Trace Messaging Facility

Table of Contents

Introduction

Traces are dynamically generated messages that convey information about the status and execution of the robot's software. There are two general reasons to generate a trace:

1. status reporting during normal operation
2. debug tracepoints when troubleshooting unexpected program flow, performance or data values

This document describes a method of generating traces created for development of HexaFloorRide, a 6 legged robot. Features of this system are:

- ability to capture and display the context where the trace was generated, including:
  - the name of the folder and source file containing the trace
  - the name of the C++ function containing the trace
  - the line number within the source file for the trace
- ability to include a descriptive text string, and the contents of a variable in the trace messages
- ability to direct trace messages to the serial monitor, the MQTT health topic, or both
- ability to independently handle 5 different types of traces
- ability to selectively enable each type of trace on a global basis
- ability to selectively enable each type of trace within a specified C++ function
- ability to modify trace enabling dynamically at run time via MQTT commands
- simplified trace insertion by automatic pre-population of much of the trace information

Information In A Trace

The information that can be displayed in a trace message includes the following:

1. The name of the source file containing the tracepoint, optionally including the name of its folder
2. The name of the function that contained the tracepoint
3. The line number within the source file where the tracepoint is located
4. A numeric ID uniquely associated with the function, used for control purposes
5. A bit mask identifying the type of trace, used for control purposes
6. A text string describing the reason for generating the trace
7. An optional variable (String, int or float) to be appended to the trace message.

There are 20 trace macros that simplify trace definition, and automatically fill in much of this information

Examining these pieces of information in more detail:

1. The name of the source file containing the tracepoint, optionally including the name of its folder
   - This comes from the compiler's automatic macro called __FILE__
   - It contains the name of folder that contains the source file, such as lib, include, or src
   - There are options to format the trace message, and some of them suppress the folder name
   - Another trace formatting option suppresses the file information altogether

2. The name of the function that contained the tracepoint
   - This comes from the compiler's automatic macro called __func__
   - Example: a function starting with void identifyDev(int address) would have the name identifyDev
   - This name is always present in the trace message

3. The line number within the source file where the tracepoint is located
   - This comes from the compiler's automatic macro called __LINE__
   - This line number is relative to the start of the source file, not the start of the function
   - The trace message can be formatted to suppress the line number

4. A numeric ID uniquely associated with the function, used for control purposes
   - This is a requirement for the dynamic control of trace enabling, which is table driven
   - The macros get this value from the symbol localRNum
   - localRNum must be defined at the beginning of any routine that has embedded tracepoints
   - The process for setting this up is described later in the section "Inserting a tracepoint"

5. A bit mask identifying the type of trace, used for control purposes
   - There are 5 different types of traces, described in the following section
   - Bit masks with symbolic names like t$E for error messages are used to represent the trace type
   - There is a dynamic mechanism to enable/disable traces based on their type and their routine.

6. A text string describing the reason for generating the trace
   - This is present in every trace message
   - It is usually a string literal as the first argument of the trace macro

7. An optional variable (String, int or float type) to be appended to the trace message.
   - The current value of a variable is often an important part of a trace message
   - The predefined macros make it simple to append a String, int or float variable to a trace message

Trace Types

There are 5 types of traces. Trace messages start with a letter that identifies the trace type:

Trace Type	Usage	Symbol	Mask
H	High level information or status. This is summarized information on general topics that is generated as part of normal operation. An example might be: Hardware initialization completed, with anomaly count: 3	t$H	1
M	Medium level information or status. Summarized information with slightly more granularity and details than H messages. Example: I2C buses initialized. Detected devices = 5	t$M	2
L	Low level information or status. Very detailed information normally used for debugging or troubleshooting performance problems. Example: I2C bus scan found device: bus=1, addr= 41, devtype= 17	t$L	4
W	Warning. An unusual or unexpected situation has been detected, but the software is able to continue operating. Example: CPU Utilization 5 second average percent = 91	t$W	8
E	Error. An unusual or unexpected situation has occurred which compromises the robot's ability to continue operating properly. Defensive action might be required for safety. Example: Leg position calculation caused a divide by zero error	t$E	16

The Trace Macros

Although each trace provides the 7 pieces of information described above, it is not necessary to specify all this information in the macro that defines the tracepoint. Some of the information (source file, function and line number) is taken from automatically generated compiler macros. Some of the information is implied by variations in the name of the macro used. The only arguments that are needed for the macro are the descriptive text string, and optionally the name of a variable to be displayed at the end of the message.

