Aprx 2.01 Manual

A special-purpose Ham Radio software
for
UNIX-like environments, including Linux

Manual version 1.01

Aprx 2.01 Manual, version 1.01

By Matti Aarnio, OH2MQK, 2009-2010

Table of Contents

1 What is APRX?

The Aprx program is for amateur radio APRS™ networking.

The Aprx program is available at:

http://ham.zmailer.org/oh2mqk/aprx/

Discussion forum is at:

http://groups.google.com/group/aprx-software

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

1. APRS iGate
2. APRS Digipeater
3. DPRS-to-APRS gateway

The digipeater functionality can also be used for other forms of AX.25 networking, should a need arise.

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 64 MB ram disk, a web-server, and much much more.

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 seamlessly the 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.

The telemetry data is viewable via APRSIS based services, like `http://aprs.fi`

The Aprx program has been written is maintained by Matti Aarnio, OH2MQK.

Reachable for example at: oh2mqk (at) sral.fi

This page is intentionally left blank

2 Configuration Examples

2.1 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.aprs2.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.

You need to look at http://www.aprs2.net/ for possible other suitable servers to use. The "rotate.aprs2.net" uses global pool of servers, however some regional pool might be better suited – for example: noam.aprs2.net, euro.aprs2.net, etc.

In most common case of wanting to make an i-gate, you do not want to use APRSIS service port parameter at all! *Using "all traffic" port of 10152, or anything else than "user port" 14580 is bound to cause all sorts of troubles.*

2.2 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>
     transmitter $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.

> ***Do not put* "alias wide1-1" *or any other of that type on the <interface> definitions! Aprx does understand the "new-n-paradigm" digipeater routing in fundamental level, and does not need kludges that old AX.25 digipeater systems did need.***

The digipeater handles AX.25 UI frames with APRS packet types using APRS rules, including duplicate detection, "new-n-paradigm" processing, etc.

At the same time it also handles all kinds of AX.25 frames as basic AX.25 digipeater matching next-hop with interface callsigns and aliases (see chapter 3.5 "The "<interface>" sections")

2.3 Filtering APRS Digipeater

To make a single interface digipeater, you will need:

4. mycall parameter
5. <interface> definition
6. <digipeater> definition

Additional bits over the minimal digipeater are highlighted below:

```
mycall  N0CALL-1

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

<digipeater>
     transmitter $mycall
     <source>
          source     $mycall
          filter     p/prfx   # additive filters
          filter    -p/prfx   # substractive filters
     </source>
</digipeater>
```

Without any *filter* parameters, the system will not inspect packet content to determine if it should be digipeated or not. It will digipeat everything.

When *filter* parameters are defined, there are two kinds:

- Additive: Packet matching on this may be eligible for digipeat
- Subtractive: Packet matching on this will be rejected even if it matched on Additive.

There can be unlimited number of both kinds of filters.

Note: These filters are applied only on APRS type packets.

2.4 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>
     transmitter $mycall
     <source>
          source  $mycall
     </source>
</digipeater>
```

It really is as simple as that. When an <aprsis> section is defined, all declared <interface>s are Rx-iGate:d to APRSIS in addition to what else the system is doing.

In most common case of wanting to make an i-gate, you do not want to use APRSIS service port parameter at all! *Using “all traffic” port of 10152, or anything else than “user port” 14580 is bound to cause all sorts of troubles.*

2.5 Doing Transmit-iGate

The APRSIS source on a digipeater will enable APRSIS -> RF iGate functionality.

Mandatory parts are “source APRSIS”, and “relay-type 3rd-party”.

Everything else is optional.

```
mycall  N0CALL-1

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

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

<digipeater>
     transmitter $mycall
     <source>
          source         APRSIS
          relay-type     3rd-party
          via-path       WIDE1-1   # default: none
          # viscous-delay 5        # prefer RF path
          # filter       p/prfx    # additive packet filters
          # filter        -p/prfx  # substractive packet fltrs
     </source>
</digipeater>
```

Available packet filters are described in detail at 3.8.3.1 Filter entries at page 37.

Use of “filter” statements has two variations:

- Additive: These are in addition to what Tx-iGate would relay.
- Substractive: These reject what Tx-iGate would relay.

There can be unlimited number of both kinds of filters.

The algorithm above has subtle difference of what non-APRSIS source filters do!

The APRSIS server port **must** be 14580 (user port). Using anything else is likely to create all kinds of “mysterious troubles.”

2.6 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>
     transmitter $mycall
     <source>
          source  $mycall
     </source>
     <source>
          source         APRSIS
          relay-type     3rd-party
          viscous-delay  5
     </source>
</digipeater>
```

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

Again, the APRSIS server port **must** be 14580 (user port). Using anything else is likely to create all kinds of “mysterious troubles.”

