Monitor configuration file syntax

James Finnis, jaf18@aber.ac.uk

2013

Contents

1 Blocks

A file consist of blocks, which are either:

- variable blocks, describing the variables expected from the UDP client;

- window definitions, describing the windows and the frames and widgets within them;

- audio warning definitions. they contain;

- configuration options.

```
file        ::= { block }

block       ::= varblock
            |   window
            |   configoption
            |   audio
            |   waypointdef
```

1.1 Configuration options

These are defined at the top level:

- **validtime** : if a data buffer has not received data for this long, the data in the buffer is considered to be invalid. This is useful when there is a reasonably good connection and a long wait between updates is a symptom of a fault. It is set by default to a very long interval (about two years.)

- **port** : this is the port to which the system should listen for UDP packets — by default 13231, although it can be set with the `-p` option on the command line, which will override any value set in the configuration file.

- **sendport** : this is the port number for UDP control packets, by default 33333.

- **sendaddr** : this is the address to which control packets are sent. Only set this if you want to override the default setting, which is to send to the address from which the first telemetry packet comes.

- **sendinterval** : the interval between sends of UDP packets for *always send* widgets, by default 2 seconds.

- **updateinterval** : the time between ticks at which all graphics are updated, by default 2 seconds.

```
configoption::= 'port' int
            |   'sendport' int
            |   'sendaddr' string
            |   'validtime' float
            |   'sendinterval' float
            |   'updateinterval' float
```

1.2 Waypoint definition

Waypoints, by default, consist of just latitude and longitude. Extra fields can be added using the **waypoint** keyword. Note that the **lat** and **lon** fields do not need to be specified, they are assumed to be always present. A waypoint definition simply adds extra fields to these, along with default vales for new waypoints.

```
waypointdef ::= 'waypoint' '{' {ident float} '}'
```

1.3 Variable blocks

Variable blocks describe the variables which arrive on the UDP connection, and consist of a list of variable definitions surrounded by curly brackets[1].

```
varblock    ::= 'var' '{' {vardef} '}'
```

Currently only floating point variables are supported. Some floating point variables may be *linked.*

A normal floating point variable definition specifies the name of the variable (an identifier,) the size of the cyclic buffer backing the variable, and a range specification.

A linked variable definition consists of a number of variables, separated by commas, in brackets; followed by a buffer size. Linked variable specifications give the name of the variable and a range (which cannot be 'auto' — see below.)

```
vardef      ::= 'float' varname buffersize 'range' rangespec
            |   'linked' '(' { linkedvar ',' } linkedvar ')' buffersize

linkedvar   ::= 'float' varname 'range' float 'to' float

buffersize  ::= int
varname     ::= ident
```

When a message arrives giving the value of a linked variable, dummy entries with the same timestamp are created for other variables in the link with the previous value those variables had. Linked variables are used for sets of variables which comprise a single entity, such as latitude and longitude or *xyz* coordinates, where if a change is received on one variable the others should be considered to have changed even though an explicit change was not sent.

Range specifications describe the values a variable can take in normal operation — these are the values any widget viewing this variable will be able to show. If the range is specified as *auto*, the range will be determined dynamically. Linked variables cannot have auto range.

```
<rangespec> ::= <float> to <float>
                | auto
```

1.4 Window blocks

Window blocks describe the contents of a window. Windows can appear on any display, and can be fullscreen if desired. There are a number of options:

- the title of the window can be set;

- the window can be set to fullscreen;

- the size of the window can be set — default is as small as possible, and this option is ignored in fullscreen windows;

- a 'screen' can be set, in which case the system will be scanned for a display of the given size and the window placed on that window;

- a window can be specified as "inverse" — black on white. This may be easier to read in bright light, but looks ugly;

- an window number can be specified — pressing this key will switch to that window.

---

[1]change from first version, for syntactical orthogonality.

```
window      ::= 'window' [ {windowopt} ] '{' {frame|widget} '}'
windowopt   ::= 'title' string
            |   'fullscreen'
            |   'size' int ',' int
            |   'screen' int ',' int
            |   'inverse'
            |   'number' int
```

1.5 Frame blocks

Frame blocks describe a frame within the window, optionally surrounded by a border, and its contents. The frame definition consists of a position (see below) giving the position and size of the frame within its container, some options, and then a list of contents in curly brackets — which can be widgets or more frames, just like for a window.

```
frameblock  ::= 'frame' pos [{frameopt}] '{' {frame|widget} '}'

frameopt    ::= 'borderless'
            |   'spacing' int
```

1.5.1 Positions

Positions describe where elements appear in a container's grid layout, and how many rows and columns they take up. They are either an x,y pair (with an implied width and height) or a full x, y, w, h set.

