FLEXRADIO 6000
SmartSDR

THE APPLICATION PROGRAMMING INTERFACE

A PRIMER

John Linford, G3WGV

Table of Contents

Change control

Date	Version	Change
2016-09-13	0.000	Initial document created
2016-11-12	0.001	Additional material Draft placed on my Dropbox for comments
2016-11-13	0.002	Further additional material Copyright and license notices added
2016-11-13	1.000	First formal release
2016-11-15	1.001	Minor corrections Added section on VITA49 protocol Added acknowledgements Added new section: *Notes on sending commands to the API*
2016-11-20	1.002	Peer review completed by FlexRadio. Minor changes and corrections. Added trademark note

About this document

Copyright

License

Disclaimer

Although I have tried to make this document complete and accurate, no warranty is offered in this respect. The purpose of this document is to encourage the reader to embark on his or her own voyage of discovery, learning along the way as I did. If you do find any errors then please let me know!

Acknowledgements

Particular thanks to Eric Wachsmann, KE5DTO, Vice President, Software Development at FlexRadio for fielding my numerous questions and providing such great insight as I travelled the API road. Eric also reviewed this document, making useful corrections and improvements.

Thanks too, to the many great people on the FlexRadio Forums that have helped me in so many ways.

Trademarks

Introduction

This document is a repository of knowledge gained during the development of my FLEX-6000 Radio Controller project.

My development environment is Delphi XE7 and I use the Indy (Internet Direct) TCP and UDP client components to interface with the API. I will try to keep this document language agnostic.

It is assumed that you have the necessary knowledge to create and manage UDP and TCP clients, together with an understanding of handling byte and character stream data to and from these client ports.

This document is an attempt to capture the learning processes that I went through when I first started looking at the FlexRadio API. The API Wiki is incomplete and in some cases no longer correct because the API has moved on since the documentation was created.

It is perhaps unreasonable to expect FlexRadio to maintain this when it should be using its resources on developing products. I would argue that it is up to the API programming community to put some effort into providing the documentation it needs.

A list of resources including the FlexRadio API Wiki is given at the end of this document.

Overview of the API

The FlexRadio 6000 API is an Ethernet-based interface that exposes a wide range of radio parameters and permits a detailed level of control of the radio. It features a mix of TCP/IP and UDP [1] traffic.

The radio is, in networking terms, a server. Applications that wish to interface with the radio are clients. A single radio server can have multiple clients. It is also possible to run multiple clients connecting to the same radio on a single computer system.

In the Flex API, UDP is used for the Discovery phase. In addition, it is used to stream information such as S-meters, audio, FFT (waterfall/panadapter) and so on. TCP/IP is used to receive radio parameters such as slice settings. It is also used to send commands to the radio.

Broadly speaking, for every parameter the radio is able to send to the client, there is an equivalent command that permits the client to control that parameter. So, for example, the radio can tell the client what frequency a slice is currently on and the client has a command that lets it set the slice frequency.

[1] TCP/IP is a connected protocol, that is, data can only move between the server and a client when that client is connected on a unique port on the server (radio). UDP is an unconnected protocol. It can be used to broadcast information to any clients that choose to listen on the UDP port.

Anatomy of an API session

Discovery

Whenever the radio is operating, it sends out discovery messages on UDP approximately once per second. Discovery messages are always sent on port 4992. The discovery message, or protocol, contains information such as the radio type, serial number, user call sign, nickname, IP address & port, etc.

This message contains all the information a client needs to make a TCP/IP connection to the radio, thereby establishing a client session.

The client starts off listening for these discovery messages. There may be multiple radios on the network and the client needs to decide which of these it wishes to connect to, based on serial number or some other parameter.

This is the end of the discovery phase for this client. The client can now close UDP port 4992. The radio continues broadcasting discovery messages so that other clients may initiate the discovery process.

TCP/IP Connection

Once the client has discovered the radio it wants to connect to it creates a TCP/IP port and then attempts to establish a TCP/IP connection with the radio using the IP address and port from the discovery message.

Upon connection, the radio sends its software version and a unique handle for this connection. The TCP/IP connection is used to send commands to the radio and to receive status information back from the radio. These data are relatively low volume but it is important than messages are not missed or corrupted, so TCP/IP is the ideal transport.

UDP streaming data

By default the radio streams UDP data to port 4991. As there can be multiple clients, it is necessary to be able to set a unique port for the client. This is done by issuing a TCP/IP command 'client udpport <port#>'.

UDP is used to stream information from the radio that needs to be sent with minimum delay or latency but where the occasional packet loss is unimportant. Examples of UDP streamed data include various meters (S-meter, Power output, SWR and many more), digital audio streams and panadapter/waterfall display data.

Subscription

When other clients change radio parameters, e.g. sending an updated slice frequency, the radio can be instructed to send the new parameter value out to other clients. Clients wishing to receive these data “subscribe” to the update stream.

There are various subscription classes, e.g. Slice, Radio, Transmit, Meters etc. The client subscribes to those classes of data it wishes to receive.

The subscription method is also used to subscribe to streaming data sent over the UDP port. Hence, the client can subscribe to meter data (Signal strength, Tx power out, SWR, etc.), audio or FFT and other streams.

Using the API

Once you've connected to the radio, set the streaming UDP port and subscribed to the classes you are interested in you can now start using the API. Sequentially numbered commands are sent to the API and the API responds with the command sequence number and a completion code indicating success or the reason for failure.

