E1UC Switch/Groomer

API REFERENCE GUIDE

Applicable Products

E1UC-API-3103	E1UC Interface API
Part Number:	E1UC-API-3103

Document Reference

E1UC-SAPI-0001

Publication Date

28 April 2016

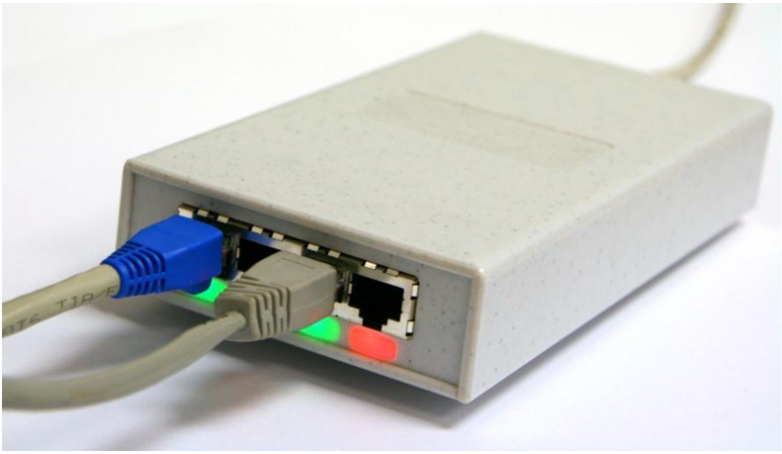
Published by

Somerdata Ltd.
1 Riverside Business Park
Bristol
BS4 4ED
UK

Sales & Customer Support

Phone:	+44 (0)117-9634050
E-Mail:	*sales@somerdata.com* *support@somerdata.com*
Website:	*www.somerdata.com*

E1UC ***Switch/Groomer***



API reference Guide

REVISION HISTORY

Issue	Date	Notes
1	28 Oct. 11	Initial Issue. Relates to DLL version 3103.0.7.0.
2	1 Nov. 11	Added structures. Relates to version DLL 3101.0.8.0
3	4 Nov. 11	Added Recording to file information.
4	Sep 2014	Added Functions
5	April 2016	Changed data presentation

CONTENTS

1. INTRODUCTION

In this Section

What's in this User Guide

This User Guide covers SomerData's E1UC-API-3103 E1UC interface API.

Section 2 – *PRODUCT DESCRIPTION* gives an overview of your unit's capabilities and features.

Section 3 – API function listing

Section 4 – *SUPPORT* describes the procedure and contact details for obtaining customer support on this product.

User Guide Availability

Printed copies of Hardware and Software User Guides are supplied with the original products on request.

Additional printed copies, including the Programmer's Reference Guide can be supplied on request. Please contact your local supplier or SomerData for ordering details.

Electronic copies (Adobe Acrobat files) are included on the SomerData CD-ROM that is supplied with the original products.

The electronic User Guide library, which also includes product data sheets, can be accessed by browsing the *\Documents* folder for the required document.

Additional and updated copies of the CD-ROM can be supplied on request. Please contact your local supplier or SomerData for ordering details.

2. DESCRIPTION

E1UC is a versatile USB controlled Switch and Groomer for E1/G.703 signals.
An E1UC board has four E1 IO ports and 1 dedicated Grooming output.
Each Input stream can be routed to any output stream.
When a G.704 framed signal is detected, each timeslot from each input can be routed to the groomed output.
An E1UC system can be made of 1 E1UC board in a self contained portable box or up to 4 boards connected to each other in a 1U Rack mountable case.
The API provides all the functions required to control and retrieve status from E1UC.
This Application Programming Interface defines functions, procedures, constants and entry points that can be used in a Microsoft Windows programming environment.
The API is dependent on the .NET framework and this must be installed before use.
The API is targeted towards the C# programming language, but a Dynamic Link Library may be used in other Windows environments
The API is presented as a series of functions comprising passed variables and returned error codes and/or exceptions.
Since the E1UC uses the FTDI USB communications chipset, many of the functions have corollary functions in the FTDI API and structures are similar.
Some indication is given of programme development in this guide, contact Somerdata support for help and advice if developing your own application.
Development of the API is continuous so this document may not always reflect the latest available issue.
The E1UC suite of applications uses this API, so it is continuously being tested and verified, however if you find errors please let us know, support@somerdata.com