There are 5 basic macros, corresponding to the 5 types of traces, that are used to enter a tracepoint. The names of these macros are the word "trace" with the type of macro appended as an upper case letter. The argument for the macro is the descriptive text, which is usually a text literal. Here are some examples:

Putting this macro in the code	...would generate a trace message like this
traceH("No class 1 faults in 60 sec");	H) include/perfMon.cpp\review60{21}-L198> No class 1 faults in 60 sec
traceM("max memory normal");	M) include/perfMon.cpp\heapMon{34}-L52> max memory normal
traceL"("sensor saw wall");	L) include/navigate.cpp\eyeScan{63}-L126> sensor saw wall
traceW("task queue depth red");	W) include/navigate.cpp\qManager{44}-L216> task queue depth red
traceE("avoid wall failed *****");	E) include/navigate.cpp\sensorCheck{93}-L93> avoid wall failed *****

(We'll examine the layout of a trace message in the following section.)

For the case where you want to append the value of a variable to the trace message, there are 2 additions to this scheme:

1. You append a lower case letter to the macro name to identify the type of variable:
   - i is used for an integer
   - f is used for a floating point number
   - s is used for a string

2. You add the name of the variable as the second argument for the macro

This source code	...would generate a trace message like this
traceHi("Max job queue in 60 sec=",jmax);	H) include/perfMon.cpp\review60{21}-L198> Max job queue in 60 sec= 17
traceMf("max memory % =",memMax);	M) include/perfMon.cpp\heapMon{34}-L52> max memory % = 38.64
traceLs"("sensor saw wall-",sensorName);	L) include/navigate.cpp\eyeScan{63}-L126> sensor saw wall- left eye

The Layout Of Trace Messages

Sample Trace Message:

```
M) include/perfMon.cpp\heapMon{34}-L52> max memory %= 38.64
```

Trace messages are built up from the following pieces, in order of appearance

- An upper case letter identifying the trace type (H, M, L, W or E)
- a right parenthesis that confirms this is a trace message
- the source location of the trace (described in detail below)
- the unique numeric identifier for the function, in brace brackets (useful to configure trace enabling)
- "-L" to prefix the line number, if it's present
- the optional source code line number relative to the start of the file
- "> " for readability
- the descriptive text string included in the trace macro that generated the trace
- optionally, the value of the variable that was specified in the trace macro

The format of the information in the trace message identifying the source location is variable, and is controlled by the following 4 parameters in the global_variables.cpp file, which have names reflecting the fields included:

```
t$SFolderFileFuncLine
t$SFileFuncLine
t$SFuncLine
t$SFunc
```

1) If t$SFolderFileFuncLine is defined (i.e. this line's uncommented in global_variables.cpp) the source looks like:

source-folder / source-file \ function-name {function ID} – L line-number trace-text variable-value

Note the use of slash / and backslash \ as separators to improve readability.

Example macro call: `traceMf("max memory % =",memMax);`

Resulting trace message: `M) include/perfMon.cpp\heapMon{34}-L52> max memory % = 38.64`

2) If t$SFileFuncLine is defined (i.e. this line is uncommented in global_variables.cpp) the source location looks like:

source-file \ function-name {function ID} – L line-number trace-text variable-value

Example macro call: `traceMf("max memory % =",memMax);`

Resulting trace message: `M) perfMon.cpp\heapMon{34}-L52> max memory % = 38.64`

3) If t$SFuncLine is defined (i.e. this line is uncommented in global_variables.cpp) the source location looks like:

function-name {function ID} – L line-number trace-text variable-value

Example macro call: `traceMf("max memory % =",memMax);`

Resulting trace message: `M) heapMon{34}-L52> max memory % = 38.64`

4) If t$SFunc is defined (i.e. this line is uncommented in global_variables.cpp) the source location looks like:

function-name {function ID} trace-text variable-value

Example macro call: `traceMf("max memory % =",memMax);`

Resulting trace message: `M) heapMon{34}> max memory % = 38.64`

Only one of these formatting options should be selected (i.e. uncommented). If more that one are enabled, the last one will override previous ones.