The API will also send unsolicited updates to the application whenever there is a change to the radio's environment. This change could be as a result of another client's actions or may be generated autonomously by the radio.

In addition to TCP/IP traffic, which is generally command/response generated, data are continuously streamed on the streaming UDP port. The application must be able to read these data at any time.

Disconnection

When the client no longer wishes to communicate with the radio it issues a TCP/IP disconnect. The radio drops the TCP/IP connection and closes any subscriptions that may be in force for that client. UDP streaming on the radio's UDP port for that client also terminates.

Discovery

The discovery packet is always sent by the radio server on port 4992, so the client need only open a UDP client listening on that port in order to receive discovery messages.

Figure 1 is a “Wireshark” capture of an actual discovery packet. The first 42 bytes are UDP protocol stuff that need not bother us. The VITA49 data payload is highlighted in blue.

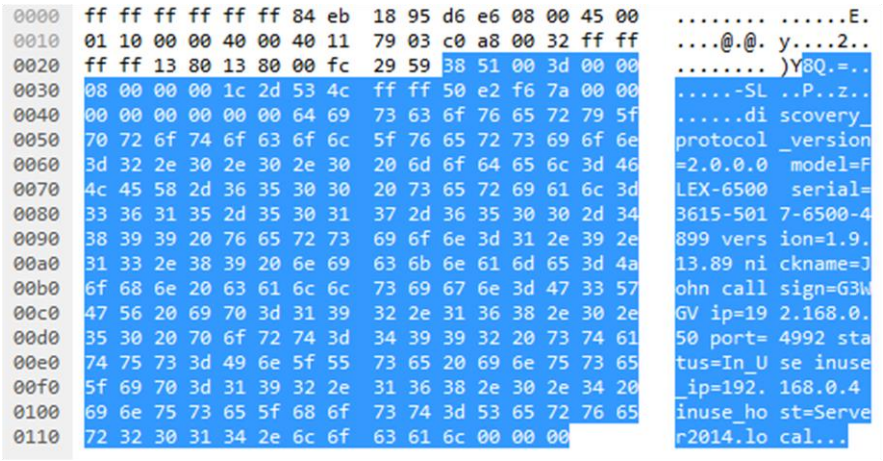


Figure 1

Datagram byte #	Payload byte #	Content
2A..2B	0..1	VITA49 header
2C..2D	2..3	Size of payload in words. Multiply by 4 to get byte count. Byte 2 is MSB.
2E..31	4..7	Stream ID: 0x00 00 08 00. This is the unique ID used by FlexRadio for discovery packets.
32..39	8..15	Discovery class ID: 0x00 00 1C 2D 53 4C FF FF
3A..45	16..27	Date stamps (not used)
46..end	28..end	Discovery message parameters in ASCII

Table 1

In Table 1 the payload byte number (in decimal) is the offset you need to get to the associated field. Use the Datagram byte number (in Hex) to find the field in Figure 1, which contains the UDP protocol header.

Start by extracting the length and converting this into bytes.

Next, extract the Class ID. Discovery messages have a class ID of 0x00001C2D534CFFFF. Other UDP messages from the Flex radio are the same apart from the last two bytes, so it is only necessary to compare those bytes for 0xFFFF.

At this stage you know that you have a discovery packet. Fields within the packet start at byte number 28 and are in the form

Field_name=field_value

Fields are separated by spaces but there are no spaces within the field itself. It is therefore a simple matter to extract field pairs one by one and obtain the field_name and field_value for each field. My method of parsing this is to extract a word at a time. This word is a field_name=field_value pair. Now replace the = with a space and you have two words, the first of which is the field_name and the second is the field_value.

Example fields:

discovery_protocol_version=2.0.0.0
serial=3615-5017-6500-4899

Note that the 3rd group in the serial number gives the model number. This might be useful in your code!

Here is a complete list of fields:

```
'discovery_protocol_version', //1
'model',                      //2
'serial',                     //3
'version',                    //4
'nickname',                   //5
'callsign',                   //6
'ip',                         //7
'port',                       //8
'status',                     //9
'inuse_ip',                   //10
'inuse_host'                  //11
```

Once you've extracted all the field values you have more than sufficient information to move on to the next step: connecting to the radio using TCP/IP.

Connection

The discovery phase gives us all the information we need to connect to the radio using TCP/IP. In particular, as TCP/IP is a connected protocol, we have to specify both the server (radio) IP address and server port number in the TCP client interface. These are the discovery 'ip' and 'port' parameters respectively.

When the TCP client connects to the radio the API sends two parameters to the client:

The API version number, prefixed with 'V', e.g. **V**1.2.0.0
A unique 4-byte 'handle' for this session, prefixed with 'H', e.g. **H**BE013ADC

Neither of these are terribly important to the operation of the client software. I capture both parameters but have yet to find a use for either!