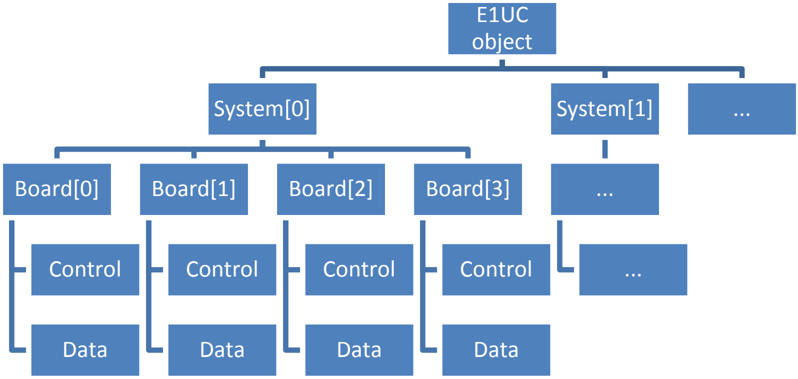
3. API function listing

This section details the API functions as a reference list, in an order which reflects the way a device may be accessed and used.
E1UC has two endpoint interfaces, one for control and the other for data. Mediation within the device and sophisticated buffering ensures that data flow is not interrupted due to status requests, and that status is not unreasonably held up due to long data transfers.
Return values are enumerated as constants, defined at the end of the reference section.

createListE1UCDevice

Summary:

This function builds an E1UC object containing the list of all E1UC systems connected. Each system is composed of a unique serial number and a list of all boards accessible in the system. Finally each board has two structures, <DEV_INFO>, used to open a device. One is used to control the board, the other is to capture data.



Definition:

UInt32 createListE1UCDevice()

Parameters:

None

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_IO ERROR_DEVICE_NOT_FOUND ERROR_INSUFFICIENT_RESSOURCES ERROR_NUMBER_BYTE_READ ERROR_NUMBER_BYTE_WRITTE ERROR_UNKNOWN

openE1UCDevice

Summary:

Opens the device and return a Handle which will be used for subsequent accesses. Use for single board access.

Definition:

IntPtr openE1UCDevice(DEV_INFO DeviceInfo, ref UInt32 CodeError)

Parameters:

DeviceInfo	Structure which contains all information needed to open a device
CodeError	Represents the error code return by the function

Return Value:

Success :	a handle to the E1UC device requested + ERROR_SUCCESS
Failure :	NULL if no device found + Specific error code which could be : ERROR_DEVICE_NOT_FOUND ERROR_DEVICE_NOT_OPENED ERROR_UNKNOWN

openE1UCSystem_ControlPorts

Summary:

Opens all the connected devices in a system and returns an array of handles which will be used for subsequent accesses.

Definition:

IntPtr[] openE1UCSystem_ControlPorts(UInt32 IndexSystem, ref UInt32 CodeError)

Parameters:

IndexSystem	The number of the E1UC system requested
CodeError	Represents the error code return by the function

Return Value:

Success :	a handle to the E1UC device requested + ERROR_SUCCESS
Failure :	NULL if no device found + Specific error code which could be : ERROR_DEVICE_NOT_FOUND ERROR_DEVICE_NOT_OPENED ERROR_UNKNOWN

closeE1UCDevice

Summary:

Close an opened device.

Definition:

UInt32 **closeE1UCDevice**(IntPtr Handle)

Parameters:

Handle	Handle of the device

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_IO ERROR_INVALID_HANDLE ERROR_UNKNOWN

getE1UCDeviceFirmware

Summary:

This function provides information about the device firmware.

Definition:

UInt32 **getE1UCDeviceFirmware**(IntPtr Handle, ref DEV_FW DeviceFirmware)

Parameters:

Handle	Handle of the device
DeviceFirmware	Pointer to a DEV_FW structure to receive device firmware information

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

getE1UCDevConfiguration

Summary:

This function provides information about the device configuration.

