

Aprx 1.97 Manual

Table of Contents

| | | |
|----|--|----|
| 1 | What is APRX? | 2 |
| 2 | Configuration Examples: | 3 |
| 2 | Minimal Configuration of Rx-iGate | 3 |
| 3 | Minimal Configuration APRS Digipeater | 4 |
| 4 | Combined APRS Digipeater and Rx-iGate | 5 |
| 5 | Doing Transmit-iGate | 6 |
| 6 | Digipeater and Transmit-iGate | 7 |
| 7 | A Fill-In Digipeater | 8 |
| 8 | Using Multiple Radios | 9 |
| 9 | A Digipeater with Multiple Radios | 10 |
| 10 | A Bi-Directional Cross-band Digipeater | 11 |
| 11 | Limited Service Area Digipeater | 12 |
| 3 | Configuration in details | 13 |
| 1 | Aprx Configuration Parameter Types | 13 |
| 2 | The "mycall" Parameter | 14 |
| 3 | The "<aprsis>" section | 15 |
| 4 | The "<logging>" section | 16 |
| 5 | The "<interface>" sections | 17 |
| 6 | The "<beacon>" sections | 23 |
| 7 | The "<digipeater>" sections | 26 |
| 4 | Running the Aprx Program | 30 |
| 1 | Normal Operational Running | 30 |
| 2 | Testing Configuration | 31 |
| 3 | The aprs.fi Services for Aprx | 32 |
| 5 | Compile Time Options | 33 |

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 64 MB ram disk, a 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.

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

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.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 might want to have a peek 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: euro.aprs2.net

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>
```

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.

5 Doing Transmit-iGate

At the time of the writing, doing Tx-iGate is still lacking some bits necessary for correct functioning, and it is relaying too much traffic from APRSIS to RF.

```
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
    via-path     WIDE1-1    # default: none
    filter       t/m
  </source>
</digipeater>
```

This is Rx/Tx-iGate in a form that Aprx version 1.97 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!

At the time of the writing, doing Tx-iGate is still lacking some bits necessary for correct functioning, and it is relaying too much traffic from APRSIS to RF.

```
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 digipeater.

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”.

```
<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, all others must have explicit 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. XOR checksum on KISS: So called “**BPQCRC**” alias “XKISS”. 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!**

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

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-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.

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
    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.

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. <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)
- Very long parameter lines can be folded 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.

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 *NOCALL-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.

The KISS variations:

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

1. **Stuttgart Modified Amateur-radio-CRC-KISS (SMACK)**
2. **BPQCRC** alias **XKISS**
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!

It is available for TNC2 clones from:

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

Another KISS variation that is **not** supported is FLEXCRC. It is a bit like SMACK, but with different CRC polynomial. Adding support for it is possible, if somebody really wants it. Something else entirely would be 6PACK – which has low latency timing data

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, 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.

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 call-signs, 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, 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.

Networked tcp-stream connected terminal devices:

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

<interface>
    tcp-device      hostname-or-ip-address portnumber TNC
    .....via-
</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!

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, 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:

- *to interface-callsign*
- *for callsign*
- *dest callsign*
- *via “viapath”*
- *timefix*

... continued ...

The *to* 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 *for* 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 *dest* 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.

... continued ...

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

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 beacons packet. It has no default value.

Aprx 1.97 Manual

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" lat "6044.09N" lon "02612.79E" \
    symbol "/r" comment "434.775MHz TOFF -1600kHz R50k OH2RAY"
</beacon>
```

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.

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      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 upper limit on number of packets to be relayed per minute. System tracks number of packets sent per 3 second timeslots, and keeps total on 60 seconds under the ratelimit value. Default value is 60. Maximum value is 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.

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.

FIXME: “heard direct” behaviour when maxreq is reached or exceeded is to mark all VIA fields with H-bit, and then to transmit it. On other situation observed excess leads to silent dropping of packet.

The *keys* entry defines multiple “new-n paradigm” style keyword stems that are matched for the first VIA field without H-bit set.

FIXME: keywords to choose, source specific keywords, ...

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.

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
  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 `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 auxiliary parameter usable on *relay-type directonly* digipeaters. 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.

The `filter` entries define javAPRSSrvr style adjunct filter entries that must pass through the arriving packet. Not all adjunct filters make any sense on a digipeater...

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

Filter entries:

Only following of javAPRSSvr adjunct filter entries are supported on <source> section:

- 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>

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.

4 Running the Aprx Program

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 startup parameter file.

On RedHat/Fedora/SuSE/relatives:

```
/etc/sysconfig/aprx
STARTAPRX="yes"
DAEMON_OPTS=""
```

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

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

On Debian/Ubuntu/derivatives:

```
/etc/default/aprx
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
```

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
}
```

2 Testing Configuration

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

- -d
- -dd
- -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.

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

3 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.

5 Compile Time Options

TO BE WRITTEN