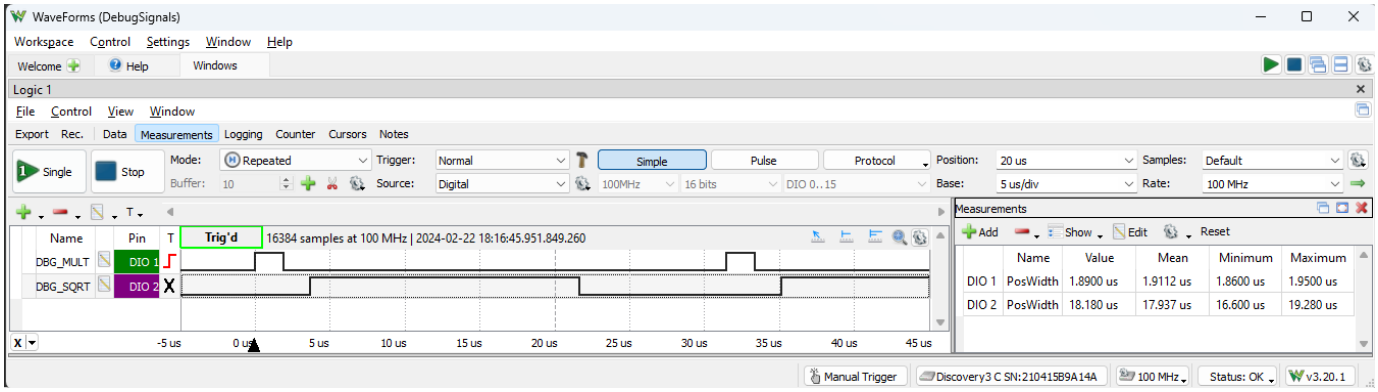
VISUALIZING CODE TIMING WITH USER-DEFINED SIGNALS

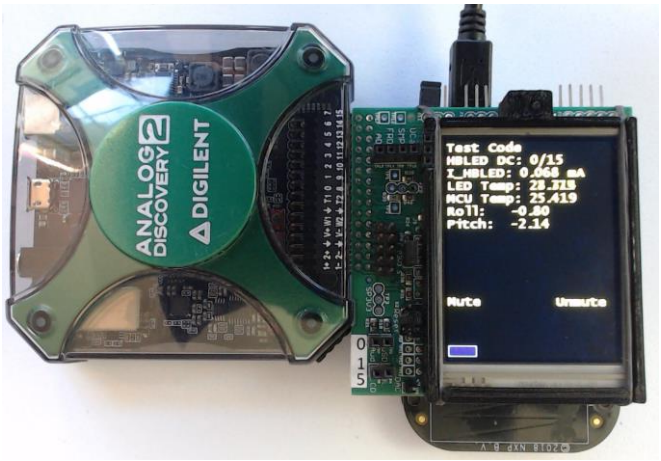
OVERVIEW

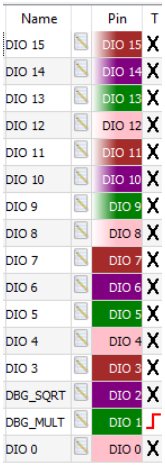
We may need to understand the timing of software. Here we look at how long it takes to perform two different floating-point math operations with slowly changing input data. There are other possible timing questions: When does a function start and stop relative to other functions? When does an ISR run? How much time elapses between an interrupt request and the ISR starting? We can use the logic analyzer and debug signals to see the timing characteristics of these events.



The files debug.c and debug.h provide debug signal generation. The following table shows how each DBG_ signal is allocated to a specific MCU port bit and Analog Discovery DIO signal when plugged into the Analog Discovery with the connector as shown. (Of course, you'll need to plug a USB cable into the Analog Discovery to use it.)

Signal Name in Code	MCU Port Bit	AD2 DIO Signal
DBG_0	D0	DIO 0
DBG_1	D2	DIO 1
DBG_2	D3	DIO 2
DBG_3	D4	DIO 3
DBG_4	B8	DIO 4
DBG_5	B9	DIO 5
DBG_6	B10	DIO 6
DBG_7	B11	DIO 7





EXAMPLE PROJECT: DEBUGSIGNALS

The DebugSignals project (available on Github in the class repository in the Tools\TestCode\DebugSignals directory is used here to demonstrate the concepts and methods. It contains debug.c, debug.h, main.c, and several other files.

