Aprx 1.96 Manual

Table of Contents

By Matti Aarnio, OH2MQK, 2009

Version 0.02

1 What is APRX?

The Aprx program is for amateur radio APRS™ networking.

The Aprx program can do job of at least two separate programs:

1. APRS iGate
2. APRS Digipeater

The program has ability to sit on a limited memory system, it is routinely run on OpenWRT machines with 8 MB of RAM and Linux kernel. 128 MB RAM small PC is quite enough for this program with 100 MB ram disk, web-server, etc.

The program is happy to run on any POSIX compatible platform, a number of UNIXes have been verified to work, Windows needs some support code to work.

On Linux platform the system supports also Linux kernel AX.25 devices.

This program will also report telemetry statistics on every interface it has. This can be used to estimate radio channel loading, and in general to monitor system and network health.

2 Configuration Examples:

2 Minimal configuration of Rx-iGate

To make a receive-only iGate, you need simply to configure:

1. mycall parameter
2. APRSIS network connection
3. Interface for the radio

```
mycall  N0CALL-1

<aprsis>
     server    rotate.aprs.net    14580
</aprsis>

<interface>
     serial-device   /dev/ttyUSB0  19200 8n1    KISS
</interface>
```

You need to fix the "*N0CALL-1*" callsign with whatever you want it to report receiving packets with (it must be unique in global APRSIS network!)

You will also need to fix the interface device with your serial port, network TCP stream server, or Linux AX.25 device. Details further below.

In usual case of single radio TNC interface, this is all that a receive-only APRS iGate will need.

3 Minimal configuration APRS Digipeater

To make a single interface digipeater, you will need:

1. mycall parameter
2. <interface> definition
3. <digipeater> definition

Additional bits over the Rx-iGate are highlighted below:

```
mycall  N0CALL-1

<interface>
     serial-device /dev/ttyUSB0  19200 8n1    KISS
     tx-ok          true
</interface>

<digipeater>
     transmit $mycall
     <source>
          source  $mycall
     </source>
</digipeater>
```

The interface must be configured for transmit mode (default mode is receive-only)

Defining a digipeater is fairly simple as shown.

4 Combined APRS Digipeater and Rx-iGate

Constructing a combined APRS Digipeater and Rx-iGate means combining previously shown configurations:

```
mycall  N0CALL-1

<aprsis>
     server    rotate.aprs.net    14580
</aprsis>

<interface>
     serial-device /dev/ttyUSB0  19200 8n1    KISS
     tx-ok         true
</interface>

<digipeater>
     transmit $mycall
     <source>
          source  $mycall
     </source>
</digipeater>
```

5 Doing Transmit-iGate

At the time of the writing, doing Tx-iGate is still lacking some bits necessary for full function.

```
mycall  N0CALL-1

<aprsis>
     server    rotate.aprs.net    14580
</aprsis>

<interface>
     serial-device /dev/ttyUSB0  19200 8n1    KISS
     tx-ok         true
</interface>

<digipeater>
     transmit $mycall
     <source>
          source         APRSIS
          digi-mode      3rd-party
          filter         t/m
     </source>
</digipeater>
```

This is Rx/Tx-iGate in a form that Aprx version 1.96 can do.

It does omit couple important bits on controlling transmission from APRSIS to radio, and thus a kludge definition of filtering “pass only type M packets” (APRS Messages).

6 Digipeater and Transmit-iGate

This is fairly simple extension, but shows important aspect of Aprx's <digipeater> definitions, namely that there can be multiple sources!

```
mycall  N0CALL-1

<aprsis>
     server    rotate.aprs.net    14580
</aprsis>

<interface>
     serial-device /dev/ttyUSB0  19200 8n1    KISS
     tx-ok         true
</interface>

<digipeater>
     transmit $mycall
     <source>
          source  $mycall
     </source>
     <source>
          source         APRSIS
          digi-mode      3rd-party
          filter         t/m
     </source>
</digipeater>
```

Using both the radio port, and APRSIS as sources makes this combined Tx-iGate, and digi-peater.

7 A Fill-in Digipeater

Classically a fill-in digipeater means a system that digipeats heard packet only when it hears it as from first transmission. Usually implemented as “consider WIDE1-1 as your alias”, but the Aprx has more profound understanding of digipeating.

```
<digipeater>
     transmit $mycall
     <source>
          source  $mycall
          relay-type directonly
     </source>
</digipeater>
```

With Aprx you can add condition: *and only if nobody else digipeats it within 5 seconds.*

```
<digipeater>
     transmit $mycall
     <source>
          source        $mycall
          relay-type    directonly
          viscous-delay 5
     </source>
</digipeater>
```