The radio now sends a bunch of initial information via the API. Something like this:

```
V1.2.0.0

H545A4ACD

M10000001|Client connected from IP 192.168.0.4

S545A4ACD|radio slices=3 panadapters=3 lineout_gain=53 lineout_mute=0
headphone_gain=51 headphone_mute=0 remote_on_enabled=0 pll_done=0
freq_error_ppb=0 cal_freq=15.000000 tnf_enabled=0 snap_tune_enabled=1
nickname=John callsign=G3WGV binaural_rx=0 full_duplex_enabled=0
band_persistence_enabled=1 rtty_mark_default=2125
enforce_private_ip_connections=1

S545A4ACD|radio filter_sharpness VOICE level=2 auto_level=1

S545A4ACD|radio filter_sharpness CW level=2 auto_level=1

S545A4ACD|radio filter_sharpness DIGITAL level=2 auto_level=1

S545A4ACD|radio static_net_params ip= gateway= netmask=

S545A4ACD|interlock timeout=0 acc_txreq_enable=0 rca_txreq_enable=0
acc_txreq_polarity=0 rca_txreq_polarity=0 tx1_enabled=1 tx1_delay=0
tx2_enabled=1 tx2_delay=0 tx3_enabled=1 tx3_delay=0 acc_tx_enabled=1
acc_tx_delay=0 tx_delay=0

S545A4ACD|client 0x545A4ACD connected
```

Figure 2

This gives us our first insight into the way the API communicates with client applications.

Each message starts with a single letter that defines what the message type is. So far we have the following:

V	Version number
H	Handle
M	radio message
S	Status message

All messages follow this convention

After the Version and Handle messages the next message we see is

```
M10000001|Client connected from IP 192.168.0.4
```

This is an example of a radio *Message*. The radio sends these out when something happens that it thinks everyone should know about. In this case, it is telling all clients, including us, that we have just connected.

```
S545A4ACD|radio slices=3 panadapters=3 lineout_gain=53 lineout_mute=0
headphone_gain=51 headphone_mute=0 remote_on_enabled=0 pll_done=0
freq_error_ppb=0 cal_freq=15.000000 tnf_enabled=0 snap_tune_enabled=1
nickname=John callsign=G3WGV binaural_rx=0 full_duplex_enabled=0
band_persistence_enabled=1 rtty_mark_default=2125
enforce_private_ip_connections=1
```

This is a *Status* message. The API sends a considerable volume of status messages, usually as a result of another client making some change, e.g. changing frequency. In this case the status message is giving us some initial data about the radio.

Following the Status letter 'S' is the handle associated with this status change. In this case it is our own handle. If the status message is as a result of another client's activity then that client's handle is shown. This enables you to determine which client did what… if you need to.

The handle is followed by a vertical bar '|' and then a status descriptor, in this case 'radio'. This tells us that what follows are values associated with the radio as a whole. All status messages follow this convention.

```
slices=3 panadapters=3 lineout_gain=53 …
```

Next, follows a series of Field_name=field_value pairs, using exactly the same format as we saw in the discovery protocol. The extract above tells us that there are 3 unused slices, 3 unused panadapters and that the line output gain is set to 53. Other fields follow the same convention and should be largely self-explanatory.

```
S545A4ACD|radio filter_sharpness VOICE level=2 auto_level=1
```

These next few lines are new as of SmartSDR version 1.9.xxx. They relate to the new feature that permits the user to set different filter sharpness/latency values on a per mode basis. This message is slightly odd in that it does not follow the usual convention of Field_name=field_value pairs. The intent is clear enough though and it should be easy enough to parse.

```
S545A4ACD|interlock timeout=0 acc_txreq_enable=0 rca_txreq_enable=0
acc_txreq_polarity=0 rca_txreq_polarity=0 tx1_enabled=1 tx1_delay=0
tx2_enabled=1 tx2_delay=0 tx3_enabled=1 tx3_delay=0 acc_tx_enabled=1
acc_tx_delay=0 tx_delay=0
```

This next status message introduces a new status descriptor, 'interlock'. This tells us a load of information about the various Tx/Rx interlocks and should be self-explanatory.

```
S545A4ACD|client 0x545A4ACD connected
```

The final status message introduces another new status descriptor, 'client'. It provides the handle of this client session to any other client that might need it.

The connection sequence is now completed and we can move on to the next phase, setting the UDP streaming port. First, we need to understand the command structure.

Command/response structure

Commands are sent to the radio on the TCP/IP channel. Every command receives at least a response acknowledgement message from the radio and in some cases further information as well.

Command structure `CD<command_number>|<command>`

`C`	Indicates that this is a command to the radio
`D`	Optional flag to request additional diagnostic information
`<command_number>`	A sequential command number, starting at zero
`<command>`	The command text

Response structure `R<command_number>|<result>|<diagnostic_text>`

`R`	Indicates that this is a response from the radio to a command
`<command_number>`	The command number that this response relates to
`<result>`	The result of the command's execution. Zero indicates OK
`<diagnostic_text>`	Additional diagnostic text if the D flag was set in the command

The vertical bar separators are important and must be present.

Examples

`c0|client udpport 4993` `R0|0|`	Set the streaming UDP port to 4993
`c1|sub slice all` `R1|0|`	Subscribe to all slice status updates
`c2|slice t 0 7.025` `R2|0|`	Tune slice 0 to 7.025MHz

If an error occurs then the response will be non-zero. Error response numbers are not generally very helpful as most are not documented and there is no numbering consistency between the various commands. If you get an error that you can't make sense of then you can always set the diagnostic flag in the command that failed and see if that provides useful assistance.

Setting the UDP streaming port

In a multi-client environment each client needs its own UDP streaming port if it wants to receive streamed data. Most clients will want streaming data. It includes, for example, meter values such as signal strength, power output, etc. The default streaming port is 4991 and that is fine if you will be in the unusual position of being the only client talking to the radio. In most practical cases you'll need to set the port to something else.

