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 they contain;
- configuration options.

```
file        ::= { block }

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

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.
- **sendaddr** : this string is a hostname to which UDP control packets can be sent by output widgets, such as switches. By default it is the local host.
- **sendport** : this is the port number for UDP control packets, by default 33333.
- **sendinterval** : the interval between sends of UDP packets for *always send* widgets, by default 2 seconds.

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

1.2 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.)

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

```
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
```

1.3 Window blocks

Window blocks describe the contents of a window. Currently, there is only one window block, and that covers the whole primary display. The contents of a window block are either widgets or frames.

```
window      ::= 'window' '{' {frame|widget} '}'
```

1.4 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
```

1.4.1 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.

```
pos         ::=  x ',' y  ',' w ',' h

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

1.5 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
```

1.5.1 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 "!"

To Do

Oversight – no binary logical operators!

1.5.2 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
            |   '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
```

1.5.3 Number widgets

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

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

1.5.4 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
```

1.5.5 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 map points or vectors to render.

```
map         ::= 'map' pos '{' { mappoint|mapvector } '}'
```

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 a base colour (white by default), a hue clause specifying a source which can be mapped onto the hue range, similar saturation and value sources, a default size, and a size range clause mapping a source onto a size range which replaces the default size. A trail size can also be specified to allow a number of historical points to be rendered.

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

pointmod    ::= 'colour' colour
            |   'hue' source
            |   'saturation' source
            |   'value' source
            |   'size' float
            |   'trail' int
            |   'sizerange' minsize maxsize 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.

```
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

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

1.5.6 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 currently are all of type *floatrange.*

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

Each floatrange indicator consists 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 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'
```

1.5.7 Compass

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

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

1.5.8 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 non-feedback mode, the switch has two states:

- **red** corresponds to 0
- **green** corresponds to 1

In feedback mode, the switch has four states:

- **red** corresponds to 0 sent and acknowledged;
- **green** corresponds to 1 send and acknowledge;
- **grey** 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** means a value has been sent (indicated by the colour) but a feedback value has not yet been received;
- **full crosshatch** 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.

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.

```
switch      ::= 'switch' pos '{'
                    'out' ident
                    'title' string
                     source
                     'always'
                     'immediate'
                '}'
```

2 Other notes

2.1 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.

2.2 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.