Definition:

UInt32 **getE1UCDeviceConfiguration**(IntPtr Handle, ref DEV_CONFIGURATION DeviceConfiguration)

Parameters:

Handle	Handle of the device
DeviceConfiguration	Pointer to a DEV_CONFIGURATION structure to receive device configuration information

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

setE1UCDeviceConfiguration

Summary:

This function sets the device configuration.

Definition:

UInt32 setE1UCDeviceConfiguration(IntPtr Handle,
DEV_CONFIGURATION DeviceConfiguration)

Parameters:

Handle	Handle of the device
DeviceConfiguration	Pointer to a DEV_CONFIGURATION structure to provide device configuration information

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

getE1UCXFifo

Summary:

This function provides information about the device’s FIFO buffer. (See later section)

Definition:

UInt32 **getE1UCXFifo**(IntPtr Handle, ref USB_XFIFO XilinxFifo)

Parameters:

Handle	Handle of the device
XilinxFifo	Pointer to a USB_XFIFO structure to receive FPGA FIFO status information

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

setE1UCXFifo

Summary:

This function sets the device’s FIFO controls.

Definition:

UInt32 setE1UCXFifo(IntPtr Handle, USB_XFIFO XilinxFifo)

Parameters:

Handle	Handle of the device
XilinxFifo	Pointer to a USB_XFIFO structure to provide FPGA FIFO control information

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

getE1UCNumberStreamAvailable

Summary:

This function provides the number of streams available for the entire system. If the system is a Standalone it will be 4, otherwise it will be up to 16.

Definition:

UInt32 **getE1UCNumberStreamAvailable** (IntPtr Handle, ref UInt32 NumberStreamAvailable)

Parameters:

Handle	Handle of the device
NumberStreamAvailable	Pointer to a UInt32 that will contain the number of stream available

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

getE1UCStreamStatus

Summary:

This function provides status about the stream selected.

Definition:

UInt32 getE1UCStreamStatus(IntPtr[] Handle,UInt32 Stream, ref STREAM_STATUS StreamStatus)

Parameters:

Handle	Array which contains all handles of the system concerned. This array could be provided by the function **openE1UCSystem**
Stream	The canonical number of the stream requested
StreamStatus	Pointer to a STREAM_STATUS structure to receive Stream Status information

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

setE1UCStreamControl

Summary:

This function sets the controls for the selected stream.

Definition:

UInt32 setE1UCStreamControl(IntPtr[] Handle, UInt32 Stream, ref STREAM_STATUS StreamStatus)

Parameters:

Handle	Array which contains all handles of the system concerned. This array could be provided by the function **openE1UCSystem**
Stream	The number of the stream requested
StreamStatus	This parameters provides Stream Status information to the system

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_WRITE_REGISTER ERROR_UNKNOWN

getE1UCSigStatus

Summary:

This function provides the signal status about the stream selected.

Definition:

UInt32 getE1UCSignalStatus(IntPtr[] Handle, UInt32 Stream, ref SIG_STATUS SignalStatus)

Parameters:

Handle	Array which contains all handles of the system concerned. This array could be provided by the function **openE1UCSystem**
Stream	The number of the stream requested
SignalStatus	Pointer to a SIG_STATUS structure to receive Signal Status information

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

getE1UCMatrixStatus

Summary:

This function provides the current switch matrix settings. The variable StreamDestination of the MATRIX_BUF structure has to be initialised before calling this function.

Definition:

UInt32 getE1UCMatrixStatus(IntPtr[] Handle, ref MATRIX_BUF MatrixBuf)

Parameters:

Handle	Array which contains all handles of the system concerned. This array could be provided by the function **openE1UCSystem**
MatrixBuf	Pointer to a MATRIX_BUF structure to receive the current switch matrix settings

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_NOT_ENOUGH_BOARD ERROR_INDEX_STREAM_OUT_OF_RANGE ERROR_UNKNOWN

setE1UCMatrixControl

Summary:

This function sets the switch matrix settings, the variable StreamDestination of the MATRIX_BUF structure has to be initialised before calling this function.