2.7 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 when it hears something as “directly from the source”, and therefore there is no need for kludges like that “alias WIDE1-1”.

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

Do not put* “alias wide1-1” *or any other of that type on the <interface> definitions! Aprx does understand the “new-n-paradigm” digipeater routing in fundamental level, and does not need kludges that old AX.25 digipeater systems did need.

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

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

Note: Use of “viscous-delay” parameter without this being “relay-type directonly” is generally not a good idea!

Use of “viscous-delay” makes better sense when you have multiple radio receivers in your digipeater so that even though digipeaters are sending simultaneously, you will still be able to receive them individually by means of using directional antennas.

A non-zero “viscous-delay” value gets always an additional per packet randomized 0 to 2 second delay. This allows similarly configured Aprx servers to hopefully not transmit all at same time.

2.8 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, all others must have explicite and unique callsign:

```
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. **XKISS** alias **BPQCRC**: XOR checksum on KISS. Not recommended because it is unable to really detect data that has broken during serial port transmission. Slightly better than plain basic KISS.
4. **FLEXNET**: Adds a CRC16 on each serial link packet.
   Algorithm is not CCITT-CRC16.
5. **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!**

The KISS protocol variations support multiplexing radios on single serial port.

2.9 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>
     transmitter $mycall
     <source>
          source  $mycall
     </source>
     <source>
          source  N0CALL-R2
     </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.

The <digipeater> section transmitter has local APRS packet duplicate filter so that receiving same packet from multiple diversity receiver sources sends out only first one of them.

2.10 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>
     transmitter N0CALL-1
     <source>
          source  N0CALL-1
     </source>
     <source>
          source  N0CALL-2
     </source>
</digipeater>

<digipeater>
     transmitter 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.

2.11 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>
     transmitter N0CALL-1
     <source>
          source     N0CALL-1
          relay-type directonly
          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 yachts.

Available filters are described in detail at 3.8.3.1 Filter entries at page 37.

2.12 Sending telemetry to radio interface

The Aprx is always collecting radio interface statistics from all of its interfaces.

It sends that data out to APRS-IS, but in case you do not have APRS-IS connection, like in remote digipeater, it can also be transmitted on radio:

```
     <telemetry>
          transmitter    $mycall
          via            WIDE1-1
          source         $mycall
     </telemetry>
```

You can define multiple <telemetry> sections with different transmitters.

On each <telemetry> section you can define multiple sources.

There is no default “via” path, if something is needed to reach near-by iGates, add some.

2.13 DPRS-to-APRS Gateway

To make a DPRS-to-APRS gateway, all you need is to define an <interface> source for DPRS data streams, and a <digipeater> where that DPRS interface is defined as one of <source>s:

```
<interface>
     serial-device ..... DPRS
     callsign    N0CALL-DG
</interface>

<interface>
     tcp-device 192.168.1.1 9000 DPRS
     callsign    N0CALL-D2
</interface>

