a time instead of 128 bytes. You are of course free to handle the data anyhow you like, but it may help to optimise your code if it copes with larger chunks all in one go.

D-channel (control) information from the keyset is now turned OFF by default. Very few people chose to decode this information. You can turn it back on using `MP_ControlMonitor` if you need it.