Enabling And Disabling Trace Messages

What trace information you want to see depends on what kind of activity you're doing.

You may be doing an activity that doesn't make use of trace information, but you'd like to know if something unusual happens. For example, if you are evaluating the operation of a new script that causes the robot to move with an unusual gait, you're probably focused on the movement and appearance of the robot. Trace info probably isn't relevant, unless something goes wrong, perhaps because a bug is encountered. In this case you might want apply a universal policy of generating Error and Warning traces only, for all defined traces. We call this global enables.

If you're troubleshooting one particular function, you likely want to see most or all the possible trace information for it, and perhaps a few other related functions. However, you don't want to see this level of detail for all functions, because the large volume of generated traces would obscure the traces that are relevant to the task at hand. Thus it is desirable to enable or disable tracing for certain individual functions, and perhaps even control the types of traces that are enabled. We call this per-function trace enables.

As you progress through troubleshooting an issue, you may discover that you need additional trace information from a function that you didn't anticipate being involved in your investigation. The dynamic control feature of tracing allows you to change the trace enables, both global and per-function on the fly. This is done be sending an MQTT command (TR) that modifies the table in memory that controls trace enabling. This table is the control structure for trace enables, and we'll describe it more detail.

The Trace Table

The trace table $traceTab is located in global_variables.com. It has an entry for every function that has a trace defined. Note that there is an entry per function, not per trace. There is a unique identifier for each function which is used as an index into this table to access the enable information for all traces in that function. The enable information is represented by bit masks, one for each trace type:

Trace Type	Mask Symbol	Mask in decimal
High	t$H	1
Medium	t$M	2
Low	t$L	4
Warning	t$W	8
Error	t$E	16

This table is initialized in the function setupTracing in configDetail.cpp. The entry for a particular function is expressed as the sum of the enabled trace types. For example, to set up the function forwardTilt() with the numeric identifier 14 to enable Warning and Error traces only, you would have the statement:

```
$traceTab[14]  = t$W + t$E          // routine 14 = forwardTilt in flows.cpp
```

It is important to include the function name and source file in the comment, to document the assignment of numeric identifiers to functions. Otherwise you'd need to search through all source files to find where a particular number is assigned. More on this assignment process in the next section.

There are 2 special entries in this table that don't actually represent function enables. The first one is index 1, with the symbol t$global. This controls the overall enables that apply to all functions. This can be regarded as the default enabling for traces in function that don't have any enables in their own entry. An example definition of this entry is:

```
$traceTab[t$global] = t$H + t$M + t$E + t$W ;  // Globally enable High, Medium, Error, and Warning messages
```

The second special entry is index 2, symbol t$routing. This is also bit encoded, and determines where trace messages are sent:

```
t$SM       // means send traces to the Serial Monitor
t$MQ       // means send races as MQTT messages to the health topic
```

Note that it is possible to set both bits, and have trace messages go to both destinations. Example definition:

```
$traceTab[t$routing] = t$SM          // traces messages go to the Serial Monitor = console
```

Creating a Tracepoint

There are 2 parts to inserting a trace into a function:

1. Setting up the function for tracing (which is done only once)
2. Inserting the trace macro at the appropriate place in the function.

Setting Up A Function for tracing

If no traces have been installed in your function before, you will need to create an entry in the trace table, and put 2 lines of setup code at the beginning of the function. You can tell if this has already been done by checking to see if the first lines in your function are the #undef and #define statements described below. If they are already present, you can skip ahead to the “Inserting the Trace Macro” section.

To create an entry in the trace table for your function, do the following:

- note the function name, and the source file that contains it.
- Open configDetails.cpp and find the trace table setup by searching for t$global
- go to the end of the trace table entries, and create a new one, using the next available index value
- include the enable bits you want, and comments that document the function name and source file
- note the table index (= function number) that you just allocated. We’ll refer to this as “n” below
- open the source file that contains your function, and add these 2 lines at the beginning of your function:

  ```
  #undef localRNum
  #define localRNum n
  ```

  The #undef avoids compiler warnings that you’re redefining the symbol localRNum (local Routine number)
  The #define creates a symbol that the macros use to automatically embed the function ID in trace macros