<digipeater>
     transmitter N0CALL-1
     <source>
          source     N0CALL-DG
          relay-type third-party
     </source>
     <source>
          source     N0CALL-D2
          relay-type third-party
     </source>
</digipeater>
```

This shows also, how to use so called D-RATS data stream reflector as source for D-PRS data stream.

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. <beacon>
6. <telemetry>
7. <digipeater>

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

3.1 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.

3.2 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)
- Very long parameter lines can be folded on input file by placing a lone `\`-character at the end of the configuration file text line to continue the input line with contents of following line, for unlimited number of times.

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.

3.3 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:

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

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

You need to look at http://www.aprs2.net/ for possible other suitable servers to use. The “rotate.aprs2.net” uses global pool of servers, however some regional pool might be better suited – for example: noam.aprs2.net, euro.aprs2.net, etc.

Use of other APRSIS server port than 14580 will likely do things that you absolutely do not want to happen.

Additional optional parameters are:

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

The *login* defaults to global *$mycall*. Set it only if you login with other value than that stored on “mycall” parameter.

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

NOTE: For an Rx/Tx-iGate you do not need any additional explicit *filter* definitions. The option exists just in case there is a need for it.

3.4 The “<logging>” section

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

Note: The *erlangfile* related bits are dependent on software being compiled using “--with-erlangstorage” option, which is not the default way in Aprx-2 series.

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 want-ed.

3.5 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, however this is done with Linux */etc/ax25/axports* file by only referring on AX.25 callsigns on interfaces.

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 ways to make an <interface> section.

3.5.1 The KISS variations

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

1. **S**tuttgart **M**odified **A**mateur-radio-**C**RC-**K**ISS (**SMACK**)
2. **FLEXNET** KISS
3. **BPQCRC** alias **XKISS**
4. 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 CRC-16 algorithm (not the same as CCITTT-CRC used on AX.25 HDLC frame,) 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 **FLEXNET** is a KISS protocol variation alike that of SMACK, but control byte has slightly different bit meanings. The FLEXNET has its own CRC checksum at the last two bytes of the KISS 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!

It is available for TNC2 clones from:

http://www.symek.com/g/tnc2firmware.html

http://www.symek.com/down/smack.zip

Something else entirely would be 6PACK – which interleaves bulk packet data and low latency real time event data bytes on serial link. The 6PACK is not supported in Aprx itself, but a Linux system can interface with 6PACK TNC, and present it thru AX.25 network.

3.5.2 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. (Do **not** define “WIDE1-1” type kludge aliases that old TNC digipeaters seem to need.)

3.5.3 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, for receive-only ports it can be anything that APRSIS accepts.

The *initstring* is a byte-string to be sent to the kiss devices. You can use this to send initialization values to KISS modems. Difficulty is that you must manually encode here everything, including KISS framing.

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. (Do **not** define “WIDE1-1” type kludge aliases that old TNC digipeaters seem to need.)

The *speed* is one of pre-defined standard baud rate speeds: 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 500000, 576000. Probably not all of them are supported in your platform, but at least the ones up to 38400 should be quite common.

3.5.4 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.

The *initstring* is a byte-string to be sent to the kiss devices. You can use this to send initialization values to KISS modems. Difficulty is that you must manually encode here everything, including KISS framing.

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 on SMACK mode, and 0 to 15 on other KISS modes. 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 unique identity for the sub-interface. For transmit capable sub-interfaces it must be of valid AX.25 callsign form, for receive-only ports it can be anything that APRSIS accepts.
- 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. (Do **not** define "WIDE1-1" type kludge aliases that old TNC digipeaters seem to need.)

3.5.5 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.

3.5.6 POSIX serial-port devices, DPRS mode

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

This enables receive-only DPRS to APRSIS Rx-iGate in case APRSIS connection is defined.

This interface can also be used as a source on <digipeater> to enable DPRS-to-APRS RF gateway.

3.5.7 Networked tcp-stream connected terminal devices

```
<interface>
     tcp-device    hostname-or-ip-address portnumber KISS
.....................
</interface>

<interface>
     tcp-device    hostname-or-ip-address portnumber TNC2
.....................
</interface>

<interface>
     tcp-device    hostname-or-ip-address portnumber DPRS
.....................
</interface>
```

These work identical to local physical serial ports described above.

The *hostname-or-ip-address* and *portnumber* point to remote terminal server, where remote serial port is configured to attach on a radio modem. The connection must be such that no extra bytes are added on the datastream, nor any byte codes are considered command escapes for the terminal server. That is, plain TCP, no TELNET service!

Recommendation is to use IP address literals instead of domain names.

Both IPv4 and IPv6 protocols are supported, where platform supports them.

3.5.8 NULL-DEVICE

A development tool, and by default a “transmitter device”.

```
<interface>
     null-device    callsign
     #tx-ok         false
     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. (Do **not** define “WIDE1-1” type kludge aliases that old TNC digipeaters seem to need.)

3.6 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.

The <beacon> section has following entries:

- cycle-size *interval*
- beaconmode *{ aprsis* | *both* | *radio }*
- beacon ...

The *cycle-size* entry is global setter of beacon transmission cycle. Default value is 20 minutes, but if you want to beacon 3 different beacons on average 10 minutes in between each, then use interval value: 30m

The *beaconmode* setting defaults to *both* at the start of <beacon> section,, and affects *beacon* entries following the setting, until a new setting. The *aprsis* setting will send following beacons only to APRSIS, and the *radio* setting will send following beacons only to radio interface(s).

The *beacon* entry parameters:

- interface *interface-callsign*
- srccall *callsign*
- dstcall *callsign*
- via “*viapath*”
- timefix

... continued ...

The *interface* parameter sets explicit interface on which to send this beacon. If no *to* parameter is given, then the beacon is sent on all interfaces.

The srccall parameter sets beacon packet source callsign, and it gets its default value from transmit interface's callsign. In single tx case you do not need to set this. In multi-tx case you may want to set this.

The *dstcall* parameter sets beacon packet destination callsign, default value is program release version dependent, but possible override could be like: “APRS”.

The *via* parameter adds fields on beacon AX.25 VIA path. Default is none, example value would be like: via “WIDE1-1”

The *timefix* parameter sets a flag to modify transmitted APRS packets that have a time stamp field (two kinds of basic position packets, and objects.) Set this only if your machine runs with good quality NTP time reference.

Then either of following two:

- raw “*APRS packet text*”
- file *filepath*

or a combination of following:

- One of following three (default is type “!”)
  - type “*single-character-type*”
  - item “*item-name*”
  - object “*object-name*”
- lat *latitude*
- lon *longitude*
- symbol “*two-character-symbol*”
- comment “*text*” (optional)

There are three different ways to define beacon data: *file, raw,* and third way is a combination of a number of data parameters. The third way has a benefit of being able to validate packet structure at configuration time.

