Flexilink

Control programs for Flexilink networks

Introduction

Although a Flexilink network is essentially “plug and play”, and some terminal devices will have a user interface through which flows can be set up and cleared down, there is still a need for network control applications. Functionality includes: monitoring, controlling, and trouble-shooting routing through the core; updating software in network elements; and emulating crosspoint audio and video routers.

The development environment includes a Controller program that gives low-level access to Aubergine units; it’s intended mainly for debugging their software and to demonstrate the network’s capabilities. Here we document the internal structure of that program and indicate which parts of the code would be useful for more user-friendly control applications.

The description here assumes some familiarity with the functionality the Controller provides, and with ETSI GS NIN 005.

The program was developed as a Microsoft Foundation Class (MFC) application for Windows 10 in Visual Studio 2019. Much of the code is in platform-independent C++, but Microsoft-specific code is used for platform-specific functionality such as file access and rendering images on the screen.

Program outline

The broad functionality is as follows; more detail is given in later sections.

The main program (CControllerApp class) imports the command line options and creates the three “documents” and their associated “view” windows. The first two use class CCrosspointDoc; OnNewDocument() sets the first up to collect media sources and the other to collect destinations.

The third uses class CControllerDoc, within which OnNewDocument() creates a LinkSocket object which attempts to connect a link to the Flexilink network.

When the link is connected, a MgtSocket object is created for the unit at the other end of the link (the “link partner”). It makes a management connection to the unit and collects information including the identities of other units to which it has links. MgtSocket objects are created for them and the process is repeated until all units on the network have been found.

Each MgtSocket object requests its target unit to send “status broadcast” messages to report various pieces of information within its Management Information Base. These are sent every 20 seconds, or immediately to report changes. The MgtSocket object keeps a record of the

most recently-received value of each MIB object. Information collected from all the units is reported in the Crosspoint windows.

The Controller window displays information from one unit, selected by the user. There are three contexts, selected by CControllerDoc::display_select. The default displays the information from the unit's MIB, and a dump of the packets sent and received. The "console" display shows text received from the unit, mostly in response to single-keystroke requests; in earlier versions this was carried on a separate serial interface. The "analyser" display provides access to an Analyser object (see below).

Link to Flexilink network

The link over which all communication with Flexilink network elements takes place is implemented by the LinkSocket class, which is derived from CAsyncSocket. (This means communication with the network is done via Winsock; I couldn't find a way to get the Windows implementation of Berkeley Sockets to do non-blocking recv() calls.) See B.1.3 and B.3 in GS NIN 005 for the encapsulation and the process to establish a link.

For each flow that is connected (other than the signalling flow) there is an object of a class derived from FlexilinkSocket. The base class handles sending and receiving messages and maintaining the connection while the derived class handles the higher-layer protocols and may also interact with the user.

To initiate the link, LinkSocket::Init() sends a Link Request message to the address specified by "-server" on the command line, or broadcasts it if no address was specified.

LinkSocket::OnReceive() is called for each incoming packet. It processes link-layer messages and some signalling messages; all others are passed on to the relevant FlexilinkSocket object.

Management connections

For each network element (or "unit") there is a MgtSocket object; MgtSocket is a class derived from FlexilinkSocket which implements a protocol similar to SNMP. The MIB is based on the MIB specified in IEC 62379; the exact set of MIB objects can be found in <InitMib> in the TeaLeaves code. More information about an object can be found in the routine that is <leaf.f> at the time the object is added to the MIB.

For future devices we might use a different protocol which might be implemented as a new class derived from FlexilinkSocket. For audio devices it could be based on AES70. A unit might support one MIB for network functionality and a different one (on a different connection, and maybe using a different protocol) for controlling audio or video functions.

The messages begin with a 2-octet header formatted as follows:

- 1 bit: 0 = request, 1 = response
- 3 bits: message type: 0 = Get, 1 = GetNext, 2 = Status (see "Status broadcasts" below), 3 = Set, 4 = non-volatile Set, 5 and 6 reserved, 7 = console data (see below)
- 4 bits: for GetNext request: object count (see below); for other requests: reserved (code as 0); for response: status (see below)
- 1 octet: serial number (chosen by sender of request except for status broadcasts)

Except where otherwise noted below, the remainder of the message is coded according to ASN.1 Basic Encoding Rules (as in SNMPv1), and consists of:

optional OCTET STRING (see “passwords” below)
OID
value (absent if not required, see below)
further {OID, value} pairs as above if reply to GetNext

Status values in responses are as follows; codes 0-5 are the same as in SNMP:

0	normal reply (no error)
1	message has been truncated
2	no object corresponding to the requested OID
3	value is of the wrong type for the object, or otherwise illegal
4	object is read-only
5	other (unspecified) error, including message type not recognised
6	non-volatile Set not supported for this object
7	resource not available, e.g. can't create a new record because the table is full
8	required internal data area found to be corrupt
9-13	reserved
14	temporarily unable to access the requested object
15	last Status Response message of cycle

Except as described under “status reporting” below, the serial number has no significance other than as a handle on the request to associate a reply with it. The recipient of a request shall simply copy the serial number to the response message, without making any assumptions about serial number values; in particular, it shall not ignore or reject a message which appears to it to be out of sequence. The sender might use a single number space, or a separate number space for each message type, and might reserve a value (zero, perhaps) for messages where the OID is sufficient to identify to what the response is replying.

The OID and the value form an SNMPv1 VarBind except that the enclosing SEQUENCE is omitted. Note that the tag in the 3rd octet shows whether password information is present. The value shall be omitted in the Get and GetNext requests, and in any response which cannot include a correct value, in which case no further OIDs shall follow.

Set (including non-volatile Set) is the only request that can change the state of the managed unit (except to the extent that requesting a status report, or reading objects such as `unitNextFlowId` and `unitNextCallId`, changes its state); it asks to change the value of the object indicated by the OID to the value in the message. If the object cannot have the requested value, it should be set to the nearest possible value; for instance, if it can take values 0, 10, 20, ..., 100 and the request is to set it to 37 then it should be set to 40.

Non-volatile Set is processed in the same way as Set, but also written to the flash; commands in the flash are processed on start-up. The value written to the flash is the value in the request, not the one in the reply, and will replace any command for the same OID that is already in the flash; if a non-zero status code is returned, nothing is written to the flash. If status code 6 is returned, the command has been processed as a normal (volatile) Set. A non-volatile Set with an OID but no value will delete any command for that OID in the flash; the status returned is 0 if a command was deleted, 2 if none was found, otherwise 5 or 6.

A response message has the same message type and serial number as in the corresponding request. For Get and Set the OID is also the same as in the corresponding request, and the

value is the current value of the object with that OID if possible, otherwise the value is omitted (so the message ends at the end of the OID, or possibly earlier). Status code 1 (the SNMP “tooBig” code) indicates that the response has been truncated at the maximum message size; the VarBind should be truncated rather than being omitted altogether.

The response to a Set shows the value of the object immediately after the change is made, so in the example above the value in the message will be 40 (as specified in IEC62379, though SNMP requires it to be 37, which is not helpful). If the value in the request is of the wrong type for the object (status code 3) or the object is read-only (status code 4), the response shows its current value in the same way as a Get response.

If the OID in a request message is corrupt, or the message type is not recognised, a response should be returned which has status code 5 and is a copy of the message up to and including the point where the error occurs (but always at least 2 octets), for instance it is 2 octets long if the message type is not recognised, 3 if the tag in the 3rd octet is neither OCTET STRING nor OBJECT IDENTIFIER.

The object count supplied with a GetNext request is 1 less than the maximum number of objects to be included in the response, or 15 to indicate “as many as will fit in the maximum MTU size”. The first object in the response is the “lexicographical successor” (as in SNMP) of the OID in the request, and each of the others is the “lexicographical successor” of its predecessor. This is similar to SNMP GetNext when the object count is zero, GetBulk otherwise. If there are no further objects in the MIB, the OID is coded as a single byte with value 127 (so that it is greater than any legitimate OID), and the value is omitted; note that this is different to the SNMP response which shows the requested (or preceding) OID and the special value “endOfMibView”. If there is more than one OID in the reply, the status shall be 0; if an error occurs when accessing an object other than the first, the reply shall end with the preceding object, so that a subsequent GetNext can return the error code in a reply that does not include any other objects. The “end of MIB” OID is regarded as an OK value, even if it is the only OID in the reply; thus, unlike SNMP, GetNext will not return the noSuchName status value. Also unlike SNMP, it will not report the object (if any) whose OID is in the request.

Either party may send requests (as was the case with ILMI in ATM networks), so Set messages for a MIB in the management station take the place of SNMP Traps.