8 Using multiple radios

There is no fixed limit on number of radio interfaces that you can use, however of them only one can use the default callsign from “$mycall” macro:

```
mycall  N0CALL-1

<interface>
     # callsign  $mycall
     serial-device /dev/ttyUSB0  19200 8n1    KISS
     tx-ok         true
</interface>

<interface>
     callsign  N0CALL-R2
     serial-device /dev/ttyUSB1  19200 8n1    KISS
</interface>
```

Supported interface devices include:

1. On Linux: Any AX.25 network attached devices
2. On any POSIX system: any serial ports available through “tty” interfaces
3. Remote network terminal server serial ports over TCP/IP networking

On serial ports, following protocols can be used:

1. Plain basic **KISS**: Binary transparent, decently quick.
2. **SMACK**: A CRC16 two-byte CRC checksum on serial port KISS communication. Recommended mode for KISS operation.
3. XOR checksum on KISS: So called “**BPQCRC**”. Not recommended because it is unable to really detect data that has broken during serial port transmission. Slightly better than plain basic KISS.
4. **TNC2** monitoring format, receive only, often transmitted bytes outside printable ASCII range of characters are replaced with space, or with a dot. **Not recommended to be used!**

9 A Digipeater with Multiple Radios

Extending on previous multiple interface example, here those multiple interfaces are used on a digipeater. Transmitter interface is at “$mycall” label, others are receive only:

```
<digipeater>
     transmit $mycall
     <source>
          source  $mycall
     </source>
     <source>
          source  N0CALL-2
     </source>
     <source>
          source  N0CALL-3
     </source>
</digipeater>
```

Adding there a source of APRSIS will merge in Tx-iGate function, as shown before.

It is trivial to make a multiple receiver, single transmitter APRS Digipeater with this.

10 A Bi-directional Cross-band Digipeater

Presuming having transmit capable radio <interface>s on two different bands, you can construct a bi-directional digipeater by defining two <digipeater> sections.

```
<digipeater>
     transmit N0CALL-1
     <source>
          source  N0CALL-1
     </source>
     <source>
          source  N0CALL-2
     </source>
</digipeater>

<digipeater>
     transmit N0CALL-2
     <source>
          source  N0CALL-1
     </source>
     <source>
          source  N0CALL-2
     </source>
</digipeater>
```

Now both transmitters will digipeat messages heard from either radio.

You will probably want more control parameters to limit on how much traffic is relayed from one source to other, more of that in the detail documentation.

11 Limited Service Area Digipeater

A digipeater that will relay only packets from positions in a limited service area can be done by using filtering rules:

```
<digipeater>
     transmit N0CALL-1
     <source>
          source  N0CALL-1
          filter    t/m       # All messages (position-less)
          filter    a/60/23/59/25.20
          filter    a/60.25/25.19/59/27
     </source>
</digipeater>
```

This example is taken from a limited service area digipeater on a very high tower in Helsinki, Finland. The coordinates cover Gulf of Finland, and northern Estonia. Especially it was not wanted to relay traffic from land-areas, but give excellent coverage to sail yacts.

3 Configuration in details

The Aprx configuration file uses sectioning style familiar from Apache HTTPD.

These sections are:

1. mycall
2. <aprsis>
3. <logging>
4. <interface>
5. <digipeater>

Each section contains one or more of configuration entries with case depending type of parameters.

1 Aprx Configuration Parameter Types

The Aprx configuration has following types of parameters on configuration entries:

- Parameters can be without quotes, when such are not necessary to embed spaces, or to have arbitrary binary content.
- Any parameter can be quoted by single or double quotes: `“ .. “  ' .. '`
- Any quoted parameter can contain \-escaped codes. Arbitrary binary bytes are encodable as `“\xHH”`, where “`HH`” present two hex-decimal characters from `“\x00”` to `“\xFF”`. Also quotes and backslash can be backslash-escaped: `“\””  “\\”`
- Arbitrary binary parameter content is usable only where especially mentioned, otherwise at least `“\x00”` is forbidden.
- UTF-8 characters are usable in parameters with and without quotes.
- Callsign definitions (see below)
- Interval definitions (see below)

The *interval-definition* is convenience method to give amount of time in other units, than integer number of seconds. An *interval-definition* contains series of decimal numbers followed by a multiplier character possibly followed by more of same. Examples:

```
    2m2s
    1h
```

The multiplier characters are:

1. s (S): Seconds, the default
2. m (M): Minutes
3. h (H): Hours
4. d (D): Days
5. w (W): Weeks

The *callsign* parameters are up to 6 alphanumeric characters followed by optional minus sign (“-”, the “hyphen”) and optional one or two alphanumeric characters. Callsigns are internally converted to all upper case form on devices. Depending on usage locations, the “SSID” suffix may be up to two alphanumeric characters, or just plain integer from 0 to 15. That latter applies when a strict conformance to AX.25 callsigns is required. Callsign parameter with suffix “-0” is canonicalized to a string without the “-0” suffix.