Definition:

UInt32 setE1UCMatrixControl(IntPtr[] Handle, MATRIX_BUF MatrixBuf)

Parameters:

Handle	Array which contains all handles of the system concerned. This array could be provided by the function **openE1UCSystem**
MatrixBuf	Pointer to a MATRIX_BUF structure to provide the current switch matrix settings

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_WRITE_REGISTER ERROR_NOT_ENOUGH_BOARD ERROR_INDEX_STREAM_OUT_OF_RANGE ERROR_UNKNOWN

getE1UCGroomStatus

Summary:

This function provides the current groomed output settings. Variables E1Number and TimeslotDestinationE1 of the GROOM_BUF structure have to be initialised before calling this function.

Definition:

UInt32 getE1UCGroomStatus(IntPtr[] Handle, ref GROOM_BUF GroomBuf)

Parameters:

Handle	Array which contains all handles of the system concerned. This array could be provided by the function **openE1UCSystem**
GroomBuf	Pointer to a GROOM_BUF structure to receive the current groomed output settings

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_NOT_ENOUGH_BOARD ERROR_INDEX_STREAM_OUT_OF_RANGE ERROR_UNKNOWN

setE1UCGroomControl

Summary:

This function sets the groomed output settings. Variables E1Number and TimeslotDestinationE1 of the GROOM_BUF structure have to be initialised before calling this function.

Definition:

UInt32 setE1UCGroomControl(IntPtr[] Handle, GROOM_BUF GroomBuf)

Parameters:

Handle

Array which contains all handles of the system concerned. This array could be provided by the function **openE1UCSystem**

GroomBuf

Pointer to a GROOM_BUF structure to provide the current groomed output settings

Return Value:

Success : ERROR_SUCCESS

Failure : ERROR_INVALID_HANDLE
ERROR_WRITE_REGISTER
ERROR_NOT_ENOUGH_BOARD
ERROR_INDEX_STREAM_OUT_OF_RANGE
ERROR_UNKNOWN

getE1UCSystemName (not implemented)

Summary:

This function provides the non-volatile system name field.

Definition:

UInt32 getE1UCSystemName(IntPtr Handle, ref string strName)

Parameters:

Handle	Handle of the device
strName	Pointer to a string variable to receive the current system name

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

setE1UCSystemName (not implemented)

Summary:

This function sets the non-volatile system name.

Definition:

UInt32 setE1UCSystemName(IntPtr Handle, string strName)

Parameters:

Handle	Handle of the device
strName	String variable which contains the system name

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_WRITE_REGISTER ERROR_UNKNOWN

getE1UCTestFeature

Summary:

This function gets the status of the Active LED and test LED. This is useful for “Hello World” type functions.

Definition:

UInt32 getE1UCTestFeature(IntPtr Handle, ref TEST_SYSTEM TestSystem)

Parameters:

Handle	Handle of the device
TestSystem	Pointer to a TEST_SYSTEM structure to receive the current state of the Test LED and the Heartbeat

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_UNKNOWN

setE1UCTestFeature

Summary:

This function sets the test LED.

Definition:

UInt32 setE1UCTestFeature(IntPtr Handle, TEST_SYSTEM TestSystem)

Parameters:

Handle	Handle of the device
TestSystem	Pointer to a TEST_SYSTEM structure to provide the current state of the Test LED. Heartbeat is a read only bit

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_WRITE_REGISTER ERROR_UNKNOWN

resetS2CFifo

Summary:

This function does a reset of the FIFO from the host system (S) to the E1UC card(C), clearing all data.

Definition:

```
UInt32 resetHostToE1UCSystemFifo(IntPtr HandleControl,
IntPtr HandleData)
```

Parameters:

HandleControl	Handle of the control device concerned
HandleData	Handle of the data device concerned

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_WRITE_REGISTER ERROR_UNKNOWN

resetC2SFifo

Summary:

This function does a reset of the FIFO from the Card(C) to the Host system (S), clearing all waiting data.

Definition:

```
UInt32 resetE1UCSystemToHostFifo(IntPtr HandleControl,
IntPtr HandleData)
```

Parameters:

HandleControl	Handle of the control device concerned
HandleData	Handle of the data device concerned

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_WRITE_REGISTER ERROR_UNKNOWN

InitDataTransfer

Summary:

This function initializes the system to be ready for a read or a write data operation.