The *file* parameter is one alternate way to followed by a pathname to local file system at which an other program can place APRS packet content to be sent out at next time it is encountered in the beacon cycle. The Aprx program reads it at beacon time for transmitting.

The *raw* parameter is followed by full raw APRS packet text. The packet data is not validated in any way!

... continued ...

The multi-component packet data content construction is done with following parameters:

The *type* parameter defaults to “!” and can be set to any of: "!", "=", "/", "@".

The *item/object* parameter sets name field for *item* and *object* type packets (“;” and “)”). There is no default value.

The *symbol* parameter sets two character APRS symbol on the beaconed packet. It has no default value.

The *lat* parameter sets (and validates) APRS format latitude coordinate: ddmm.mmX where X is 'S' or 'N' indicating latitude hemisphere. There is no default value.

The *lon* parameter sets (and validates) APRS format longitude coordinate: dddmm.mmX where X is 'E' or 'W' indicating longitude hemisphere. There is no default value.

The *comment* parameter sets tail of the packet where an arbitrary text can appear, you can use UTF-8 characters in there. There is no default value, use of this field is optional.

In order to construct a packet with these multi-component fields, you must use at least parameters: *symbol, lat, lon*.

Some examples:

```
<beacon>
  # Load beacon message content from a file:
  beacon  file /tmp/wxbeacon.txt

  # Define fixed beacon from components:
  beacon  via TRACE1-1 \
          symbol "R&" lat "6016.35N" lon "02506.36E" \
          comment "Aprx v1.97 - a Digi + Rx-iGate"

  # When all else fails, “raw” can be used:
  beacon  raw "!6016.30NR02506.36E&Aprx v1.97 - a Digi + Rx-iGate"

  # Define an OBJECT for a local voice repeater:
  beacon  object "OH2RAY" \
          symbol "/r" lat "6044.09N" lon "02612.79E" \
          comment "434.775MHz TOFF -1600kHz R50k OH2RAY"
</beacon>
```

3.7 The “<telemetry>” sections

The Aprx is always collecting interface statistical data on all of its interfaces.

By default that data is sent out only to APRS-IS. If there is no APRS-IS connectivity, nothing is sent in default case.

Defining a <telemetry> section can let you control alternate transmission to radios:

```
      <telemetry>
            # Select where to transmit
            transmitter $mycall

            # Optional digipeat path for transmission to near by IGates
            via               WIDE1-1

            # Multiple sources permitted
            source            $mycall
            source            N0CALL-2
      </telemetry>
```

The telemetered source ports must have valid AX.25 callsigns. Pure APRS-IS telemetry is far more permissive in this regard.

If you want telemetry to multiple radio transmitters, define multiple <telemetry> sections.

The transmission will have same data content and frequency as APRS-IS transmissions have: Telemetry packet once per 20 minutes, associated label packets every 2 hours, but with 2 minute offset to telemetry data.

Note: If you define multiple sources, all the telemetry records are sent “rapid fire” one after another without any delay! This does not harm APRS-IS, but may cause packet drops on radio transmission.

Telemetry related data labels are sent at different time than data records, but they too cause “rapid fire” clusters on radio – one label per source. Each of 3 kinds of labels are sent out with 2 hour intervals in between them.

Development assumption has been that big multi-receiver systems have also internet connectivity and won't transmit telemetry over radio, or as a compromise they send RF telemetry only regarding digipeater transmitter port(s).

3.8 The “<digipeater>” sections

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

The Aprx will apply full set of controls on APRS type packets, but it will handle also plain AX.25 digipeat of other types of AX.25 packets.

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

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

The Aprx implements also basic AX.25 digipeater for non-APRS frames, where radio interface callsign (and interface aliases) is matched against first AX.25 address header VIA field without its H(as been digipeated)-bit set.

The structure of each <digipeater> section is as follows:

```
    <digipeater>
        transmitter    callsign
        ratelimit      60 120
        <trace>
            ...  (defaults are usually OK)
        </trace>
        <wide>
            ...  (defaults are usually OK)
        </wide>
        <source>
            source    callsign
            ...  (defaults are usually OK)
        </source>
        ...       (more sources can be defined)
    </digipeater>
```

The *transmitter* entry defines callsign of transmitter radio port to be used for this section. There is no default, but macro *$mycall* is available.

The *ratelimit* gives *average* and *upper* limit on number of packets to be relayed per minute out on this transmitter. System tracks number of packets sent per 5 second timeslots using “token bucket filter” algorithm. Default values are 60/120. Maximum values are 300/300. (Default limit saturates 1200 bps AFSK channel, but 9600 bps channel has some slack.)