> *We need to stop this happening in the case where the management station is not set up to accept unsolicited messages, maybe by saying not to do it unless a Status request has been sent. The Windows version was set up with OnReceive() fielding incoming packets whenever they appeared, and the Mac seems to support callbacks in CFSocket that would do something similar, but I haven't found a way to do it in the platform-independent Berkeley Sockets code yet. Maybe we need to structure it so that the platform-dependent callback routine calls a platform-independent routine when a packet arrives. Currently it's set up so that it only looks for replies to requests it has sent.*

There is thus some simplification compared to SNMP but the basic functions of reading and writing the values of objects and walking the OID tree are the same. With SNMP the model was that the software in the management station sent a fixed-format data structure (but in a variable-format encoding) which the software in the managed unit (or its agent) filled in and sent back, so fields that were going to be needed in the response (such as the value of a variable) had to be included in the request also, and an errored response included "junk" data which could be misleading.

We only allow one object to be read or written per message (except in the reply to GetNext); when using a Flexilink background flow there is less benefit in batching requests together in a

single message than on a connectionless packet network, and as well as simplifying the implementation of the protocol it avoids the misfeature that if any of the objects raises an error the operation fails on all the others as well.

Unit information

When a management connection is made, MIB objects are read to discover the kind of equipment the unit is and the version of software it is running, and potentially to update the software; MgtSocket::upd_state shows what stage this process has reached.

The first request, sent by MgtSocket::ConnectionMade(), is a GetNext for 1.0.62379.1.1.1 (unit-information). Responses are processed by MgtSocket::ReceiveData(); in some cases another request is sent immediately, in others the state is updated and the next request is sent by MgtSocket::OnIdle().

If the controller has “maintenance” privilege, it then downloads a map of what is in the unit’s flash memory (see MgtSocket::flash) and checks whether the software is up-to-date. If not, a CQuery object is created to ask the user whether to upload the version in the product file.

Passwords

The current regime for password protection is described under the “Privilege and passwords” and “Security issues | Passwords” headings in the “Extensions to Flexilink implementation for BCU” document. The passwords are stored in clear as part of the MIB, including in the “non-volatile configuration” page in the flash. We expect that it will be replaced by a more robust and less computationally-intensive protection mechanism at some point in the future; the mechanism to be used on any particular management connection will be specified (or negotiated) in the FindRoute messages that set it up.

For any encryption/authentication mechanism, any per-message information it requires is supplied in an octet string at the start of the message, i.e. with its OCTET STRING tag in the 3rd octet. If the 3rd octet is not an OCTET STRING tag, the message does not include any such information.

Currently, Status messages are outwith the protection mechanism.

Status broadcasts

The “status broadcast” mechanism was originally intended as a scalable way of keeping a potentially large number of controllers informed of a unit’s state by the unit providing a multicast flow on which it would report a suitable subset of its MIB objects, reporting any change immediately and (with a lower priority) periodically reporting all the objects in the set.

As currently implemented, the reports are sent to each controller individually, on the management connection. The controller sends a 2-octet Status request message which the unit acknowledges with a 2-octet Status response message. This needs to be repeated periodically as a “keep-alive” mechanism. Later versions might allow individual objects to be monitored by including their OIDs in the request message, or by writing a table indexed by the OIDs. However, it may be preferable to multicast the status messages, which would work better if there aren't too many options; it requires a way for the VM to send on guaranteed-service flows, which may be useful for other purposes also. The details of the current implementation are as follows.

The managed unit reports state in Status response messages which include zero or more (OID, value) pairs (at least one if the “status” field is coded as 0), up to a maximum message size of 1400 octets. See <StatusBroadcast> in the TeaLeaves code for the objects that are monitored in each “cycle”.

The *n*th message of a cycle has serial number *n*; if *n* reaches 255, it then wraps to 2, missing out 0 and 1. These messages are referred to as *in-cycle* messages; note that they never have serial number 0 and only the first of a cycle has serial number 1. The “status” field is coded as 15 for the last in a cycle, 0 for all others. The last message in a cycle may be just 2 bytes long; this will occur if all the objects it would have reported have ceased to exist since the previous in-cycle message was sent. Objects which have changed are reported in *change* messages which have serial number 0 and do not form part of the cycle.

With each object is associated a flag which is set when the object’s value changes and cleared when it is reported in a message (not necessarily a change message). The managed unit decides what messages to send as follows.

If it is more than 2 seconds since the last in-cycle message was sent, then if the current cycle is incomplete the next message in the cycle is sent, or if the cycle is complete and it is more than 20 seconds since its first message was sent then the first message of a new cycle is sent.

Otherwise, if any object has its flag set, and it is more than 100ms since the last message was sent, a message with serial number 0 is sent, including as many of the objects with their flags set as will fit in 1400 octets.

Each message is sent to all management stations that have sent a Status Request within the last 2 minutes. Thus the first message a management station receives after requesting Status messages will not necessarily be the first of a cycle, and it needs to repeat the request periodically in order to continue to receive Status messages. Note that if a management station is connected via UDP/IP there is no explicit indication to the managed unit when a management station is shut down, and the timeout on the UDP “connection” is quite long.

