Monitor configuration file syntax

James Finnis, jaf18@aber.ac.uk

2013

Contents

1 Blocks

A file consist of blocks, which are either variable blocks or frame blocks. A few other settings are specified at this level:

- the port number to receive data from;
- the port and address to send output data (e.g. from switches) to.

```
file        ::= { block }

block       ::= varblock
            |   frameblock
            |   'port' integer
            |   'sendport' integer
            |   'sendaddr' integer
```

To Do

Port, sendport and sendaddr configuration. Switches.

1.1 Variable blocks

Variable blocks describe the variables which arrive on the UDP connection, and consist of a list of comma-separated variable definitions.

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

1.2 Frame blocks

Frame blocks describe the contents of a predefined frame on the screen. The name of the frame must be one of the frames set up by the C++ code. Each frame block consists of a set of widget blocks, contained by curly brackets.

```
frameblock  ::= 'frame' framename '{' {widget} '}'

framename   ::= ident

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

1.3 Widgets

All widget specification consist of the widget type followed by a widget specification in curly brackets. They always contain a position, and usually at least one data source.

1.3.1 Positions

Widget positions describe where widgets appear in the frame'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         ::= 'pos' x ',' y [ ',' w ',' h ]

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

1.3.2 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!

1.3.3 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    ::=
            |   pos
            |   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
```

1.3.4 Number widgets

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

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

1.3.5 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
```

1.3.6 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
```

1.3.7 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 currently are all of type *floatrange.*

```
status      ::= 'status' '{'
                pos
                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'
```

1.3.8 Compass

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

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

1.4 Output widgets

All widgets so far have been input widgets, showing data sent from the remote system to the monitor.

Output widgets allow UDP packets to be sent back from the monitor to the rover.

Data is sent to the server specified in *sendaddr* and the port specified in *sendport* in the config file, and is in the form of key/value pairs, similar to the data sent from the remote. A single packet contains all output variables due for sending. Variables are due for sending if the time at which they were changed is more recent than the last time they were sent. This is checked every time the state of an output widget is changed, so this will always result in send.

In addition, output variables can be specified as being *always send.* Such variables are periodically sent whether they have changed or not (typically every two seconds or so.) This is in case UDP packets are dropped. **This is currently true for all variables.**

Finally, most output widgets have an optional input source. This allows for confirmation of the receipt of an output. The widget in this case has three states:

- **OK** - the widget shows the state sent, and a packet has been received since the last send in which the designated source matches the output sent.
- **NOACK** - the widget shows "barberpole": a packet has been sent but no packet has been received containing the designated input source.
- **BADACK** - the widget shows "error" (a crosshatch pattern) : a packet has been sent, and a packet has been received from the designated source, but the two values do not agree.

1.4.1 Switch

A switch allows a boolean to be output back to the rover. It acts as a toggle.

```
switch      ::= 'switch' '{'
                pos
                [ source ]
                'out' ident
                [ 'title' string ]
                '}'
```

To Do

insert switch semantics