The *<trace>* sub-section is available both at <digipeater> and at <source> levels. Source specific settings override digipeater-wide settings. More details below. The <trace> settings are looked at before <wide> settings, thus same key value in both

The *<wide>* sub-section is alike <trace> sub-section, more details below.

The *<source>* sub-sections define sources that this digipeater instance receives its packets from. Same source devices can feed packets to multiple digipeaters.

3.8.1 The <*trace*> sub-section

```
<trace>
     maxreq    N    # in range: 1 .. 7, default: 4
     maxdone   N    # in range: 1 .. 7, default: 4
     keys      TRACE,WIDE,RELAY
</trace>
```

The <trace> block settings can be at <digipeater> level, and at <source> sub-level. The <digipeater> level has above listed default values.

The <source> sub-level <trace> instance is used at first to check what to do to packet. If there is no match at <source> sub-level, the <digipeater> level <trace> entry is checked. If neither matched, then <wide> entries (see below) are used in similar manner.

The *maxreq* and *maxdone* entries (both default to 4) limit the number of requested digipeat hops to listed amount, and in case of some of those requests being done, also limit the number of executed hops.

The *keys* entry defines multiple "new-n paradigm" style keyword stems that are matched for the first VIA field without H-bit set. See detailed explanations at 4.3 "*Keywords on <trace> and <wide> sub-sections of <digipeater> sections.*"

The system has a special "heard direct" behaviour when *maxreq* or maxdone is reached or exceeded, or when a *new-n paradigm* style "KEYn-N" entry has N > n: The system will insert itself at first VIA slot, and mark original request VIA data with H-bits ("digipeating completed") and transmit the resulting packet. If packet was not recognized as "heard direct", then it is silently dropped.

3.8.2 The <*wide*> sub-section

```
<wide>
     maxreq    N    # in range: 1 .. 7, default: 4
     maxdone   N    # in range: 1 .. 7, default: 4
     keys      TRACE,WIDE
</wide>
```

The <*wide*> sub-section keywords are matched only, if <trace> sub-section keywords have not matched. Match on the <wide> sub-section keywords means that no "trace" behaviour is done on outgoing digipeated packet's AX.25 address fields, only decrementing the request counts.

Otherwise same notes apply as on <trace> sub-section.

3.8.3 The *<source>* sub-sections

```
<source>
     source         callsign
     #via-path      foo  # for “source APRSIS” only!
     relay-type     keyword
     viscous-delay  N    # in range: 0..9, default: 0
     ratelimit      60 120
     regex-filter   ...
     filter         ...
     <trace>
          ...
     </trace>
     <wide>
          ...
     </wide>
</source>
```

The *source* entry uses a *callsign* reference to <interface> sections. There is one special built-in interface in addition to those that you define: “APRSIS”, which is basis of Tx-iGate implementation.

The *<trace>* and *<wide>* sub-sub-sections define source specific instances of respective processing rules. This way a source on 50 MHz band can have special treatment rules for <trace>/<wide>, while <digipeater> wide rules are used for other sources. Presence of <trace>/<wide> sub-sub-sections overrides respective <digipeater> section level versions of themselves.

The *via-path* is used only on APRSIS case, where it defines the VIA-path for outgoing 3rd-party frame, and it defaults to empty VIA-path.

The *ratelimit* gives *average* and *upper* limit on number of packets to be relayed per minute from this source to the digipeater where this belongs to. System tracks number of packets sent per 5 second timeslots using “token bucket filter” algorithm. Default values are 60/120. Maximum values are 300/300. (Default limit saturates 1200 bps AFSK channel, but 9600 bps channel has some slack.)

The *relay-type* defines how the digipeater modifies AX.25 address fields it passes through. Available values are: *3rd-party (“third-party”)* for APRSIS Tx-iGate use only, *digipeated* (default value,) and *directonly* for special fill-in digipeaters.

The *viscous-delay* is an auxiliary parameter intended for *relay-type directonly* digipeaters, and digipeaters with Tx-iGate. It defines number of seconds that this digipeater will hold on the packet, and account any other possible arrival of same packet by means of comparing early all packets on transmitter specific duplicate filter. If at the end of the delay this is still unique observation of the packet, then it will be sent out. Using the *viscous-delay* on APRSIS sources gives RF network a small change to have the same packet reach this node. Using the *viscous-delay* on normal digipeater is not recommended.

A non-zero *viscous-delay* value gets always an additional per packet randomized 0 to 2 second delay. This allows similarly configured Aprx servers to hopefully not transmit all at same time.

The *filter* entries define javAPRSSrvr style adjunct filter entries that must pass through the arriving packet. Further details below.

The *regex-filter* supplies an ad-hoc mechanism to *reject* matching things from packets. Further details below.