Generally, it makes sense to set the streaming port at the beginning of a session. The following command is sent on the TCP port:

```
c<command_number>|client udpport <port#>
```

e.g.

```
c1|client udpport 4993
```

You may not know what UDP ports are already in use, so it's a good idea to first find a free port. Try starting at, say, 4993, test the port and keep moving up by one until you find a few port.

The radio will respond with

```
R1|0|
```

At this point, you will start receiving streaming data on the specified port. Make sure you've initialised the port!

Subscribing to objects

When radio settings change you probably need to know about them. There are three sources of change:

- The radio itself
- Your client
- Other clients

The general principle is that you should be aware of things that you've done to change the radio's settings. So the Flex API does not, in general, echo back changes to the client that requested them. But you do need to know if other clients or the radio itself have changed something.

You have full control over the types of changes that you will receive notifications for, using the Subscribe command. There are various classes of update and you can subscribe to just those that you need:

Slice	GPS
Meter	Audio_stream
TX	Xvtr
ATU	Memories
CWX	DAX
Pan	DAXIQ

Within some subscription objects you can chose to subscribe to specific updates or to all updates associated with that object.

Examples

`C4|sub slice all` Subscribe to all slice updates
`R4|0|`

`C5|sub slice 0` Subscribe to slice 0 updates
`R5|0|`

`C6|sub meter all` Subscribe to all meter updates (streaming data)
`R6|0|`

When you subscribe to an object, the API will immediately return the current settings. See figure 3.

As before, the message is a Status message and it contains everything to do with the slice. As we subscribed to all slices, this will be repeated for each slice currently in use.

For reasons that aren't immediately clear, the API also sends a separate audio gain/pan/mute update. It's probably not meant to do that!

Some subscriptions result in streaming data being sent on the UDP streaming port. Meter, Pan, Audio_stream, DAX and DAXIQ do that. There may be others too.

```
c0|sub slice all

R0|0|

SB76508BD|slice 0 in_use=1 RF_frequency=14.042540 rit_on=0 rit_freq=0
xit_on=0 xit_freq=0 rxant=ANT1 mode=CW wide=0 filter_lo=-300
filter_hi=300 step=5 step_list=1,5,10,50,100,200,400 agc_mode=med
agc_threshold=65 agc_off_level=10 pan=0x40000000 txant=ANT1 loopa=0
loopb=0 qsk=1 dax=0 dax_clients=0 lock=0 tx=1 active=1 audio_gain=75
audio_pan=50 audio_mute=1 record=0 play=disabled record_time=0.0 anf=0
anf_level=0 nr=0 nr_level=0 nb=0 nb_level=50 wnb=0 wnb_level=0 apf=0
apf_level=7 squelch=1 squelch_level=20 diversity=0 diversity_parent=0
diversity_child=0 diversity_index=1342177293 ant_list=ANT1,ANT2,RX_A,XVTR
mode_list=LSB,USB,AM,CW,DIGL,DIGU,SAM,FM,NFM,DFM,RTTY fm_tone_mode=OFF
fm_tone_value=67.0 fm_repeater_offset_freq=0.000000
tx_offset_freq=0.000000 repeater_offset_dir=SIMPLEX fm_tone_burst=0
fm_deviation=5000 dfm_pre_de_emphasis=0 post_demod_low=300
post_demod_high=3300 rtty_mark=2125 rtty_shift=170 digl_offset=2210
digu_offset=1500 post_demod_bypass=0 rfgain=0

SB76508BD|slice 0 audio_gain=75 audio_pan=50 audio_mute=1

SB76508BD|waveform installed_list=
```

Figure 3

If, having subscribed to the slice updates, another client makes a change then the change is reported to your client as well:

```
S854090FE|slice 0 RF_frequency=14.042545 wide=0 lock=0

S854090FE|slice 0 RF_frequency=14.042550 wide=0 lock=0

S854090FE|slice 0 audio_gain=76 audio_pan=50 audio_mute=1

S854090FE|slice 0 audio_gain=77 audio_pan=50 audio_mute=1

S854090FE|slice 0 filter_lo=-305 filter_hi=305 post_demod_low=300
post_demod_high=3300

S854090FE|slice 0 filter_lo=-310 filter_hi=310 post_demod_low=300
post_demod_high=3300
```

Figure 4

In this case, a couple of frequency changes were made by the client with handle 854090FE followed by a change in audio gain and new filter width settings. Note that in all cases we get a bit more than just the parameter that changed. I'm not sure why that happens but I suspect it's just that update objects are grouped together to reduce the total number of unique objects.

You can see that the updates when another client makes changes are just a subset of the full slice update that we got when we subscribed. It makes sense, therefore to write your code so it will deal with all the fields that the full update can send. It will then deal with individual control updates as well, since the syntax is the same.

Subscription not required!

Some objects send updates to a connected client without the need for any subscription. I can't find any documentation that tells me what these are but I have observed the following:

Interlock status messages

These status messages are associated with TX/RX interlocks and the various timings and other settings that are contained in the SmartSDR for Windows radio setup TX tab. This includes changes to the TX1/TX2/TX3 timings and enabled status.

Radio status messages

The radio status messages include radio-wide settings such as changes in the number of slices available, line out and headphone gain/mute, nickname and callsign.

Client status messages

Client connection and disconnection messages.

Some useful commands

The best way to find the exact syntax for API commands to the radio is to find the command in the FlexLib library (see references section). This library is reissued each time a new version of SSDR is released, so it is reasonable to assume that it is complete and accurate.