2 The “mycall” Parameter

The *mycall* entry is just one global definition to help default configuration to be minimalistic by not needing copying your callsign all over the place in the usual case of single radio interface setup.

3 The “<aprsis>” section

The <aprsis> section defines communication parameters towards the APRSIS network.

When you define <aprsis> section, all configured <interface>s will be Rx-iGate:d to APRSIS! Thus you can trivially add an Rx-iGate to a <digipeater> system, or to make a Rx-iGate without defining any <digipeater>.

The only required parameter is the server definition:

```
    server    rotate.aprs.net   14580
```

where the port-number defaults to 14580, and can be omitted.

Additional optional parameters are:

- login *callsign*
- heartbeat-timeout *interval-definition*
- filter *adjunct-filter-entry*

The *login* defaults to global *$mycall*, thus it is not necessary to define.

Adding “*heartbeat-timeout 2m*” will detect failure to communicate with APRSIS a bit quicker than without it. The current generation of APRSIS servers writes a heartbeat message every 20 seconds, and a two minute time-out on their waiting is more than enough.

The “*filter ...*” entries are concatenated, and given to APRSIS server as adjunct filter definitions. For more information about their syntax, see:

`http://www.aprs-is.net/javAPRSFilter.aspx`

4 The “<logging>” section

The Aprx can log every kind of event happening, mainly you will be interested in *rflog*, and *aprxlog*.

There is also a possibility to store statistics gathering memory segment on a filesystem backing store, so that it can persist over restart of the Aprx process. This is possible even on a small embedded machine (like OpenWRT), where statistics “file” resides on a ram-disk. This way you can alter configurations and restart the process, while still continuing with previous statistics dataset. Without the backing store this will cause at most 20 minute drop-off of statistics telemetry data.

Configuration options are:

- aprxlog *filename*
- rflog *filename*
- pidfile *filename*
- erlangfile *filename*
- erlang-loglevel *loglevel*
- erlanglog *filename*

Commonly you want setting aprxlog, and rflog entries. The erlangfile, and pidfile entries have compile time defaults, and need not to be defined unless different locations are wanted.

5 The “<interface>” sections

The <interface> sections define radio interfaces that the Aprx communicates with.

There are three basic interface device types:

1. Linux AX.25 devices (ax25-device)
2. Generic POSIX serial ports (serial-device)
3. Remote network serial ports (tcp-device)

The serial port devices can be reading TNC2 style monitoring messages (and be unable to transmit anything), or communicate with a few variations of KISS protocol (and transmit). On KISS protocols you can use device multiplexing, although cases needing polling for reception are not supported. Variations of KISS protocol are described separately.

On Linux systems the kernel AX.25 network devices are also available, and Aprx integrates fully with kernel AX.25 networking.

Each interface needs a unique callsign and to help the most common case of single radio interface, it defaults to one defined with *mycall* entry. The interface callsigns need not to be proper AX.25 callsigns on receive-only serial/tcp-device interfaces, meaning that a *N0CALL-R0 .. R9 .. RA .. RZ* are fine examples of two character suffixes usable on such receivers.

As there are three different devices, there are three different way to make an <interface> section.

Linux AX25-DEVICE:

```
    <interface>
        ax25-device    callsign
        tx-ok          boolean
        alias          RELAY,TRACE,WIDE
    </interface>
```

The *callsign* parameter must be valid AX.25 callsign as it refers to Linux kernel AX.25 device callsigns. Such Linux kernel device does not need to be active at the time the Aprx program is started, the Aprx attaches itself on it dynamically when it appears, and detaches when it disappears.

The interface *alias* entry can be defined as comma-separated lists of AX.25 callsigns, or as multiple *alias* entries. Default set is above shown *RELAY,TRACE,WIDE*. If you define any *alias* entry, the default set is replaced with your definitions.

POSIX serial-port devices, KISS mode, sub-interface 0:

```
    <interface>
        serial-device  devicepath speed KISS
        tx-ok          boolean
        callsign       callsign
        initstring     "init-string-content"
        timeout        interval-definition
        alias          RELAY,TRACE,WIDE
    </interface>
```

You can use a binary-transparent AX.25 radio modem on a KISS type connection. The above example shows case of KISS modem on sub-interface 0.

The *tx-ok* entry (default value: “false”) controls whether or not the interface is capable to transmit something.

The *callsign* entry defines system wide unique identity for the radio port, and for transmit capable interfaces it must be valid AX.25 callsign form.

Interface *alias* entry can be issued as comma-separated lists of AX.25 callsigns, or as multiple *alias* entries. Default set is above shown *RELAY,TRACE,WIDE*. If you define any *alias* entry, the default set is replaced with your definitions.