Definition:

UInt32 InitDataTransfer(IntPtr HandleControl, IntPtr HandleUSB, bool PCToBoard, byte Mask_Streams)

Parameters:

HandleControl	Handle of the control device concerned
HandleData	Handle of the data device concerned
PCToBoard	Bool variable which controls the direction for the data transfer
Mask_Streams	Byte variable which contains the streams to be read or write

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_WRITE_REGISTER ERROR_UNKNOWN

EndDataTransfer

Summary:

This function has to be called to finish a read or write operation properly.

Definition:

UInt32 EndDataTransfer(IntPtr HandleControl, IntPtr HandleUSB)

Parameters:

HandleControl	Handle of the control device concerned
HandleUSB	Handle of the data device concerned

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_WRITE_REGISTER ERROR_UNKNOWN

readStreamBuffer

Summary:

This function saves up to 65280 bytes from the 4 streams (or less: depend on configuration of **initDataTransfer**) into different files.

Definition:

UInt32 readStreamBuffer(IntPtr Handle, FileStream[] myFiles)
UInt32 readStreamBuffer(IntPtr Handle, FileStream myFile)

Parameters:

Handle	Handle of the device
myFile	Single file in which data from each stream for this card is saved: data from each stream is interleaved, 1 frame per stream at a time. (See section 0 Data presentation for the definition of a frame)
myFiles	Array of files in which data are saved: Stream 0,1 ,2, 3 into myFiles[0] ...

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_WRITE_REGISTER ERROR_UNKNOWN ERROR_INVALID_DATA_LENGTH

writeStreamBuffer

Summary:

This function sends an array of bytes (buffer) to the E1UC USB data port. This data buffer must have data from all streams that have been enabled in **initDataTransfer** E1 frame interleaved.

Each E1 frame is preceded with an 8 byte header (See section 0 Data presentation for the definition of a frame), therefore the buffer must be modulo 40 bytes * number streams enabled in length. It must also be less than 64Kbytes in length.

For example: a 65280 byte buffer containing data from 4 streams will be made up from 408 E1 frames from each stream:

Frame number	Stream number	Number of bytes
1	1	40
2	2	40
3	3	40
4	4	40
5	1	40
...	...	...
1632	4	40

Definition:

```
UInt32 WriteStreamBuffer(IntPtr HandleData, byte[]
buffer, uint bufferSize)
```

Parameters:

HandleData	Handle of the data port of the device
`buffer`	Array of bytes to be sent. See summary for restrictions on the format of this data array.
`bufferSize`	Size of buffer

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_WRITE_REGISTER ERROR_UNKNOWN ERROR_INVALID_DATA_LENGTH ERROR_REPLAY_TIMEOUT

sendFileToCard

Summary:

A higher-level function for sending a file of data to E1UC data port. Includes initialisation.

Definition:

```
UInt32 sendFileToCard(IntPtr HandleControl, IntPtr
HandleData, string FileLoc)
```

Parameters:

HandleControl	Handle of the control device concerned
HandleData	Handle of the data port of the device
FileLoc	String representing the location on the system of the file to be send

Return Value:

Success :	ERROR_SUCCESS
Failure :	ERROR_INVALID_HANDLE ERROR_READ_REGISTER ERROR_WRITE_REGISTER ERROR_FILE_NOT_FOUND ERROR_UNKNOWN

Structures

STRUCT DEV_INFO

```
string DeviceName
string Channel
string SerialNumber
UInt32 LocationID
UInt32 DeviceIndex
```

STRUCT DEV_FW

```
UInt32 PartNum
UInt16 Version
UInt16 Revision
UInt64 SerNum
```

STRUCT DEV_CONFIGURATION

```
bool NvisBusy
bool isStandAlone
bool InputImedance120
bool InputImedanceHigh
bool isMaster
byte RxFifoControl
byte TxFifoControl
```

STRUCT USB_XFIFO

```
bool C2SFifoFull
bool C2SFifoOverRun
bool C2SFifoReset
bool S2CFifoEmpty
bool S2CFifoUnderRun
bool S2CFifoReset
bool FifoS2CDir
```

STRUCT STREAM_STATUS

```
UInt16 E1Stream;
bool isStreamActive
bool isExternalClock
bool isBypass
bool DataToUSB
byte FramingSelected
```