If status is multicast on a guaranteed service flow, messages should be sent at the rate supported by the flow, with a new cycle starting as soon as the previous one has finished and “change” messages taking priority except that no more than three “change” messages for an object shall be sent during any one cycle. The format should be reviewed, to be more appropriate to guaranteed service packets, e.g. omitting the 2-octet message header and setting f=0 in the last packet of a cycle and f=1 in the rest.

Console data

Messages of type 7 carry data from the management station's user in one direction, and text for display on the management station's screen in the other direction. In each case the remainder of the message (from the third octet onwards) consists of UTF-8 text; currently messages from the management station are always a single character.

The response acts as an acknowledgement; status values are as follows:

0	text received and passed on to the relevant process
1	text received but could not be passed on
2-15	reserved

Currently, all console output from a unit is sent on the management connection on which a type 7 message was most recently received; when there is no such connection, up to about

70k bytes of data are saved up and will be output when the next type 7 is received. If the limit is exceeded, the oldest text is discarded.

Within the text, newline is coded as line feed (which is `\n` in TeaLeaves strings).

I haven't checked what the rules are about CRLF and newline in UTF-8.

Analyser connections

The analyser downloads and interprets diagnostic information using the Flexilink version of the Simple Control Protocol specified in Annex C of GS NIN 005. (Note that the earlier UDP version, described in the February 2016 version of the Flexilink Protocols document, is no longer supported.) There is an AnalyserDoc object for each connection.

The SCP engine is defined in GS NIN 005 to use bit addresses, in a 32-bit field; it might have been better to use a 48-bit field for the address.

The SCP engine communicates over a Flexilink physical link, either as a "simple" device or shared with the port's normal functionality. Currently, each Aubergine unit has a SCP engine on its port 4, which identifies itself as an Aubergine debug interface and is listed in the Sources window under its link partner (through which any communication with it must pass).

The line in the Sources window is in black if it is free; in that case a left click on it creates an AnalyserDoc object which connects to it. A right click deletes the AnalyserDoc object. The line has a red highlight if the CPU in the target unit has stopped, or green if it is waiting for software to be uploaded by the TeaLeaves Compiler. (This only happens if the FPGA logic is loaded via the JTAG interface.) Note that only one connection can be made to a SCP engine, so if an analyser is connected to it the Compiler will not be able to upload software.

Typing a question mark followed by ENTER in an analyser window lists the available commands.

The dRAM is read and written via the VM's data cache, so the address is in the same format as in the VM, including the "area" field, except that if the area is coded as "nil" it selects resources other than the dRAM (such as the debug buffer, the VM's registers, and the identification word that is located at address zero).

The 'd' command reads the debug buffer, which contains 512 samples of each of 512 signals. Which signals are sampled depends on the setting of the SCP_DEBUG constant in the logic; logic loaded from flash usually has it set to "VM SOFTWARE" which traces the VM code. The result is not meaningful unless the status is shown as "data captured".

The format in which the debug buffer is displayed is defined by a "debug dump format" file with name `vm.9t4dbf` if tracing the VM code, `logic.9t4dbf` else. The file is plain text (UTF8) and should be prepared with a text editor. If it is not found (in the same directory as the `.exe` file from which the controller was run) the contents are displayed in hex.

The last non-empty line in a `.9t4dbf` file (lines with only one or two characters may count as empty) defines how the information from the debug buffer is presented, and any other non-empty lines are a heading. The 'd' command outputs the heading lines to the console window at the head of the data, and repeats them periodically. Each data line begins with its line number (in hex, in the range 000 to 1FF, followed by a colon and a space); lines that are identical to the previous line are omitted. The following table lists the format characters; each character consumes the number of bits indicated in the second column from the 512-bit data

value, and (except for '$') generates one character of the line, so the output is the same length as the format string.

format	*bits*	*output*
digits '1' to '4'	1 to 4	value as a hex digit
'8','c','e'	1 to 3	value left aligned in a hex digit (e.g. 'c' produces '0', '4', '8', or 'C')
'm'	1	'M' if a 1, 'R' if a 0 (for microcode ROM addresses in a previous version of the VM; now deprecated, may be withdrawn)
upper case letter	1	letter if a 1, space if 0 (intended for flag bits, letter = mnemonic)
'?'	1	'?' if a 1, space if 0 (intended for bits that should always be 0)
'b'	1	space (intended for skipping unwanted bits)
'h'	4	space (intended for skipping unwanted bits)
'w'	16	space (intended for skipping unwanted bits)
'$'	5	3-character register name; following 2 characters in format ignored (so 3 characters of the format string are consumed)
others	0	same character as the format (intended for separators, e.g. space)

If the total number of bits represented is less than 512, the remaining bits are not displayed, though they are still used when comparing lines to see whether they are identical. If more than 512, the values of bits beyond the 512th will be rubbish.