The command set is not particularly consistent in that sometimes it uses an = sign when setting a value and on other occasions it does not. Boolean (on/off values) are indicated by a number. 0 means off, any non-zero number means on.

What follows is only a tiny subset of the commands that are available to the API programmer. Again, the FlexLib code should contain all possible permutations.

Subscribe

`C0|sub slice all`	Subscribe to all slice updates
`C1|unsub slice all`	Unsubscribe from all slice updates

Slice control

`C2|slice c 7.025 ANT1 CW`	Create a new slice on 7.025MHz, CW, using ANT1
`C3|slice tune 0 7.026`	Tune slice 0 to 7.026MHz
`C4|slice set 0 mode=USB`	Set slice 0 to USB
`C5|slice lock 0`	Lock slice 0 tuning
`C6|slice unlock 0`	Unlock slice 0 tuning
`C6|audio client 0 slice 0 gain 50`	Set slice 0 audio gain to 50 (mid scale)
`C7|audio client 0 slice 0 pan 50`	Set slice 0 pan to 50 (centre)
`C8|audio client 0 slice 0 mute 1`	Mute slice 0 (mute 0 for unmute)
`C9|slice set 0 apf=1`	Set slice 0 audio peak filter on (apf=0 for off)
`C10|slice set 0 apf_level=30`	Set slice 0 audio peak filter level to 30
`C11|slice r 1`	Stop slice 1

Radio control

`C12| mixer headphone gain 25`	Set headphone audio gain to 25
`C13| mixer headphone 1`	Mute headphone audio (0 to unmute)
`C14| mixer lineout gain 30`	Set line out audio gain to 25

Transmit control

`C15| transmit set max_power_level=20`	Set maximum Tx power to 25w
`C16| transmit set rfpower=70`	Set current Tx power to 70w
`C17| cw break_in 1`	Enable CW break in (0 to disable)
`C18| cw break_in_delay 100`	Set CW break in speed to 100ms
`C19| cw pitch 600`	Set CW side tone pitch to 600Hz
`C20| transmit set mon_gain_cw=40`	Set CW side level to 40

This should give you a flavour of the level of control available to the API programmer. As you can probably tell, I am a CW operator but you'll be pleased to hear that you can do all the same things for SSB or data mode operation. As always, see the FlexLib code for more.

Notes on sending commands to the API

Generally you can send commands to the radio as you please. Each command has a command number and the radio will execute them in the order they were sent, send back a receipt message with the same number and then progress to the next command. It is not necessary to wait for one command to be completed with response before sending the next command.

That said, it is not a good idea to flood the API with commands. A good example is fast VFO tuning using a spinner knob with fairly small frequency steps. It's not hard to send hundreds or even thousands of updates a second and that is a bad idea!

A realistic update rate for the radio is once every 25ms to 50ms. The radio seems comfortable with that. If you go faster than that you will likely start hearing odd audio clicks and other artefacts. The radio takes some time (typically around 50ms) to respond to commands, so nothing is gained by sending updates to the radio more often.

What I do is store the number of frequency steps that the VFO has moved during a 30ms period and then send the aggregated change to the radio. Zero the counter and start the process over again for the next 30ms period. In fact I do this for all controls, even audio gain, filter changes, etc. There is no issue with this from a human perception perspective – after all, you're still sending over 30 updates a second to the radio!

Another thing to consider is setting up a circular array that you fill with outbound commands and a pending flag. When the command response is received for that command number, set the flag as completed. This permits you to easily see if there are outstanding commands and get some idea of how fast you can send commands without having excessive outstanding requests.

You could even use this system to sequence the sending of commands, so you don't send the next command until the previous one has been acknowledged but this is not really necessary and might result in perceptible latency issues.

Using streaming UDP data

As of this edition of the API Primer, I have only attempted to handle two classes of streaming data:

- Discovery class
- Meter class

It may surprise you to find Discovery class listed as a streaming protocol but in reality that is what it is: it uses the VITA49 protocol specification and is sent via UDP. I dealt with the Discovery protocol earlier, so I will now look at the Meter protocol. It is significantly more complicated than the TCP/IP updates we have discussed already. It took me quite a while to figure it all out but hopefully this explanation will make it easier for other to understand.

It is important to realise is that there are potentially dozens, even hundreds of meters. Most will never be used but they are there. I am not aware of any full listing of possible meters that the Flex can support, indeed it is possible to create your own meters, although I cannot really see why.

The meter manifest

First, we have to understand the **meter manifest**. When you subscribe to Meter objects, the meter manifest is sent via the TCP port. Its purpose is to describe every meter that is currently being used by the radio. Figure 5 is an extract from the meter manifest that shows the meters we are most likely to be interested in. It is sent immediately after the meter subscription command, 'sub meter all'.

```
C3|sub meter all

R3|0|

S7B213E58|meter
7.src=RAD#7.num=208#7.nam=+13.8A#7.low=10.5#7.hi=15.0#7.desc=Main radio
input voltage before fuse#7.unit=Volts#7.fps=0#

S7B213E58|meter
8.src=RAD#8.num=210#8.nam=+13.8B#8.low=10.5#8.hi=15.0#8.desc=Main radio
input voltage after fuse#8.unit=Volts#8.fps=0#

S7B213E58|meter 9.src=TX-
#9.num=1#9.nam=FWDPWR#9.low=0.0#9.hi=53.0#9.desc=RF Power
Forward#9.unit=dBm#9.fps=20#

S7B213E58|meter 10.src=TX-
#10.num=2#10.nam=REFPWR#10.low=0.0#10.hi=53.0#10.desc=RF Power
Reflected#10.unit=dBm#10.fps=20#

S7B213E58|meter 11.src=TX-
#11.num=3#11.nam=SWR#11.low=1.0#11.hi=999.0#11.desc=RF
SWR#11.unit=SWR#11.fps=20#

S7B213E58|meter 12.src=TX-
#12.num=4#12.nam=PATEMP#12.low=0.0#12.hi=100.0#12.desc=PA
Temperature#12.unit=degC#12.fps=0#

S7B213E58|meter 14.src=SLC#14.num=0#14.nam=LEVEL#14.low=-
150.0#14.hi=20.0#14.desc=Signal strength of signals in the filter
passband#14.unit=dBm#14.fps=10#
```