In addition, a position for a widget can optionally be followed by a size, giving the *minimum* pixel size of the widget in both x and y. If only one number is given, the same value is used for both dimensions.

```
pos         ::=  x ',' y [ ',' w ',' h ] [ 'size' minw [ ',' minh ] ]

x           ::= int
y           ::= int
w           ::= int
h           ::= int
minw        ::= int
minh        ::= int
```

Minimum size options on frames and windows are ignored.

1.6 Widgets

All widget specification consist of the widget type, followed by the position, followed by widget specification in curly brackets.

```
widget      ::= gauge | number | graph | map | status |
                compass | switch | momentary | slider
```

1.6.1 Sources

A source specification describes a data source. It is either a variable, or an expression and a range.

```
source      ::= 'var' varname
            |   'expr' string 'range' rangespec
```

An expression is an infix expression in double quotes consisting of:

- names of variables declared in the **var** blocks;

- float constants;

- the four arithmetic operators with their usual precedence;

- the comparison operators < > >= <= != =

- the logical negate operator "!"

- the logical operators && and ||.

1.6.2 Gauge widgets

A gauge widget consists of the position, followed by any of the following, some of which are mandatory:

- a **source specification** (mandatory)

- a **position specification** (mandatory)

- a **title string** giving the gauge's label — without this, the variable name or expression string from the source are used;

- a **subtitle string** giving a smaller label, which is empty by default;

- a **levels specification** giving the value of the warning and danger levels in terms of the input source range — if the source range is auto, these should be 0-1. If *previous* is specified, the preceding level specification is used;

- a **colours** specification giving colours for the normal, warn and danger ticks;

- a **darken** factor, giving the value used to darken the colours to show the "off" ticks on the gauge. The default is 400, which means that the dark colour is a quarter of the bright colour. The higher the value, the darker the colour. A *previous* value is also accepted.

```
gauge       ::= 'gauge' pos '{' { gaugemod } '}'
gaugemod    ::=
            |   source
            |   'title' string
            |   'subtitle' string
            |   'levels' levelspec
            |   'colours' grcolspec
            |   'fontscale' float
            |   'darken' (int | 'previous')

levelspec   ::= warnlevel dangerlevel
            |   'previous'

grcolspec   ::= normcol warncol dangercol
            |   'previous'

normcol     ::= colour
warncol     ::= colour
dangercol   ::= colour

colour      ::= colourname
            |   '"#' hexdigit hexdigit hexdigit '"'

warnlevel   ::= float
dangerlevel ::= float
```

1.6.3 Number widgets

Number widgets consists of a position, an optional title and a source.

```
number      ::= 'number' pos '{' [ 'title' string ] source '}'
```

1.6.4 Graph widgets

This consists of a position, a time value (giving the width of the graph in seconds) and a list of graph sources.

Each source consists of a source specification and a set of graph modifiers in curly brackets, which describe the style of line drawn for that source.

```
graph       ::= 'graph' pos '{' [ 'time' float ] { graphsource } '}'

graphsource ::= source '{' { graphmod } '}'

graphmod    ::= 'width' float
            |   'colour' colour
```

1.6.5 Maps

A map shows an map image with points overlaid. It consists of a screen position, as with other widgets, followed by a set of items to render: map points, vectors or lines.

There may also appear the longitude and latitude at which to initially centre the map, and the zoom level as specified by the height of the camera in kilometers above the ground.

Finally, a pair of out values may be specified. These values are written (immediately, if 'immediate' is also given) to the output port as a latitude, longitude pair. See the sections on momentaries and switches for more details of how output variables work. Note that there is no feedback system available for these variables.

```
map         ::= 'map' pos '{'
                    [ 'centre' lat ',' lon ]
                    [ 'height' km ]
                    [ 'out' ident ',' ident ]
                    [ 'immediate' ]
                    [ 'always' ]

                    { mappoint|mapvector|mapline|mapimage|waypointview}
                '}'
km          ::= float
lon         ::= float
lat         ::= float
```

Autocentering: Each map item can be specified as the autocentering item by adding the word `centre` to its specification. If an autocentering item is given, the first datum received on that item will cause the map to center to that position.