3.8.3.1 Filter entries

Only following of javAPRSSrvr adjunct filter *like* entries are supported on <source> sub-sections:

- A/latN/lonW/latS/lonE
- B/call1/call2...
- F/call/dist_km
- O/object1/object2...
- P/aa/bb/cc...
- R/lat/lon/dist_km
- S/pri/alt/overlay
- T/.../call/km
- U/unproto1/unproto2...

For more information about their syntax, see:
http://www.aprs-is.net/javAPRSFilter.aspx

Filter usage rules:

1. These filters apply only on APRS packets. Digipeating other types of packets is not subject to these filters.
2. If there are no filters, then arriving APRS packet is always accepted.
3. You can define as many filter entries as you want.
4. The filter entries are evaluated in definition order.
5. The filter entries apply only on the <source> sub-section they are in. Different <source> sub-sections have separate filtering rule sets.
6. When you define filter entries, only those packets matching a filter rule are passed on.
7. With multiple filter definitions, a match terminates filter chain evaluation, non-match continues to next filter entry.
    1. Prefixing a filter entry with minus ("-") will cause the match to reject the packet.
    2. Rejection filters are evaluated after acceptance filters, thus you can reject something that e.g. area acceptance accepted.
8. If none of filters matched, then:
    1. the packet will be rejected on RF->RF digipeating
    2. the Tx-IGate passes messages onwards with its own algorithm when no filter matched.

Example at “Limited Service Area Digipeater” used these settings:

```
        filter     t/m      # All messages (position-less)
        filter     a/60.00/23.00/59.00/25.20
        filter     a/60.25/25.19/59.00/27.00
```

3.8.3.2 Regex-filter entries

A set of *regex-filter* rules can be used to reject packets that are not of approved kind. Available syntax is:

- regex-filter source RE
- regex-filter destination RE
- regex-filter via RE
- regex-filter data RE

The keywords “source”, “destination”, “via”, “data” tell which part of the AX.25 packet the following regular expression is applied on. Defining multiple entries of same keyword will be tested in sequence on that data field. First one matching will terminate the matcher and cause the packet to be rejected.

The *regex-filter* exists as ad-hoc method when all else fails.

3.8.4 Digipeating other than APRS packets

The Aprx can also be used as generic AX.25 RF digipeater.

The Aprx will handle RF->RF digipeating of other than APRS packets by matching “next hop” (via) field matching on transmitter callsign or transmitter interface aliases.

The “*new-n-paradigm*” “WIDEn-N” rules do not apply for other than APRS packets.

Digipeater *ratelimit* filter applies to all kinds of packets on digipeater sources and transmitters.

The regex filters can be applied to block packets, they work on all kinds of packets.

4 Running the Aprx Program

Following talks about how the pre-compiled binaries for Debian and RedHat style systems are run. If you have some other platform, you must adapt these instructions in applicable ways to your environment.

4.1 Normal Operational Running

The Aprx program is intended to be run without any command line parameters, but some are available for normal operation:

- -f -- tell location of runtime aprx.conf file, if default is not suitable for some reason.
- -L -- log a bit more on aprx.log

Depending on your host system, you may need to setup init-script and its associated start-up parameter file.

4.1.1 On RedHat/Fedora/SuSE/relatives

After doing installation on this type of machine either from pre-compiled binary package, or from sources:

In order to use package contained startup scripts, you will need to edit the `/etc/sysconfig/aprx` to contain following:

```
    STARTAPRX="yes"
    DAEMON_OPTS=""
```

After installation you may need to execute following two commands as root:

```
    chkconfig aprx on
    service aprx start
```

4.1.2 On Debian/Ubuntu/derivatives

After doing installation on this type of machine either from pre-compiled binary package, or from sources:

In order to use package contained startup scripts, you will need to edit the `/etc/default/aprx` to contain following:

```
    STARTAPRX="yes"
    DAEMON_OPTS=""
```

After installation you may need to execute following two commands as root:

```
     update-rc.d aprx defaults 84
     /etc/init.d/aprx start
```

The second command here should be postponed until the configuration is completed.

4.1.3 Logrotate (Linux systems)

You will also need logrotate service file. This one is handy to rotate your possible logs so that especially embedded installations should never overflow their RAM-disks with useless log files. The file in question is something like this:

```
/var/log/aprx/aprx-rf.log /var/log/aprx/aprx.log /var/log/aprx/erlang.log {
        monthly
        rotate 24
#       compress
        missingok
        notifempty
        create 644 root adm
}
```

4.2 The aprs.fi Services for Aprx

The Aprx sends telemetry packets for each active radio interface, and aprs.fi can plot them to you in time-series graphs per APRS specification 1.0.1 Telemetry packet rules.