Figure 5

The meter manifest isn't the easiest of things to parse reliably but it does contain everything we need. Our objective is to extract the manifest into an array of meter definitions. Then, when we get streaming data, say, from meter 14, we can look up meter 14 in the manifest and determine that it is the S-meter value. Let's look at the S-meter manifest record in detail:

```
S7B213E58|meter 14.src=SLC#14.num=0#14.nam=LEVEL#14.low=-
150.0#14.hi=20.0#14.desc=Signal strength of signals in the filter
passband#14.unit=dBm#14.fps=10#
```

Immediately after the status message flag and handle, we see the word meter, which tells us that this is a meter manifest message. Unlike virtually all other messages, # is used as a field delimiter rather than a space character. This is probably so that spaces can be used in the meter descriptions. Therefore, the first field is **14.src=SLC**

The number 14 tells us the meter number and **src** is the code for Source and **SLC** is short for Slice. So this is Meter #14 and it is a Slice meter.

The next field, **14.num=0** tells us that this meter belongs to Slice 0.

14.nam=LEVEL indicates that this is a Level meter and later on **14.desc=Signal strength of signals in the filter passband** further advises us what it does: in layman's terms it is the S-meter.

Finally we have some fields that tell us the range and units of the meter:

14.unit=dBm	tells us the meter is scaled in dBm
14.low=-150.0	tells us the minimum value is 150.0dBm
14.hi=20.0	tells us the maximum value is +20dBm
14.fps=10	tells us that the "frames per second" or refresh rate is 10 times/second.

We need to extract each of these fields into a meter array or structure (depending on your language). When meter data arrives on the streaming UDP port we can look the meter up in the meter array to determine its function and characteristics.

Looking at the other meter manifest records you can see similar details for each. Once you've worked out how to parse the S-meter manifest you've pretty well cracked all the meter manifests.

Meter streaming data

Now we have the meter manifest safely tucked away, we can start on the meter streaming data. Meter streaming data have a Stream Identifier of 0x00 00 70 00 and a Class Identifier of 0x00 00 1C 2D 53 4C 80 02. I use the class identifier to differentiate between meter and discovery packets but it would appear that you could also use the stream identifier, as that too is different.

These are hexadecimal data, starting from byte 28 of the VITA49 packet, as follows:

```
00 01 DD C0 00 02 DA 07 00 04 83 00 00 09 00 00 00 0A 00 00 00 0B 00 80
00 0E D1 E9 00 0F FA 2C
```

Figure 6

Each meter is represented by four bytes of data. The first two bytes are the meter number and the second two bytes are the value as a 16-bit signed (twos complement) number. Rearranging the above data into two 16-bit fields we get an array like this:

```
0001 DDC0
0002 DA07
0004 8300
0009 0000
000A 0000
000B 0080
000E D1E9
000F FA2C
```

It is easy to see now that this packet contains meter updates for meters 1, 2, 4, 9, 10, 11, 14 and 15 (decimal). From our manifest we know that meter 14 is our Slice-0 S-meter and that it is calibrates in dBm. With this information we can now analyse the meter data.

The value is 0xD1E9. The most significant bit is the sign bit and that is set, so this is a negative number. The twos complement value is therefore -11799 decimal – a somewhat odd number!

Now we have to introduce another concept: value scaling. dBm meters, of which the S-meter is one example, are value scaled by 128.0 to permit greater granularity in the numbers that can be represented. So our meter reading is in fact:

-11799 /128.0 = -92.18dBm or about S5.

Picking another couple of meters, we know from our meter manifest that meter 9 is our TX forward power and meter 10 is TX reverse power. As we are on receive it is something of a relief to find by inspection that both these meters are indicating zero!

Meter data scaling

I mentioned earlier that meters that are measuring dBm are scaled by a factor of 128. Other meters have different scaling factors. Here's the (I think) complete list

Meter type	Scaling factor
dB(m) (S, power, etc.)	128.0
SWR	128.0
Temperature	64.0
Voltage & current	1024.0
All others	1.0 (no scaling)

Figure 7

Using the data

We now, finally(!) have everything we need to display our meter. As we extract meter 14 from our streaming data, we look up meter 14 in the manifest and find that it is the S-meter for Slice 0. From the manifest we also know how to scale it (divide by 128) and how the meter data is represented (in dBm). Armed with all that information, we are now in a position to display the meter however we see fit in our application.

Closing the API connection

Ideally your program should gracefully disconnect the TCP/IP port as it closes. No great harm will come to pass if you don't but it is good programming practice. The radio will automatically stop sending streaming data to your UDP port and reset any subscriptions that you might have started. In other words, the radio/API deals with client close down gracefully.