Manual centre and zoom: If a momentary button has the special "resetmaps" (see Momentary Buttons, section 1.6.12) pressing that momentary button, or its mapped key, will cause the next draw to automatically center and zoom the map to make all its items visible. Here's an example of such a button:

```
momentary 3,4 {
    title "reset"
    special "resetmaps"
    key "r"
}
```

Map points represent data as circles on the map whose colour and screen size depends on data sources. Their definitions consist of the word 'point' followed by a point specification in curly brackets. This consists of a location specification (two sources separated by commas for latitude and longitude respectively) followed by a list of map point modifiers. These describe how the point is drawn. Finally, there should be an 'on' clause, which specifies how new points are drawn — a new point is created whenever a new datum value arrives in this source.

The point modifiers can be:

- a base colour (white by default),

- a default size,

- a size range clause mapping a source onto a size range (which replaces the default size,)

- a hue clause specifying a source which can be mapped onto the hue range,

- similar saturation and value sources,

- a trail size to allow a number of historical points to be rendered,

- a format string and a source to render with that string — for example

  ```
  label "waypoint %.0f" var waypoint_number
  ```

  will render the waypoint number variable as an integer. You may need to be careful with this, since the value at the label may be interpolated if it is not present in every datum.

```
mappoint    ::= 'point' '{' location [{ pointmod }] 'on' source '}'

pointmod    ::= 'colour' colour
            |   'size' float
            |   'sizerange' minsize maxsize source

            |   'hue' source
            |   'saturation' source
            |   'value' source

            |   'trail' int
            |   'label' string source

location    ::= lat ',' long
lat         ::= source
long        ::= source
```

Map vectors are similar to points, but represent data by a line coming from a point on the map. The line's starting location, length (in pixels), colour and width can be modified by data sources in a similar way to the size and colour of a point. The 'clip' word specifies that the vector is only drawn when its start point is visible within the widget.

```
mapvector    ::= 'vector' '{' location [{ vectormod }] 'on' source '}'

vectormod    ::= 'colour' colour
            |   'hue' source
            |   'saturation' source
            |   'value' source
            |   'width' float
            |   'widthrange' minsize maxsize source
            |   'length' float
            |   'lengthrange' minsize maxsize source
            |   'trail' int
            |   'clip'

location     ::= lat ',' long
lat          ::= source
long         ::= source
```

Map lines are single lines which link two coordinates on the map. Their width and colour can be specified in the same way as vectors. One set of coordinates is the start, one is the end. As with all map items, an 'on' source is required to determine which data should have a line drawn — the actual values are interpolated from the time of the most recent datum for that source. The 'clip' word specifies that the line is only drawn when both end points are visible within the widget.

```
mapline ::=      'line' '{'
                 'start' lat ',' long
                 'end' lat ',' long
                 [{ linemod }]
                 'on' source
                 '}'

lat        ::= source
long       ::= source

linemod    ::= 'colour' colour
           |   'hue' source
           |   'saturation' source
           |   'value' source
           |   'width' float
           |   'widthrange' minsize maxsize source
           |   'clip'
```

Map images are bitmaps which can be overlayed on the map, once you've worked out the correct registration. Multiple images can be used; they will be rendered in the order they are described. The format for a map image block is:

```
mapimage    ::= 'image' '{'
                    string
                    { 'alpha' float }
                    'pos' float,float
                    'image' float,float
                    'pos' float,float
                    'image' float,float
                    '}'
```

Each pos/image pair specifies a latitude and longitude (after "pos") and a pair of pixel coordinates within the image corresponding to that point (after "image".) The optional alpha clause specifies how opaque the map is rendered, from 0 to 1. An example:

```
image {
    "aberystwyth.png"
    alpha 0.5  # for a half-transparent image

    # point 1 (Pendinas monument)
    pos 52.40163,-4.08237   image 187,411
    # point 2 (The Clarach junction in Bow Street)
    pos 52.44202,-4.02809   image 500,23
}
```

1.6.6   Waypoint renderer

Waypoints are shown on the map if the map has a waypoint clause. This currently has no options, and so appears as

```
waypoint {}
```

See sections 1.2 and **??** for more details.