File formats

Product file

A “product file”, holds information about one or more product types. It contains one or more lines each consisting of zero or more characters coded according to ISO/IEC 646. Lines are separated by carriage return (code $0D_{16}$). Line feed (code $0A_{16}$) may follow the carriage return and is ignored if it does.

The first line is “`#62379.iec.ch::productfile:formatversion1`”. Subsequent lines that begin with a ‘`#`’, also empty lines, are “comment” and shall be ignored. (The first line shall not be empty.)

Apart from the first line and comment lines, each line shall consist of a keyword, optionally followed by the character ‘=’ and a value. When interpreting a line, the keyword shall be everything before the first ‘=’, and the value shall be everything after the first ‘=’; in each case leading and trailing white space characters (codes 00_{16} to 20_{16} inclusive) shall be removed, and any internal sequence of white space characters shall be replaced by a single space. If there is no ‘=’ on the line, the keyword shall be the whole line (after removing/replacing white space characters) and the value shall be empty. Keywords shall be case insensitive.

The information about each product shall begin with a line with the keyword "`product`" and a value consisting of four hex numbers separated by spaces, which are the four numbers that make up the `unitIdentity`. It shall include everything until the next line with the keyword "`product`", or until the end of the file if there are no more such lines.

There may be a line with the keyword "`description`", the value of which shall be a human-readable description of the product.

For each software component, there shall be a line with a keyword which consists of "`software`", a space, the area type in hex, another space, and the area class in hex. The value shall be the filename of the file containing the component of that type.

Binary image

The format for `.9t4bin` is the binary format specified in 5.2.4 of IEC62379-1; the data consists of 32-bit words (represented big-endianly) as follows:

1 word:	the version number, one octet per component: *v4* (0 if not a beta), *v1*, *v2*, *v3*
1 word:	32*n*
1 word:	pointer to the constants, assuming the code starts at dRAM address 0
n words:	the code, consisting of VM operations (padded to a multiple of four octets) followed by an image of the statics area; see the VM documentation for details
1 word:	checksum: sum of the other *n*+3 words (modulo $2^{32} - 1$), in range 1 to $2^{32} - 1$

There's a note in `MakefileDoc::Compile()` pointing out that the "pointer to the constants" word is redundant; changing where the code starts would be dangerous, but the 3rd word could be regarded as "reserved".

Symbol table

A symbol table file (`.9t4sym`) contains plain text (UTF-8), and the first line confirms the VM version; for the current VM it is:

```
:#ninetiles.com::9tos3:symbols:VM4:0
```

The remainder of the file contains a number of sections, each of which is introduced by a line beginning with a colon (with no leading white space) and delimited by another (the start of the next section, or the end-of-file line). Blank lines may occur between sections, and should be ignored anywhere.

Currently there are just three sections; the first section lists all the labels, introduced by

```
:labels
```

which is followed by a line for each label, containing the byte offset from the start of the code (in hex), white space, and the name; for anonymous labels, the name is "[*n*]" where *n* is a decimal number. The last line has the length of the code and the name "[top]".

The next lists the global variables, introduced by

```
:globals
```

which is followed by a line for each global variable.

The third section lists the local variables (including parameters), introduced by

```
:frames
```

which is followed by an entry for each routine. The first line of each entry contains the entrypoint (as a byte offset from the start of the code, in hex), white space, the length of the stack frame's bit string as a hexadecimal number of bits, white space, and the name.

There follows a line for each local variable. The end of the list of variables is marked by a line with no leading white space, which, if it does not begin with a colon, is the entrypoint of the next routine.

The file ends with a line containing

```
:end
```

The line describing a variable (local or global) is formatted as

```
_rct_o_l_n
```

in which '_' indicates white space. The different parts are as follows:

r is R (called by Reference) if the type was prefixed by '§' (in which case *c* and *t* refer to the target variable while *o* and *l* refer to the ident) otherwise it is blank

c is the class, coded as: B = pure Bit string, E = compound (contains Entrypoint), P = complex (contains Pointer)

t indicates the type as: N = int, U = uint, E = entrypoint, I = ident, S = struct, P = pointer, A = array, X = index

o is the offset from the start of the stack frame's bit string, as a hexadecimal number of bits

l is the length (excluding any padding at the end), as a hexadecimal number of bits

n is the name