These graphs are:

1. Channel received Erlang estimate (based on received bytes, not actual detected higher RSSI levels)
2. Channel transmitted Erlang estimate (based on number of transmitted bytes, not actual PTT time)
3. Number of received packets on channel
4. Number of packets that receiving iGate function discarded for one reason or another
5. Number of transmitted packets

All represent counts/averages scaled to be over 10 minute time period.

4.3 Keywords on <trace> and <wide> sub-sections of <digipeater> sections.

```
<trace>
     maxreq    N    # in range: 1 .. 7, default: 4
     maxdone   N    # in range: 1 .. 7, default: 4
     keys      TRACE,WIDE,RELAY
</trace>

<wide>
     maxreq    N    # in range: 1 .. 7, default: 4
     maxdone   N    # in range: 1 .. 7, default: 4
     keys      TRACE,WIDE,RELAY
</wide>
```

These definitions tell what “new-n-paradigm” word stems (up to 5 chars each) are treated as “trace”, and what are treated as “wide”.

At first the system looks at “trace” entries, and if it finds that a word-stem there is used on packet to be digipeated, trace happens. The “wide” is looked up after trace, otherwise same function.

In all cases the 'maxreq' checks that in case of:

CALL>APRS,WIDE1-1,WIDE3-3,WIDE3-3

the total number of requested hops is 1+3+3 = 7, and that as it exceeds the “maxreq” value, the packet will not be digipeated!

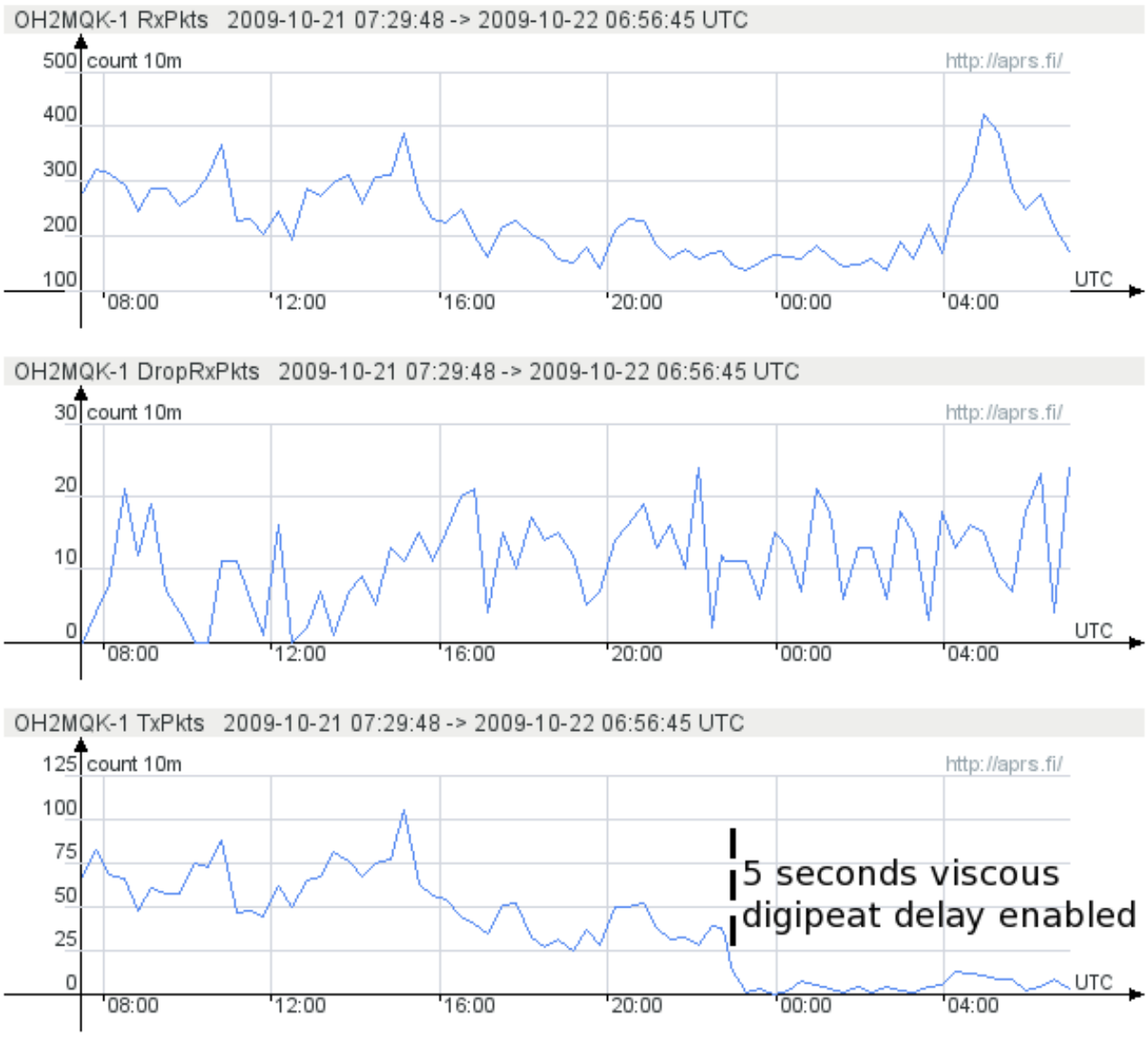
The “maxdone” limits hopping like this:

CALL>APRS,WIDE1*,WIDE3-1,WIDE3-3

it sees that “WIDE1*” means “1 done, list exhausted”, “WIDE3-1” means “2 done”, and “WIDE3-3” means “0 done”. Total is then: 1+2+0 = 3, which is not yet over “maxdone” default value.

4.4 Effect of “viscous-delay” on a Digipeater

Use of *viscous-delay 5* parameter on a <digipeater>'s <source> sub-section has dramatic effect on number of packets that it sends out, and could be useful feature for a low-priority digipeater. Normally the big digipeaters are able to pick up the traffic, but sometimes they can miss it. Alternatively of course multiple digipeaters are repeating it simultaneously, and your little node can not hear those major digipeaters because your antenna receives at least two of them equally strong and your FM receiver can not separate them.



5 Compile Time Options

Small memory footprint configuration option: --with-embedded (now default!)

Prefer to use: --with-pthreads

Use of pthreads results in a few 100 kB smaller memory footprint, and suitability to be used on system without UNIX fork(2) semantics.

5.1 Building Debian package

The package has built-in default Makefile with ability to execute following command:

```
$ make make-deb
```

which executes for a while, and finally:

```
....
dpkg-deb: building package `aprx' in `../aprx_2.01.422-1_i386.deb'.
make[1]: Leaving directory `/home/matti/aprx-2.01.svn422'
 dpkg-genchanges -b >../aprx_2.01.422-1_i386.changes
dpkg-genchanges: binary-only upload - not including any source code
dpkg-buildpackage: binary only upload (no source included)
```

and the binary package is ready for installation.

5.2 Building RPM package

The package has built-in default Makefile with ability to execute following command:

```
$ make make-rpm
```

This page is intentionally left blank

6 Debugging

6.1 Testing Configuration

Testing the Aprx configuration, and many other things, is accomplished with command line parameters:

- -d
- -dd
- -ddd
- -dddd
- -ddv

Starting the program without these parameters will run it on background, and be silent about all problems it may encounter, however these options are *not* to be used in normal software start scripts!

These give increasing amount of debug printouts to interactive terminal session that started the program.

The program parses its configuration, and reports what it got from there. Possible wrong interpretations of parameters are observable here, as well as straight error indications. Running it with these options for 10-20 seconds will show initial start phase, at about 30 seconds the first beacons and telemetry packets are sent out and normal processing loops have begun.

To observe all beacon transmissions on a beacon cycle, you must monitor it for one whole cycle-size interval (default: 20 minutes!)

Program being tested with these options is killable simply by pressing Ctrl-C on controlling terminal.

6.2 Hunting bugs

Nothing beats experience and throughout knowledge of the code, however good hints are available by using various debugging tools.

6.2.1 With gdb

The program is compiled by default with debug symbols, thus if it crashes and “drops a core”, as it is known in UNIX systems, usual method to look for a clue within the core is:

```
# gdb /usr/sbin/aprx core.123
...
(gdb) where
```

You can also run the program under debugger so that it falls back to it in case of some error, or reaching some interesting point, or whatever:

```
# gdb /usr/sbin/aprx
(gdb) run -i
...
(gdb) where
```

6.2.2 With valgrind

The valgrind is UNIX environment tool chest that can find different kind of memory errors in codes. Scribling over end of buffer, wrong allocation/free pointers, referring on freed memory, not freeing memory properly, etc.

Use of it is a bit of an art-form, but you need to compile the Aprx with suitable preparatory step for valgrinding:

```
[aprx] $ make clean
[aprx] $ make valgrind
```

Then using the valgrind and its many tools:

```
# valgrind --tool=drd -v ./aprx -d
# valgrind --tool=memcheck -v --leak-check=full ./aprx -d
```

Redirecting the tool outputs to a file can also be useful, as some pesky errors have been found only after several days of hunting. See also a tool called “script”.

7 Colophon

This document was written with OpenOffice 3.2.1 producing ODT file, and then exporting a PDF file out of it. Unfortunately this system is unable to produce decent HTML for web-pages out of same document source.

Alternate technologies, like editing DOCBOOK SGML were considered, but thought to be a bit too cumbersome for initial writeup. The DOCBOOK would give excellent printed and web format outputs.