STRUCT SIG_STATUS

```
bool isLOS
bool isLOF
bool isA1S
bool isFIFO_ERROR
```

STRUCT MATRIX_BUF

```
byte StreamSourceE1
byte StreamDestinationE1
bool OutputDestinationEmpty
```

STRUCT GROOM_BUF

```
byte TimeslotSource
byte SourceE1
byte TimeslotDestination
bool TimeslotEmpty
byte EmptyDefaultbyte
```

STRUCT TEST_SYSTEM

```
bool TestLed
bool Heartbeat
```

Error Codes

ERROR_SUCCESS	0
ERROR_INVALID_HANDLE	1
ERROR_DEVICE_NOT_FOUND	2
ERROR_DEVICE_NOT_OPENED	3
ERROR_IN_OUT	4
ERROR_INSUFFICIENT_RESOURCES	5
ERROR_READ_REGISTER	6
ERROR_WRITE_REGISTER	7
ERROR_NOT_ENOUGH_BOARD	8
ERROR_INDEX_STREAM_OUT_OF_RANGE	9
ERROR_FILE_NOT_FOUND	10
ERROR_REPLAY_TIMEOUT	11
ERROR_UNKNOWN	100

4. Recording input streams to File

Operation

The process for reading a file is as follows:

Get a data handle
Using the openE1UCDevice function eg:
HandleData =
myAPI.openE1UCDevice(myAPI.E1UC[0].CARD[0].Data, ref CodeError);

Initialise data transfer
This will reset internal buffers and prepare E1UC to transfer data over the USB port. Eg:
myAPI.InitDataTransfer(SingleHandle, HandleData, false,
E1UC_API.DEFINES.MASK_STREAM_0 |
E1UC_API.DEFINES.MASK_STREAM_1 |
E1UC_API.DEFINES.MASK_STREAM_2 |
E1UC_API.DEFINES.MASK_STREAM_3);

Read data to file
Reading data to file should be done on a separate thread. (See code snippet below). This enables data to be transferred while still being able to access status of the device to manage the transfer. Note that the system uses a first-in-first-out memory buffer. This should be cleared (reset) between mode switches, (read to write, write to read) to ensure that 'stale' data is not present.
To ensure that all data is transferred, it is necessary to flush the FIFO by reading the exact number of dwords it contains until empty, as indicated by the API call **getE1UCXFifo.**

End data transfer
When data has finished being transferred to file, some cleanup is required. This is done with EndDataTransfer.
The USB data port should be closed with `closeE1UCDevice`

Code snippet example (C#)

```
private void bReadStream_Click(object sender, EventArgs e)
{
    UInt32 CodeError = 9999;
    bReadStream.Enabled = false;
    Reading = new Thread(new ThreadStart(funct_Reading));

    HandleData =
myAPI.openE1UCDevice(myAPI.E1UC[0].CARD[0].Data, ref
CodeError);

    if (HandleControl != null)
        SingleHandle = HandleControl[0];

    myAPI.InitDataTransfer(SingleHandle, HandleData, false,
E1UC_API.DEFINES.MASK_STREAM_0 |
E1UC_API.DEFINES.MASK_STREAM_1 |
E1UC_API.DEFINES.MASK_STREAM_2 |
E1UC_API.DEFINES.MASK_STREAM_3);

    bReading = true;
    Reading.Start();

}

private void funct_Reading()
{
    string strPathFile = tbFolder0.Text;

    using (FileStream myFile = File.Open(strPathFile +
"\\Streams.bin", FileMode.Create, FileAccess.Write,
FileShare.None))
        {
            while (bReading)
            {
                myAPI.readStreamBuffer(HandleData, myFile);
                iCount++;
            }
            myFile.Close();
```

```
     }
}
private void bStopReading_Click(object sender, EventArgs e)
{
   UInt32 CodeError = 9999;
   bReading = false;
   Reading.Join();
   myAPI.EndDataTransfer(SingleHandle, HandleData);
   //close handle data
   CodeError = myAPI.closeE1UCDataDevice(HandleData);
   bReadStream.Enabled = true;

   MessageBox.Show("Thread for reading finished")
}
```