POSIX serial-port devices, KISS mode, multiple sub-interfaces:

```
<interface>
     serial-device  devicepath speed KISS
     initstring     "init-string-content"
     timeout        interval-definition
     <kiss-subif 0>
          tx-ok          boolean
          callsign       callsign
          alias          RELAY,TRACE,WIDE
     </kiss-subif>
     <kiss-subif 1>
          tx-ok          boolean
          callsign       callsign
          alias          RELAY,TRACE,WIDE
     </kiss-subif>
</interface>
```

You can use a binary-transparent AX.25 radio modem on a KISS type connection. The above example shows case of KISS modem on sub-interface 0.

You can use the *initstring* to issue a binary byte stream to the serial port to initialize the radio modem, if necessary.

You can set a *timeout* parameter to close and reopen the device with optional initstring sending, which will happen if there is *interval-definition* amount of time from last received data on the serial port. Suitable amount of time depends on your local network channel, somewhere busy a 5 minutes is quite enough (“5m”), elsewhere one hour may not be enough (“60m”).

The *<kiss-subif N>* sectioning tags have N in range of 0 to 7 or 0 to 15, depending on KISS protocol variation used on this serial-device. More about that latter. On each <kiss-subif N> sub-sections you can use:

- The *tx-ok* entry (default value: “false”) to control whether or not the sub-interface is capable to transmit something.
- The *callsign* entry to give uniquelidentity for the sub-interface. For transmit capable sub-interfaces it must be of valid AX.25 callsign form.
- The sub-interface *alias* entry can be issued as comma-separated lists of AX.25 callsigns, or as multiple *alias* entries. Default set is above shown *RELAY,TRACE,WIDE*. If you define any *alias* entry, the default set is replaced with your definitions.

The KISS variations:

The Aprx knows three variations of basic Chepponis/Karn KISS protocol, listed below in preferrence order:

1. **S**tuttgart **M**odified **A**mateurradio-**C**RC-**K**ISS (alias **SMACK**)
2. **BPQCRC**
3. Plain basic **KISS**

The **SMACK** uses one bit of CMD byte to indicate that it is indeed SMACK format of KISS frame. The bit in question is highest bit, which is highest sub-interface identity bit. Thus SMACK is not able to refer to sub-interfaces 8 to 15 of original KISS protocol. On the other hand, hardly anybody needs that many! It uses CCITT-CRC16 algorithm, and is capable to detect loss or insert of single bytes in frame as well as single and sometimes also multiple bit flips in correct number of bytes within the frame.

The **BPQCRC** alias **XKISS** uses single byte containing XOR of all bytes within the data frame (before the KISS frame encoding is applied/after it is taken off.) This is very weak checksum, as it does not detect addition/removal of 0x00 bytes at all, and is unable to detect flipping of same bit twice within the frame.

The plain basic **KISS** is adaptation of internet SLIP protocol, and has no checksum of any kind in the framing interface.

If at all possible, do choose to use SMACK!

POSIX serial-port devices, TNC2 mode:

```
     <interface>
          serial-device  devicepath speed TNC2
          callsign       callsign
          timeout        interval-definition
          initstring     “init-string-content”
     </interface>
```

If you absolutely positively must have a TNC2 monitoring mode radio modem, then it can be used for passive monitoring of heard APRS packets, but beware that such radio modems usually also corrupt some of heard APRS packets, and that this type of interface is not available for transmit mode. Only mandatory entry is “serial-device”, others have usable defaults.

The *callsign* entry defines unique identity for the radio port, but it need not to be valid AX.25 callsign.

You can set a *timeout* parameter to close and reopen the device with optional initstring sending, which will happen if there is *interval-definition* amount of time from last received data on the serial port. Suitable amount of time depends on your local network channel, somewhere busy a 5 minutes is quite enough (“5m”), elsewhere one hour may not be enough (“60m”).

You can use the *initstring* to issue a binary byte stream to the serial port to initialize the radio modem, if necessary.

6 The “<beacon>” sections

You can define multiple <beacon> sections each defining multiple beacon entries.

Beacons can be sent to radio only, to aprsis only, or to both. Default is to both.

You can configure beacons as literals, and also to load beacon content from a file at each time it is to be transmitted. That latter allows external program, like weather probes, to feed in an APRS weather data packet without it needing to communicate with Aprx via any special protocols, nor make AX.25 frames itself.

MORE TO BE WRITTEN

7 The “<digipeater>” sections

With Aprx you can define multiple <digipeater> sections, each to their unique transmitter.

At each <digipeater> section you can define multiple <source> sub-sections so that traffic from multiple sources are sent out with single transmitter.

This allows defining different behaviour rules per each transmission path.

The Aprx implements duplicate checking per each transmitter, and if same message is received via multiple (diversity) receivers, only one copy will be transmitted.

MORE TO BE WRITTEN