1.6.7 Status widgets

A status widget consists of a position, a grid size specification (giving the number of rows and columns of indicators in the grid) and then a set of status indicators, which of either of type *floatrange* or the simpler type *bool*.

```
status      ::= 'status' pos '{'
                  'size' int ',' int
                  { floatrange|boolstat }
                '}'
```

Floatrange indicators consist of 'floatrange', and a specification in curly brackets containing a grid position, a title, a source and then a set of bands. Optionally there can then follow a set of *when* clauses, which specify alternate title strings to use when the indicator is a particular colour.

```
floatrange  ::= 'floatrange' '{'
                  'pos' int ',' int
                  'title' string
                  source
                  bands
                  [{ 'when' statcolour string }]
                '}'
```

The bands in a status indicator consist of 'bands' followed by a position within the grid, a title string which appears in the box, and a set of band specifications, each of which is a less-than sign, a float, and a colour. When a value is received on the source, is goes through each of these bands is first, and the first for which the comparison is true gives the colour. The bands end with an *else* clause giving the default colour.

An alternative form of band specification is the word 'previous,' which means "copy the bands from the previous indicator's specification." It is an error to use this in the first indicator.

```
bands       ::= 'bands' ( fullbands | 'previous' )
fullbands   ::= { '<' float statcolour }
                  'else' statcolour
statcolour  ::= 'red' | 'green' | 'blue' | 'black' | 'yellow'
```

Bool indicators are used when a source contains a value which is negative for false and positive for true. They consist of 'bool' and a specification in curly brackets containing of the grid position and title, followed by a source specification. This can then be followed by colour values for true and false, with the defaults being green and black respectively.

```
boolstat    ::= 'bool' '{'
                  'pos' int ',' int
                  'title' string
                  source
                  { 'false' statcolour }
                  { 'true' statcolour }
                '}'
```

The convention of "negative is false, positive is true" is that supported by logical and comparison operators in the expression parser and bitfield system.

+-----------------------------+
| **To Do** |
| |
| Bitfield system |
+-----------------------------+

1.6.8 Compass

The compass widget simply shows a heading. Its source is a value in degrees:

```
compass      ::= 'compass' pos '{'
                 [ 'title' string ]
                 source
                 '}'
```

1.6.9 Switch

The toggle switch is an output widget — that is, its primary purpose is to send data out via a UDP packet. Switches can operate with or without a "feedback source." If such a source is set, the switch's visible state is compared with the value of this source and any mismatch indicated. Without a feedback source, the packets are sent from the switch with no indication of whether they were successfully received. The actual data sent (and received) by the switch are booleans in the form of floats, 0 or 1. In feedback mode, the switch will attempt to read an initial value from the server.

To specify a feedback source, use the 'var' clause, either with a variable name or with an expression[2]. In non-feedback mode, the switch has three states:

- **red** corresponds to 0,

- **green** corresponds to 1,

- **dark grey** means the data is queued for sending but has not yet been sent (see section 3.1.)

In feedback mode, the switch has four states — the colours indicate these states as well as the on/off state of the switch in the OK state:

- **yellow diagonal crosshatch** (INIT) means no value has been received from the server, so we don't know what value the server is currently set to.

- **red** (OK) corresponds to 0 received (or acknowledged);

- **green** (OK) corresponds to 1 received (or acknowledged);

- **grey** (UNSENT) means the value has been changed locally, but has not yet been sent. You can set *immediate* to make sure a value is sent immediately on change;

- **diagonal crosshatch** (UNACK) means a value has been sent (indicated by the colour) but a feedback value has not yet been received;

- **full crosshatch** (BADACK) means a value has been sent (indicated by the colour) and a feedback value has been received, but the two do not match — an error condition.

The switch can only be clicked in OK or BADACK states. Clicking in OK will toggle the switch and send the new value, clicking in BADACK will clear the error and set the state of the switch to that reported by the server. The state diagram in figure 1 shows this more clearly. See Section **??** for details of how to write remote code to deal with switches.