DEBUG.H

We don't have to change anything within the debug.h file. However, it will make the code easier to read and maintain if we define more descriptive names for the debug signals. Here we added #defines so that DBG_MULT is DBG_1, and DBG_SQRT is DBG_2.

```
#ifndef DEBUG_H
#define DEBUG_H

#include <stdint.h>
#include <MKL25Z4.H>

#define MASK(x) (1UL << (x))

#define DEBUG_INIT_TEST 0 // Set to 1 to enable testing debug signal in initialization

extern const uint32_t DBG_Bit[9];
extern const FGPIO_MemMapPtr DBG_PT[9];

#define DBG_0 0
#define DBG_1 1
#define DBG_2 2
#define DBG_3 3
#define DBG_4 4
#define DBG_5 5
#define DBG_6 6
#define DBG_7 7
#define DBG_NULL 8 // no effect: mapped in debug.c to a non-GPIO bit (on a used port)

// Define meaningful names for debug signals
#define DBG_MULT    DBG_1
#define DBG_SQRT    DBG_2

// Debug signal operation macros
#define DEBUG_START(x)  {DBG_PT[x]->PSOR = MASK(DBG_Bit[x]);}
#define DEBUG_STOP(x)   {DBG_PT[x]->PCOR = MASK(DBG_Bit[x]);}
#define DEBUG_TOGGLE(x) {DBG_PT[x]->PTOR = MASK(DBG_Bit[x]);}

void Init_Debug_Signals(void);

#endif // DEBUG_H
```

MAIN.C

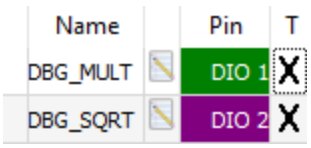
Within the main.c file, we #include the debug.h file. The main() function initializes the debugging support (Init_Debug_Signals()) and the RGB LED outputs (Init_RGB_LEDs()) on the FRDM-KL25Z. The main() function then repeats a loop performing floating point multiply and square root operations. Note how these operations are bracketed with DEBUG_START and DEBUG_STOP macro calls with a debug signal name (DBG_MULT or DBG_SQRT) as the argument.

```
#include <MKL25Z4.H>
#include <math.h>
#include "debug.h"

/*----------------------------------------------------------
  MAIN function
 *----------------------------------------------------------
int main (void) {
  volatile float x, y, z;

  Init_Debug_Signals();
  Init_RGB_LEDs();

  Control_RGB_LEDs(1,0,0);       // Red - starting up

  x = y = 0.1;

  while (1) {
    Control_RGB_LEDs(1,1,0);       // Yellow - multiply
    DEBUG_START(DBG_MULT);
    z = x * y;
    DEBUG_STOP(DBG_MULT);
    Control_RGB_LEDs(0,0,1);       // Blue - square root
    DEBUG_START(DBG_SQRT);
    z = sqrt(z);
    DEBUG_STOP(DBG_SQRT);
    x += 0.001;
    y += 0.03;
  }
}
```

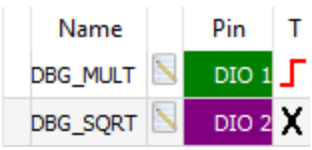
Build, download and run the program. The LED should look lavender, but it is actually switching between quickly between yellow and blue.

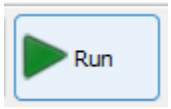
VIEWING DEBUG SIGNALS WITH WAVEFORMS

- Connect the Analog Discovery to the FRDM/shield stack. On your PC start the Waveforms program and open the DebugSignals.dwf3work workspace. The workspace has been set up for your convenience.
- Set the logic analyzer to trigger on the rising edge of DBG_MULT (DIO 1) by clicking in the cell in row DIO 1 and column T and selecting Rise.

