Trace Messaging Facility

There are two reasons for trace info:
-status reporting during normal operation
-debug tracepoints when troubleshooting unexpected program flow or data values

There are two parts to the generation of traces:

1) Trace commands embedded in the software at strategic places that indicates that a message should be generated if execution gets to this point, and provide the data that goes into the message

2) A trace message generation routine (which the trace command calls) that generates the message based on the information in the trace command, and dynamic configuration information stored in a table. This table can be displayed and overwritten by MQTT commands so that tracing can be changed on the fly

Tracepoint information in the trace statement embedded in the code includes:
- unique verbose location identifier (name of function, possibly with cause text concatenated)
- instance identifier, differentiating multiple traces within one function with same verbose location ID)
- numeric function identifier for table lookup purposes (like Id6 in examples)
- classification of trace data level:
    - top level of generality
    - medium level
    - low level
- nature of trace information
    - S - information on status or key data
    - W - warning of unusual condition
    - E - unforeseen /error condition
- optional text string providing data label or situation description
- optional variable name for data to be provided with the trace

Trace processing controls which are all in one table, writable via MQTT:
- where trace output goes to:
    - serial monitor
    - MQTT topic
    - both
- tracing enabled/disabled for
    - all routines
    - specified routines (using table indexed by numeric function ID)
    - specified generality level: top, medium, low
    - specified type: status, warning, error
    - combinations of the above

Data structure for trace enabling (optimized for fast inline functions)

Table with one entry per tracepoint

-table entries 1 and 2 are special purpose and control global enables and trace message routing respectively. Keeping this info in the table makes it dynamically changeable via a simple MQTT command. Thus tracepoint numbering starts at 3.

- entry 1 is global enables, with individual bits. If bit is one, then applicable traces are enabled:

    - Bit1: all traces are enabled
    - Bit2: all traces with top level
    - Bit3: all traces with medium level
    - Bit4: all traces with low level
    - Bit5: all traces with status info
    - Bit6: all traces with warning info
    - Bit7: all traces with error information
    - Bit8: top + status
    - Bit9: top + warn
    - Bit10: top + error
    - Bit11: med + status
    - Bit12: med + warn
    - Bit13: med + err
    - Bit14: low + status
    - Bit15: low + warn
    - Bit16: low + error

- Entry 2 controls where the trace messages go:
    - Bit1: messages go to serial monitor
    - Bit2: messages go to a MQTT topic
      (you can set both Bit1 and Bit2 at the same time)

- Entry number n controls trace messages for traces with n as their numeric identifier. n starts at 3.
- same bit definitions, applied to traces within the specified function, i.e. with the name numeric identifier

The generated trace message looks like this:

<type> <function-name><trace-sub-ID>(<numeric trace ID>): <data-label-info><data-value>

type = S, W or E
function-name = name of C++ function-name
trace-sub-ID = numeric identifier for traces within same function, starting at 1
numeric trace ID = assigned number for this function name, globally unique. Indexes the config table.
data-label-info = label for acompanying data value, or description of cause of trace message
data-value = debug or status data added to the message

example message: E functionB-1(7): g_X out of range: 18.5

Example code sequence:

```
void loop()
{
  functionA(10,20,30)
  functionB(5.7, 7.9, -10.60)
}


void functionA(x,y,z)
{
    trace(“functionA”, 1, ID6, top, status, “Z value= “, Z)
    float r = sqrt(x*x + y*y +z*z)
    trace(“functionA”, 2, ID6, low, status, “R= “,r)
    g_det = B * B – 4 * a * c    // from quadratic formula
    if (g_det<0) {trace(“functionA”, 3, ID6, top, error, “negative determinant= “, g_det)}
    return
}


void functionB(hipx, hipy, hipz)
{
    trace(“functionB”, 1, ID7, top, status, “starting”, hipx)
    g_X = arcsin( hipy/hipx) * g_hfr_width
    if( g_X > maxW || g_X < maxW)
    {
        trace(“functionB, 1, ID7, error, “g_X out of range: “,g_X)
        // recover in some way...
        g_X = 0
    }
    return
}
```

Example output on next page.

```
with this table setup:                | you would see traces like this for one loop()
                                      |
[1]  0    /no global enables          |  S functionA-1(6): Z value= 30
 ...                                  |  S functionA-2(6): R= Z value= 37.42
[6] Bit1                              |     / never encountered functionA-3 tracepoint
[7]                                   |
```

```
with this table setup:                | you would see traces like this for one loop()
                                      |
[1]  Bit1     /all traces enabled     |  S functionA-1(6): Z value= 30
 ...                                  |  S functionA-2(6): R= Z value= 37.42
[6]  *        /don't care             |  E functionB-1(7): g_X out of range: 18.5
[7]  *        /don't care             |
```

```
with this table setup:                | you would see traces like this for one loop()
                                      |
[1]  0                                |
 ...                                  |
[6]  *                                |  E functionB-1(7): g_X out of range: 18.5
[7] Bit10 + Bit13 +Bit16              |                          |
 /top+err, med+err, low+err
```

```
with this table setup:                | you would see traces like this for one loop()
                                      |
[1]  Bit7    /all traces with errors  |
 ...                                  |  E functionA-3(6): negative determinant= -87.2242
[6]  *        /don't care             |  E functionB-1(7): g_X out of range: 18.5
[7]  *        /don't care             |
```

Notes:
- this facility will distort program execution timing because it takes time to generate and transmit the trace message. It would be nice to extend it to allow setting / clearing of a small number of LEDs via a trace command, which can be done very quickly.
- the requirement to not require any arguments has not been met, and it's not clear to me how this would be possible. The argument count can be reduced by defining specialized macro's that essentially pre-populate some of the fields.
- there's some implied work to write 2 new MQTT commands:
  - one to write numeric values into the configuration table for on-the-fly trace changes
  - one to display the current configuration table, for convenience
- there's some implied work to allow the config table to be set up at compile time with a usable setup.
- this trace facility isn't a replacement for the abbreviated print commands like sp2l(a,b) which have a different purpose, primarily in debugging.
- we could assign bits in the config table to direct individual trace messages to the serial monitor, MQTT, or both, but I'm not sure this is needed.