Switch specifications consist of a position, the name of an output variable to use in the send packets, a title, an an optional feedback source. There may also be the word `always`, in which case the value for this switch is always sent, whenever any other output widget changes and when the send interval elapses; and the word 'immediate' indicating that when the switch is pressed, a UDP send update should be done immediately. This will result in the switch's data being sent, as well as any data for other 'always' output widgets. The word 'button' will make the switch look more like a momentary button than a crude toggle. It's a purely visual change.

If 'integer' is set, the switch will round all values it sends — this is important if the value is converted to an integer at the remote end, because otherwise the feedback value will not match the original float value of the slider.

---

[2]It's called 'var' for historical reasons; in early versions expressions were not permitted.

start

INIT

feedback variable exists:
data received,
set value from it

no feedback variable:
set value to off

OK

data received,
set value from it

click, change
value

no feedback variable:
scheduled data sent

click, set value
to recvd (i.e
clear the error).

data recvd.
ok

UNSENT

feedback variable exists:
scheduled data sent

click, set value
to recvd (i.e
clear the error).

UNACK

data recvd.
disagrees

BADACK

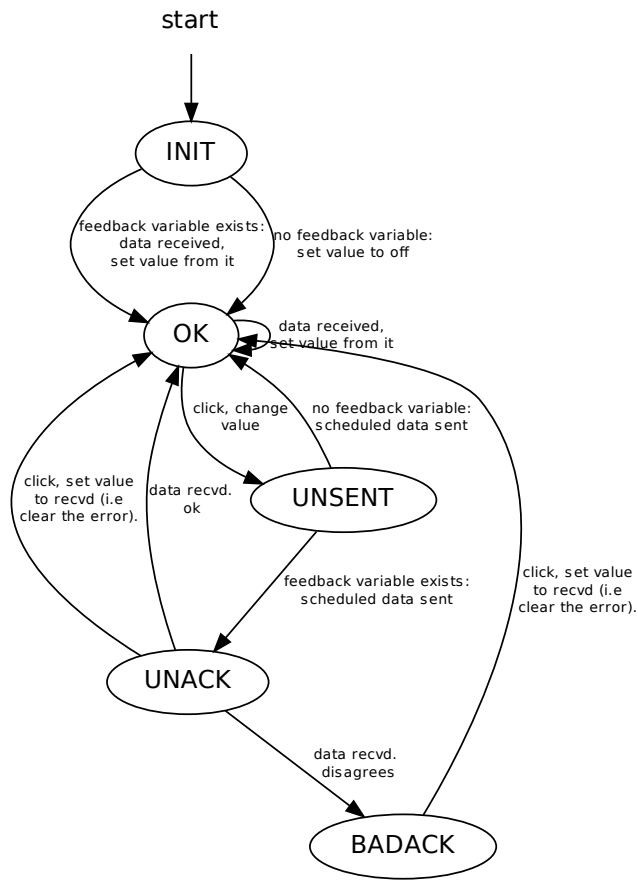Figure 1: State diagram for switches

```
switch      ::= 'switch' pos '{'
                'out' ident
                ['button']
                [ 'size' int ',' int]
                [ 'title' string ]
                [ source ]
                [ 'set' float ]
                [ 'always' ]
                [ 'immediate' ]
                [ 'integer' ]
                [ 'key' keyname ]
                [ 'colours' oncolour ',' offcolour ]
            '}'

oncolour    ::= colour
offcolour   ::= colour
```

1.6.10 Multi-way switches with 'set'

It's often necessary to be able to set a control variable to multiple discrete values — for example, an integer indicating an operating mode. To support this, switches can have a 'set' value which they send instead of the normal 'toggle' operation. The differences are:

- Instead of toggling the value between 0 and 1 and sending it when clicked, 'set' switches always send their 'set' value.

- When feedback arrives on a set switch in the OK state, the switch value is set to off if the feedback does not match the set value, and on if it does.

- When feedback arrives on a set switch in the BADACK state, the switch is set to on if the feedback agrees, and BADACK if it doesn't (i.e. the normal behaviour.)