Data presentation

Data is presented as a concatenated serial stream of blocked data, each block representing 1 E1 Stream.
Block structure is as follows:

<table>
<tr><td></td><td>Byte 3</td><td colspan="2">Byte 2</td><td>Byte 1</td><td colspan="2">Byte 0</td></tr>
<tr><td>DWORD</td><td colspan="6">bit 31 bit 0</td></tr>
<tr><td>0</td><td colspan="6">Time stamp (seconds)</td></tr>
<tr><td>1</td><td colspan="2">Timestamp(milli seconds)</td><td>Fraction of ms</td><td>Frame Status</td><td>reserved</td><td>E1 ID</td></tr>
<tr><td>2</td><td>TS 0</td><td colspan="2">TS1</td><td>TS2</td><td colspan="2">TS3</td></tr>
<tr><td>3</td><td>TS4</td><td colspan="2">TS5</td><td>TS6</td><td colspan="2">TS7</td></tr>
<tr><td>4</td><td>TS8</td><td colspan="2">TS9</td><td>TS10</td><td colspan="2">TS11</td></tr>
<tr><td>5</td><td>TS12</td><td colspan="2">TS13</td><td>TS14</td><td colspan="2">TS15</td></tr>
<tr><td>6</td><td>TS16</td><td colspan="2">TS17</td><td>TS18</td><td colspan="2">TS19</td></tr>
<tr><td>7</td><td>TS20</td><td colspan="2">TS21</td><td>TS22</td><td colspan="2">TS23</td></tr>
<tr><td>8</td><td>TS24</td><td colspan="2">TS25</td><td>TS26</td><td colspan="2">TS27</td></tr>
<tr><td>9</td><td>TS28</td><td colspan="2">TS29</td><td>TS30</td><td colspan="2">TS31</td></tr>
</table>

DWORD 0 and DWORD 1 Bytes 2 &3

Timestamp format: 32 bit second count since power up of device + 10 bit millisecond count + 6 bit fraction of millisecond (64 fractions).

DWORD 1 Byte 1

Frame Status: Bit 15: FRAMED –should the system be looking for framed data

bit 14: Signal present

Bit 13: No data in this frame (pad frame)

Bit 12: One or more frames dropped since the last good frame. (dropped frame)

Bits11-8 FRAME_DETECT Framing detected associated with this block.**Error! Reference source not found.** Currently only G.704 frame detect is defined,

Bit 11	Bit 10	Bit 9	Bit 8	
0	0	0	0	No Frame detected
0	0	0	1	G.704 Frame detected

N.B. If Bit 15 is 0, no detection is performed.

DWORD 1 Byte 0

Bits 7-5: reserved

Bit 4-0 E1_ID. The number (0-15) of the E1 stream associated with this block.

Frame status is dynamic, dependant on the detected state of received signals.

File Header

200 bytes are reserved at the beginning of the file for housekeeping information. This may be used for auditing and setup purposes for file replay and reading applications.

All files written using the API automatically include this header.

The file header is structured as follows:

5 Fields of 40 bytes each, 1 field containing overall file information, 1 field for each stream. Unused bytes are filled with 0x20 (ascii space).

Field 1 Device and File information

Byte 0 – Length of following Version information

Byte1- Byte(Version Length)

Type 4 Bytes (always ascii 3103)

1 byte .(period)

Major version number 1 Byte

1 byte . (period)

Minor version number 1 Byte

1 byte . (period)

Build Number.

Byte (2+Version Length) – Streams - bit position representation of the number of active streams

Bit3	Bit2	Bit1	Bit0
Stream4	Stream3	Stream 2	Stream 1

Byte 3 + VersionLength Filename length of following bytes

(Byte 4+Version Length) to (Byte 4+ Version Length) + FileNameLength – string representing a short version of the original filename, a maximum of 24 bytes.

This is used to hold a shortened version of the original file name as recorded, and can be used in subsequent copy and dub operations to provide a simple audit trail. The length byte indicates the length of the following (ascii) string.

Field 2 -5 Stream Information

Each field contains information about one stream.