Depending on your application environment you may need to un-assign or otherwise dispose of the client interfaces in order for your application to close down cleanly. This is very development environment dependent but in Delphi I do it like this:

```
procedure TTCP.OnTCPDisconnect(Sender: TObject);
begin
FlexDiscovery[FlexControl.RadioInUse].IPConnected:=false;
FlexControl.Connected:=false;
FlexEventControl.FTCPDisconnected(FlexControl.RadioSerialNumber);
FlexTCPClient:=nil;
FlexTCPClient.Free;
if FlexUDPData.DiscoveryPortActive then
  UDP.StopDiscover;
if FlexUDPData.Vita49PortActive then
  UDP.StopVITA49UDP;
end;
```

This 'nils' and then 'frees' the TCP client before closing and doing the same to the UDP ports, if they exist. The program then terminates cleanly.

VITA49 information

From http://www.pentek.com/tutorials/17_2/VITA.cfm:

The VITA Radio Transport (VRT) protocol is a standard for Software Defined Radio (SDR) applications. It was developed to provide interoperability between diverse SDR components by defining a transport protocol to convey digitized signal data and receiver settings.

Information seems to be somewhat sparse on the Internet unless you want to fork out $100 to buy the specification from Vita, see http://shop.vita.com/ANSI-VITA-490-2015-VITA-Radio-Transport-VRT-Standard-AV490.htm. For our purposes we don't need to know anything like that much about the protocol but I can imagine that the copy at FlexRadio's engineering department will be well thumbed!

I found a draft specification on the Internet, which is probably near enough for our purposes at www.61ic.com/code/attachment.php?aid=50954&k...t=1298364429. Don't worry that the file names are in Kanji characters! The text itself is a PDF in English.

The UDP port will send the full VITA49 packet to us. This packet has two parts:

- Header (always exactly 28 bytes and a fixed structure)
- Payload (variable length and structure)

Header

Most of the payload header is of no interest to us. Here's what it looks like:

<table>
<tr><th></th><th>Byte 0</th><th>Byte 1</th><th>Byte 2</th><th>Byte 3</th></tr>
<tr><td>Word 0</td><td colspan="2">Header setup</td><td>Count</td><td colspan="2">Payload size in words</td></tr>
<tr><td>Word 1</td><td colspan="5">Stream Identifier (4 bytes)</td></tr>
<tr><td>Word 2</td><td colspan="5" rowspan="2">Class Identifier (8 bytes)</td></tr>
<tr><td>Word 3</td></tr>
</table>

We can ignore the header setup – it just contains VITA49 settings. The Count field, bits 0..3 of byte 1 contain an incrementing packet number, Count, which increments within each Class Identifier type. This could be used to detect that you have missed packets. I didn't bother (yet).

The payload size is in words. Multiply by 4 to get to bytes. This value excludes the 28 byte header.

The stream identifier is different for differing types of streaming data. These are what I've found so far:

0x00 00 08 00 for Discovery packets
0x00 00 07 00 for Meter packets

The class identifier is also different

0x00 00 1C 2D 53 4C FF FF for Discovery packets
0x00 00 1C 2D 53 4C 80 02 for Meter packets
0x00 00 1C 2D 53 4C 80 03 for FFT packets
0x00 00 1C 2D 53 4C 80 04 for Waterfall packets
0x00 00 1C 2D 53 4C 80 05 for Opus (audio) packets

This information was gleaned from the FlexLib code VitaFlex.cs. I've not yet found out what, if anything, 0x00 00 1C 2D 53 4C 80 01 might be.

Payload

FlexRadio uses the VITA49 extension Packet protocol, which has user defined payload structures. The Discovery protocol payload is actually more similar to the TCP stream data but it is still wrapped in the VITA49 envelope.

The payload always starts at byte 28 (the start of word 4, numbering from 0). As far as I can tell the content is user defined. The payload is always an integer number of words in length (on a 4-byte boundary), with 0x00 padding at the end of the data as required.

For the Discovery protocol, the payload is merely a series of Field_name=field_value pairs, separated by spaces.