We can therefore manage multiple values with a set of switches, all controlling the same control variable with different values. For example, consider a three state system: OFF, HOLD and RUN. Let's assume that the integer values for these states are 0, 1 and 2 respectively; and that the control and feedback telemetry variables are both called 'runstate'. We can set up switches to control the state like this:

```
switch 0,0 { button out runstate var runstate set 0 title "OFF"}
switch 1,0 { button out runstate var runstate set 1 title "HOLD"}
switch 2,0 { button out runstate var runstate set 2 title "RUN"}
```

If we click HOLD, `runstate=1` is sent. Remotely, runstate will be set to 1 and feedback sent back to the monitor. The OFF and RUN switches will both be set to off, because the feedback does not agree with their set value; and the HOLD switch will be set to on, because it does.

1.6.11 A note on keys

Note that a switch can have a key mapped onto it. The key name is a string (delimited by quotes) which is either an ASCII character, or one of the following:

home	end	pgup	pgdn
ins	del	up	down
left	right		

1.6.12 Momentary

A momentary button is an output widget, similar to a switch. It also has a feedback source, in a similar manner to a switch — the button's feedback is valid if it is above 0.5 and the packet setting it arrives after the button was pressed. You can change the condition by using a boolean expression, of course. To specify a feedback source, use the 'var' clause, either with a variable name or with an expression.

A momentary button sets its output variable to 1 by default, but that can be changed by using a 'set' clause.

Momentary buttons use a similar colour scheme to that of a switch:

- **light grey** (OK) means the button is ready;

- **green** (WAIT) means the button has recently been pressed and acknowledged (if in feedback mode), and will soon return to the ready state (this a purely cosmetic state;)

- **dark grey** (UNSENT) means the button has been pressed, but data has not yet been sent. You can set *immediate* to make sure a value is sent immediately on change;

- **diagonal crosshatch** (UNACK) means the press has been sent (indicated by the colour) but a feedback value has not yet been received — if this persists it can be cleared with a click;

13

- **full crosshatch** (BADACK) means a value has been sent (indicated by the colour) and a feedback value has been received, but the two do not match (i.e. the value read is less than 0.5.) In this state, the button can be reset by clicking it again, but this will *not result in another send.* It simply clears the error state.

If there is no feedback source, the button will go straight to OK once the value is sent. The button can only be clicked in OK, UNACK and BADACK states: clicking in OK will send the value, clicking in BADACK will, as stated above, clear the error. The state diagram in figure 2 shows this more clearly.
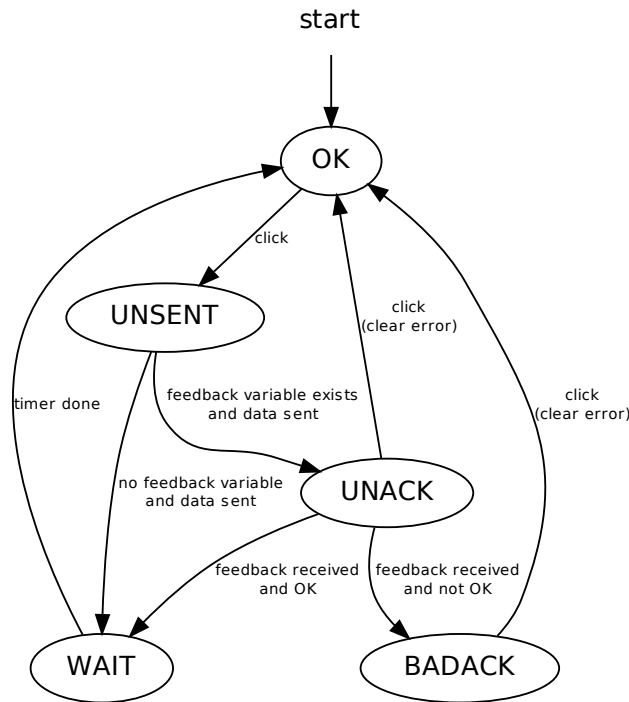

Figure 2: State diagram for momentary buttons

See Section **??** for details of how to write remote code to deal with momentary buttons.