Inserting the Trace Macro

Once you’ve done the above step, or if it has already been done, all that’s left is adding the trace macro call at the appropriate place in your code. To do this, you’ll need to decide on 3 things:

- what type of trace you want, H, M, L, W or E. This determines the base macro name: traceH, traceM ....
- what descriptive text you want to include, which will be the first argument for the macro.
  Example: traceL(“reached recalibrate function”);
- if you want to add a variable to the trace message, which will add a 2nd argument and extend the macro name
  Example: traceLi(“started queuing messages. Depth: “, qDepth);

If you want to include more information in your trace, such as more variables, you can add code before the tracepoint that creates a string containing all the desired details, and output it as a single string:

```
String rstatus= String(sColour[RED].redDC) +","+ String(sColour[RED].greenDC) +","+String(sColour[RED].blueDC);
traceLs("Composition for Red:","RGB= "+ rstatus);
```

Interaction Between Global and Per Function Tracing

There are 2 ways in which tracing for a particular trace type can be enabled.

One is via the global enables which are configured in the first entry in the trace table, $traceTab[t$global]

The second is via the enable bits in the trace table entry that is dedicated to a particular function.

The trace logic will generate a trace message if the trace is enabled in either of these locations for the trace type of the tracepoint macro.

From a logic point of view we can express trace generation as a boolean combination of these values:

- The global trace enables, which can have between 0 and 5 bits set
- The containing function’s enables, which can have between 0 and 5 bits set
- The tracepoint macro’s type identification, which will have exactly one bit set

A trace will be generated if the following expression is non-zero:

(global-enables OR function-enables) AND (macro-type)

It is desirable that tracing have minimal impact on program execution timing, to avoid impact on performance issues. If a trace is generated, there has to be some impact because of the resources used to create the trace message. However, we want to minimize the overhead when no traces are actually generated.

The decision on whether or not to generate a trace is purely based on the boolean logic above, and is coded:

```
if(($traceTab[t$global] | $traceTab[functionID]) & tType) != 0)
```

The logic for all 5 trace types is done in parallel by the logic operators. As well, the compiler should be able to optimize out the indexing for t$global, since it is a constant. We're hopeful that the instructions generated for this are minimal. To reduce tracing impact on performance, we could reduce it to setting and clearing some diagnostic LEDs, which can be done quickly. However this would reduce the available tracing functionality dramatically.

Dynamic Control of Trace Enabling

As described above, all trace enabling, both global (trace table index 1) and for individual functions (trace table indices 3 and higher) is controlled by bit values in the trace table. As well, the routing of messages to the serial monitor or to MQTT is controlled by trace table index 2.

All of these control functions can be modified at run time by means of an MQTT command, TR, which allows you to overwrite an entry in the trace table. This command has the form:

TR, index, new-value

where:
TR is the command name representing Tracetable
index is a decimal integer from 1 to 100, representing the index for the table entry to be changed
new-value is a decimal integer from 1 to 31 representing the decimal equivalent of the combined enable bit masks

If index is zero, or if the command is TR or TR, a display of all non-zero trace table entries is returned as routable trace output. The code that generates it is:

```
#define t$SFunc 1                    // just function name <id-number>
traceM("Trace Control table entries, in form: index, decimal value");
char buffer [50];
for(int ti = 1; ti < int(maxTraceCount); ti++ )    // step thru trace control table
{  if($traceTab[ti] != 0 )           // skip over boring zero entries
   {  int t = sprintf(buffer, "%3d  %3d",ti,$traceTab[ti]) ;
      traceM(buffer);
   }
}
```

Here's a sample of the Trace Table display output:

```
M) processCmd{6}> Trace Control table entries, in form: index, decimal value
M) processCmd{6}>   1   27
M) processCmd{6}>   2    1
M) processCmd{6}>   3   26
M) processCmd{6}>   4   24
M) processCmd{6}>   5   24
M) processCmd{6}>   6    2
M) processCmd{6}>   7   27
M) processCmd{6}>   8   27
M) processCmd{6}>   9   31
M) processCmd{6}>  10   31
M) processCmd{6}>  11   31
```

Here's a sample of the last line of the Trace Table display output as it appears in MQTT:

```
agingApprentice/HexaFloorRide3C61054ADD98/health M) processCmd{6}>  11   31
```