Byte 0-1 short type Stream number, 0-3

Byte 2 Stream Status – bit field containing stream information at the time of recording

Bit 0 - Active=1 : the stream was enabled

Bit 1 - External Clock =1

Bit 2 – Bypass = 1 – bypass mode selected during recording

Bit 3 – DataToUSB=1

Note these bits are defined for troubleshooting and audit purposes

Bit 4 – Corrupt =1. The file was not properly closed – header information may be incomplete or incorrect.

Bit – 5 Copy = 1. This file is a copy or dub of an original

Byte 3 Framing – Framing is selected for this stream. Only G.704 framing is supported currently. 0= No Framing 1= G704 Framing.

Bytes 4-11 Start time of recording – Windows 64-bit FILETIME structure.(UTC)

Bytes 12-19 Start Byte - 64-bit integer specifying index of first data byte written.

Bytes 20-27 Stop time of recording – Windows 64-bit FILETIME (UTC)

Bytes 28-35 Stop Byte - 64-bit integer specifying index of last data byte written.

Start and stop times are UTC and can be used to maintain audit information during copy/dub processes.

Applications can also use this time information to provide absolute time indices into the data file by synchronising with the relative data frame timestamp.

Synchronisation

E1UC is a synchronous system. Stream 1 is the master stream and system timing is derived from this stream. If there is no signal in this input, the next input with a signal present will be used as master.

To cope with situations where 2 signals (or more) with unrelated clocks are used in the same system, an entire E1 frame (256 bits) will either be dropped or a padding frame inserted to keep signals aligned.

When a frame is dropped, the subsequent frame received will have a flag indicating that the previous frame went missing. If a padding frame is inserted, the pad frame flag will be set.
The master stream will never have a frame go missing or get padded (not withstanding USB buffer over/under-runs).
Note, bit 12 does not indicate that data went missing because of a USB buffer overrun.

Buffering

Data is buffered to allow application latency variations to be accommodated. However, when the host application is too slow to either accept data in capture mode or present data in replay mode, the USB buffer will overflow or become empty. The application designer must ensure that this situation is accommodated, using the buffer over and underrun flags as appropriate.

Frame continuity in captured data can be checked by reference to the incrementing frame timestamp and monitoring of the pad/missing frame flags.

This page intentionally blank

5. SUPPORT

In this Section

What to do if you have a problem

Firstly, please ensure that you have followed the installation, connection and operation instructions in the appropriate User Guide.

Also, check the Troubleshooting section (where appropriate) to eliminate common problems.

Servicing, Maintenance and Repairs

Please contact your supplier or SomerData for all questions relating to maintenance and repairs.

Any unauthorised attempt to open, modify or otherwise repair the product will invalidate the SomerData warranty and may result in the product being left in an irreparable condition.

If you need Support

For warranty, technical and application support issues, you should initially contact your supplier to check whether your SomerData product is covered by warranty, extended warranty or maintenance contract.

At SomerData, we will make our best efforts to provide prompt and friendly support by phone, fax and e-mail.

Diagnosing a problem will require your co-operation and we expect you to provide a detailed description of the problem in the form of a detailed Fault Report.

Support Requests

When contacting SomerData for support, please provide as much information as possible about the problem or issue for which you require assistance.

We will be able to deal with your request more efficiently if you provide the following details (where available) in your Fault Report:

Part Number or Model Number (for example E1UC-4)

Serial Number (for example 2016/01/001)

Software Version (for example 2.0)

Details of any symptoms or error messages

Diagnostics information (if available)

Sequence of events/actions or other circumstances that triggered the problem

How you are able to identify that there is a problem

How you have been able to measure, log or otherwise display the problem

Details of the host PC (if appropriate) including: operating system; hardware configuration; other hardware devices (e.g. additional PCI cards); other software applications (e.g. analysis or processing programs) that are running at the same time

Sample data files (if appropriate)

When we acknowledge your support request, you will be given a *Support Tracking Number* (STN), which should be quoted in all further correspondence relating to that specific support request.

SomerData Contact Information

Address: Somerdata Limited
1 Riverside Business Park,
St Annes Road
Bristol
BS4 4ED
UK

Phone: UK 0117-9634050
International +44 117-9634050

E-Mail: *support@somerdata.com*

Website: www.somerdata.com