Another use of momentary buttons is to 'nudge' widgets controlling continuous values, such as sliders. Instead of a *out* clause with an output variable name, the word *nudge* and the name of the output variable of a slider is used, along with an action: *up, down* and so on. A feedback source is not permitted with nudge buttons.

```
momentary  ::= 'momentary' pos '{'
                  ( 'out' ident |
                    'nudge' name ('up'|'down'|'centre'|'min'|'max') )
                  [ 'title' string ]
                  [ source ]
                  [ 'always' ]
                  [ 'set' float]
                  [ 'size' int ',' int]
                  [ 'immediate' ]
                  [ 'key' keyname ]
```

14

```
                    [ 'special' string ]
            '}'
```

Special actions: If the "special" string is given, the button will perform a special action, hardwired into the code. Any other specified action, such as an output to a variable or a nudge, will be ignored. The specials are currently:

- **quit** will quit the program

- **resetmaps** will cause all maps to reset their bounding boxes to contain their items

1.6.13 Slider

Sliders are output widgets which control continuous values. They work in a similar manner to switches, but the value controlled is a floating point value over a specified range.

A feedback source can also be specified, which is compared against the value which has been sent. The two values must be within a user-configurable epsilon of each other for a match (currently 0.001). To specify a feedback source, use the 'var' clause, either with a variable name or with an expression.

The handle of the slider shows:

- **dark yellow** (INIT) a feedback value exists but no value has yet been received from the server to initialise the slider (one may have been sent if there is an 'initial' clause.)

- **white** (OK) where the value matches the value on the server (or we don't have a feedback source);

- **grey** (UNSENT or DRAGGING) means data not yet sent or we're actively dragging the slider, about to set a new value;

- **diagonal crosshatch** (UNACK) means data sent but no acknowledgement;

- **full crosshatch** (BADACK) means data sent, but the received value is not yet within epsilon of the sent value. This is not necessarily an error, it may take some time for the physical system to catch up.

In any state, the slider will move by itself as new data comes in with new values; also it's possible to drag the slider to send a new value in any state (except INIT).

See "Momentary" for how to create button widgets which can nudge the slider.

It should be noted that *immediate* with a slider works a little differently from other widgets: the value will only be sent with the slider handle is released. This is to avoid flooding the network with UDP packets as the slider moves. However, using a autorepeat key on a key-mapped momentary to nudge a slider which is set to immediate will result in many UDP packets being sent!

```
slider      ::= 'slider' pos '{'
                'out' ident
                'range' float 'to' float
                'initial' float
                'var' source
                [ 'title' string ]
                [ source ]
                [ 'always' ]
                [ 'immediate' ]
                [ 'horizontal' | 'vertical' ]
            '}'
```

The state machine for the slider is shown in Figure 3. Note how almost any state will transition to DRAGGING if the slider is pressed — this means that many packets can be sent, even from UNACK and BADACK states.
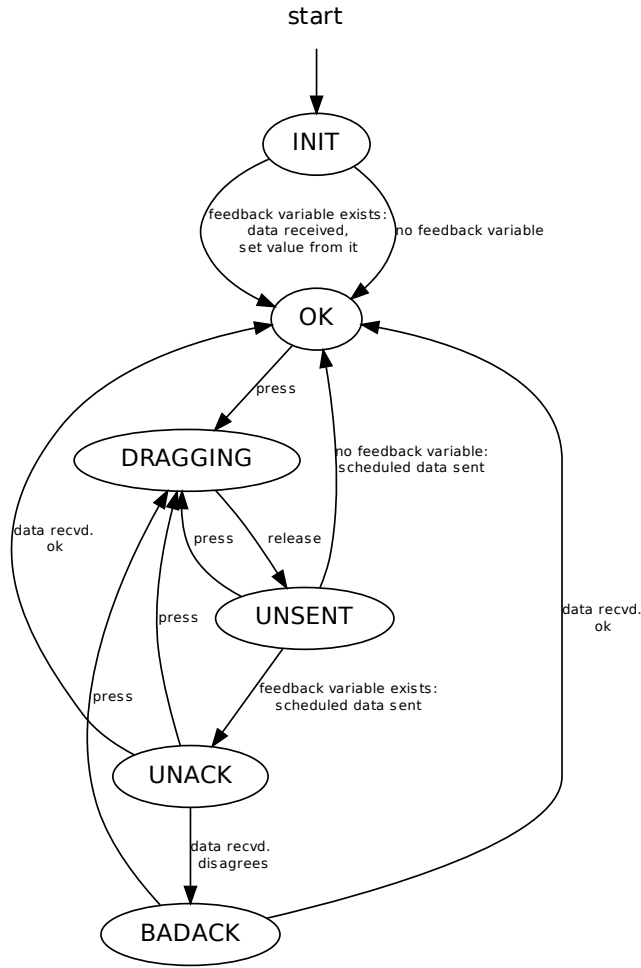
start