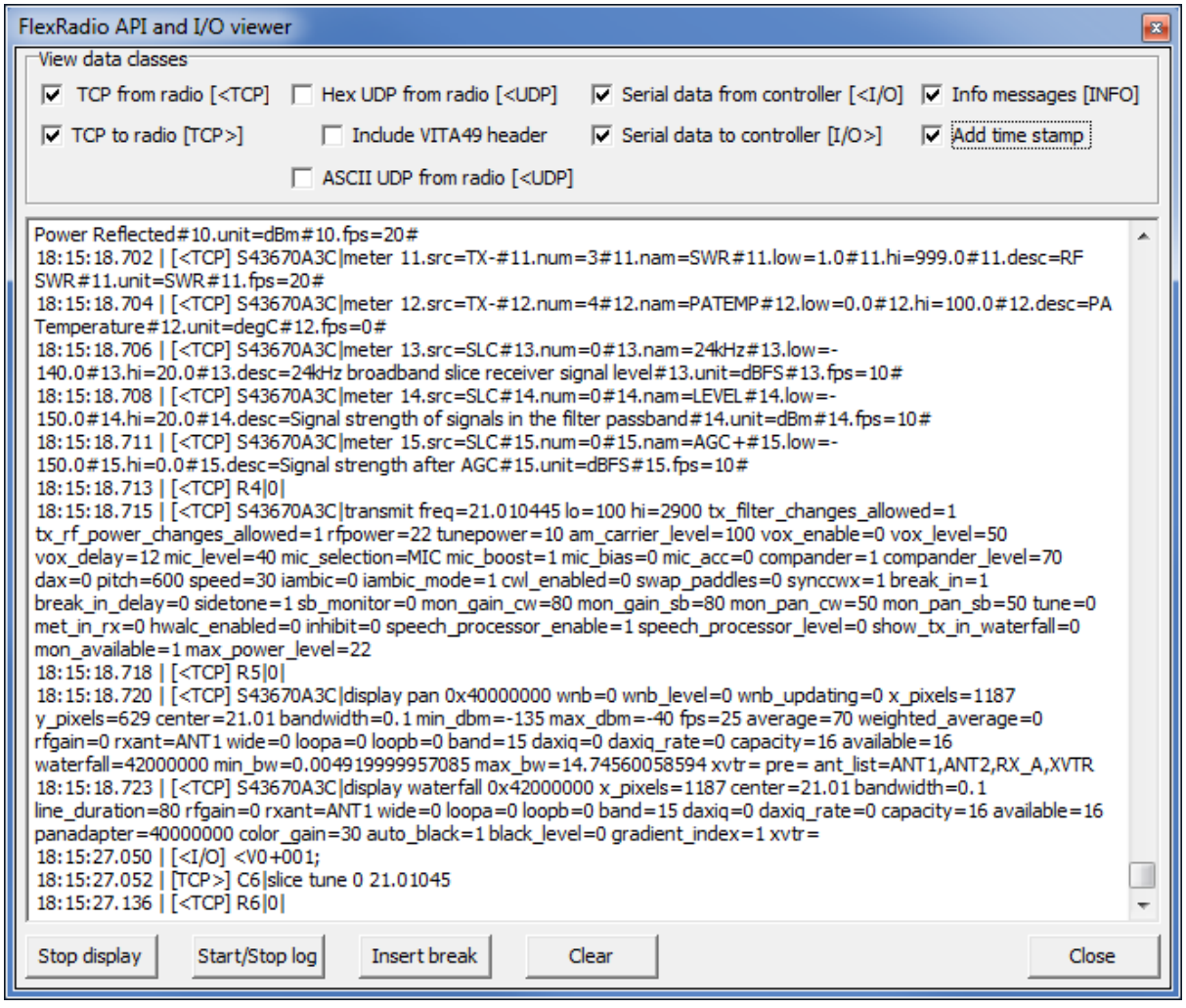
The Meter Extension Packet protocol comprises a variable number of Meter number/Meter value pairs, each occupying four bytes (one word). This is more fully described in the section on streaming data.

Thoughts on writing an API interface

At first it seems quite daunting just understanding what you need to do to get started. In fact it's not that difficult and for the most part the API protocol is consistent, so once you've cracked the basic principles they apply pretty well throughout the API.

The first challenge is to get so you can receive discovery messages. UDP is a pretty simple protocol and it should be straightforward to write some basic code that just outputs the discovery message to a simple display or even to a file that you can open with Notepad. Now you can set about writing the code to extract first a single field and then a more generic function that will extract any number of fields. That's more or less all you have to do with Discovery.

Next up is the TCP/IP connection. Again, it is really worthwhile writing your code so it just outputs whatever the API sends to you. Even now, I find the API Viewer I wrote back at the beginning of this odyssey immensely useful for debugging. Here's a screenshot of the API in full flood:



With this I can see all the commands I've sent to the API and everything that the API has sent back to me. It has enabled me to find numerous bugs and quite a few undocumented features.

Finally there is streaming data to work with. This is quite a lot more difficult and I haven't really done much with it so far. I'm just using the meter data - I catch S-meter, forward TX power, reverse power, SWR, PA temperature and supply voltages either side of the fuse.

“Here be dragons territory” for me includes the audio and panadapter streaming data. I’ve not even looked at these data streams yet, so I can offer no advice at all. I am increasingly of the opinion that I don’t need to anyway, because SmartSDR for Windows can do all those things and as far as I can see, my controller will always operate in conjunction with SmartSDR for Windows or possibly SmartSDR for iOS. Those already do a mighty fine job of the graphical stuff and handling audio feeds, etc. so it’s a problem I don’t need to fix.

No, where the API shines for me is in the facility it provides to make my own hardware controller, with lovely tuning knobs and real push button switches and proper control knobs, just like a “real” radio.

I encourage you to experiment. Get the discovery protocol sorted out and you’ve already learned most of what you need to build a full controller. It’s great fun, a proper construction project for today’s radio amateur and immensely satisfying to be able to use the controller you designed and built to work the DX!

Good luck!

Reference

The following on line references may be of assistance to budding API coders:

http://wiki.flexradio.com

This is the official FlexRadio Wiki and it contains, inter alia, an incomplete and in some cases out of date API specification. There is useful information here but the reader is cautioned to cross check elsewhere.

http://www.flexradio.com/support/downloads/

In the FLEX Signature Series SDRs (FLEX-6000 Family & Maestro) folder you'll find FlexLib API, which is an API middleware product written in C. The file includes all the source and as it is maintained and updated by FlexRadio it is ipso facto the most accurate documentation for the API.

http://g3wgvflex.blogspot.co.uk/

My Blog, in which I document my own development work on controllers that use the FlexRadio API. Here you will also find the open source code for my Mk I controller (written in C for the Arduino). It is my intention to also publish code for my Delphi API middleware and Mk II controller projects here in due course.