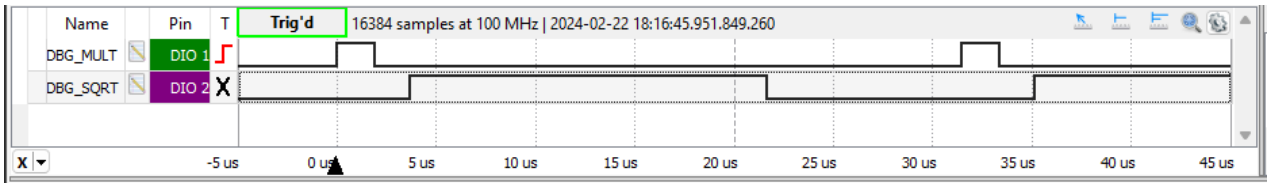


- It should now look like this:



- Make sure the program is running on the MCU board.
- Press Run in Waveforms. 

- You should see a plot like this:

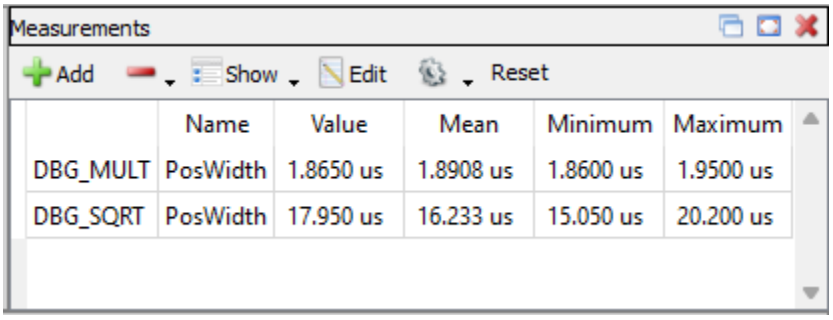


- You may need to adjust the time base (horizontal sensitivity) to zoom in or out. This control is located near the upper right corner: 

- To add more signals, press the green + to add the debug signals based on which DIO signal they are connected through.

AUTOMATIC MEASUREMENTS

The Waveforms scope tool can measure the input channel characteristics over multiple acquisitions and compute statistics for you automatically. These have already been set up in the demonstration workspace for the DBG_MULT and DBG_SQRT signals. This screen shot shows the sqrt operation is much slower than the multiplication, and also has a wider range of execution times.



	Name	Value	Mean	Minimum	Maximum
DBG_MULT	PosWidth	1.8650 us	1.8908 us	1.8600 us	1.9500 us
DBG_SQRT	PosWidth	17.950 us	16.233 us	15.050 us	20.200 us

Note that the time measurements are limited to the maximum time range shown on the horizontal axis (50 us in this case). You may need to increase the time per division as the scope is running until the maximum stops increasing.

If you need to add more measurements, follow these steps:

- In the Logic 1 menu, select Measurements. This will show or hide the Measurements window.
- Click the + Add button to get the Add measurement dialog box. Select the digital signal to measure and select PosWidth (width of positive pulse). Click Add and then click Close.
- Click the Show button and check the Average, Minimum and Maximum entries.
- Click the gear icon and check the Multiple Acquisitions box so the statistics are not reset with each new acquisistion.

CODE MODIFICATIONS FOR AN EXISTING PROJECT

- Files
  - If not present in the project directory or a subdirectory, copy debug.c and debug.h in.
- Project Settings
  - If debug.c is not already in the Project Source code file list, add it.
- debug.h
  - #define descriptive names for debug signals, assigning them to DBG_0 through DBG_7
- main.c
  - Add #include “debug.h”
  - In the function main(), add a call to Init_Debug_Signals().
- Any .c files where you’ll start or stop the debug signals
  - Add #include “debug.h”
  - For each timing event to monitor…
    - Identify the location in the code where the timing event starts. Just before that location, add a macro to raise the output signal to 1: DEBUG_START(DBG_MY_SIGNAL_POS)
    - Identify the location in the code where the timing event ends. Just after that location, add a macro to lower the output signal to 0: DEBUG_STOP(DBG_MY_SIGNAL_POS)
  - Note that there is also a DEBUG_TOGGLE macro available to invert the output signal (i.e. change 1 to 0, and 0 to 1). This may be useful in some cases.