INIT

feedback variable exists:
data received,
set value from it

no feedback variable

OK

press

DRAGGING

no feedback variable:
scheduled data sent

data recvd.
ok

press

release

UNSENT

press

feedback variable exists:
scheduled data sent

data recvd.
ok

UNACK

data recvd.
disagrees

BADACK

Figure 3: Slider state diagram

2 Audio warnings

These have the syntax:

```
audio       ::= 'audio' source ( 'sample' | 'speech') string
```

They play either an audio sample or use text-to-speech to read a phrase whenever the source is above zero. They will not trigger if an audio been played within the last two seconds. The checks are made every graphical update tick — by default two seconds, but changed with updateinterval.

Typically, an expression source will be used. For example:

```
audio expr "a>0" range auto sample "warning.wav"
```

or

```
audio expr "speed<5 && power>10" range auto speech "speed low and power high"
```

Audio warnings are currently implemented by using fork() and exec() to run /bin/espeak or /bin/aplay. This is because audio APIs in Qt are large and unwieldy, and often don't work.

3 Other notes

3.1 How output data is sent

Whenever the output update runs, output data is sent for those widgets which are either have changed since the last send, or are set to *always send*. The output update is run every *sendinterval* seconds, which defaults to 2 seconds but can be set by the configuration file. In addition, widgets set as *immediate* cause the output update to run whenever they are changed.

Packets are sent to the same address the telemetry packets are coming from.

3.2 Special variables

Some variables are created at startup:

- **timesincepacket** is the number of seconds since the last packet was received. It is updated every two seconds or so.

- **lastpacketinterval** is the number of seconds between the last packet and the packet before that. It is updated on every packet.

4 Writing remotes

4.1 Telemetry

The simple case is a telemetry-only remote, which just sends information to the monitor. This should send lines consisting of a timestamp in seconds (preferably a UNIX timestamp) and variable values, all as floats in key=value pairs, terminated with a line feed:

```
time=1368451979.922781 a=33.326191 b=1.000000
time=1368451980.823072 a=33.339432
time=1368451981.723353 a=33.352539 b=2.000000
```

Only those variables which change at each tick need to be sent — this is even true in the case of linked variables such as coordinate pairs. Each line should be sent as a UDP packet — currently we're using code like this:

```
bool udpSend(const char *hostip, int port, const char *msg){
    sockaddr_in servaddr;
    int fd = socket(AF_INET,SOCK_DGRAM,0);
    if(fd<0){
        perror("cannot open socket");
        return false;
    }

    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr(hostip);
    servaddr.sin_port = htons(port);
    if (sendto(fd, msg, strlen(msg)+1, 0, // +1 to include terminator
                (sockaddr*)&servaddr, sizeof(servaddr)) < 0){
        perror("cannot send message");
        close(fd);
        return false;
    }
    close(fd);
    return true;
}
```

4.2 Control

Controlling a remote system is a little more complex. We need to consider issues of latency and dropped packets carefully (see the above sections on the slider, switch and momentary button to see how the controls work.) Data is sent from the monitor to a UDP server on the remote, which optionally sends feedback information as part of the standard telemetry.

The control data packets are similar to the telemetry packets: a set of key-value pairs, but no timestamp. They change a different set of values on the remote from those monitored by telemetry. Control data is sent regularly (if any values have changed) and can also be sent when immediate is set on a control.

4.2.1 Framework

There is a framework for control data in the `sampleClient` directory, in `udpserver.h/.cpp`. This defines a `UDPServer` class which can be used to create control variables, get values from them, and periodically poll for changes.

There are three kinds of control:

- toggle switch — controls a 0/1 value which is toggled and resent each time the switch is flipped;

- momentary button — the same value (the "set" value, 1 by default) is sent as the value of the control variable every time the button is pressed;

- slider — controls a continuously varying floating point value, sent every time the slider is moved.

4.2.2 Feedback

It's a good idea to use a telemetry variable to shadow the control variables, sending any changes to the control variables back to the monitor. This can be a variable of the same name, or a different name. It's also possible to use an expression as a feedback value in the monitor.

4.2.3 An example using UDPServer

See `sampleClient/test.cpp` for a full example.