Università
della
Svizzera
italiana

Facoltà
di
scienze
informatiche

Edge Computing in the IoT

Programming Embedded Systems

Alberto Ferrante (alberto.ferrante@usi.ch)

What's Specific About Embedded Programming?

- Usually, programming languages are the same ones used for general purpose systems
- Usually, embedded software cannot be run on different embedded systems without (significant) modifications
  - Incredible variety of hardware in use in embedded systems, with variety of sensors and interfaces
  - Use of standardized development platforms and standardized connectors help in increasing portability
    - Still, code often needs at least some manual tuning

What's Specific About Embedded Programming?

- We need to pay more attention to
  - Memory usage
  - Performance
  - I/O
  - Energy/Power

What's Specific About Embedded Programming?

- We often work at low level
  - We may need to know about hardware
    - To actually use it
      - What's the input pin that I should use for the temperature sensor?
      - How should I configure the sensor to obtain the desired result?
      - ...
    - To be able to exploit it fully
      - Limited resources call for optimizations

ES Programming - Memory

- Usually limited: we may need to optimize its usage
  - Be aware of data types and use them efficiently
    - e.g., do not use 32-bit variables if 8 bits are enough to represent your data!
  - Use constants when suitable
    - e.g., if you need to use Pi all over your code, define pi=3.14 as a constant rather than a floating point variable
  - Write memory-efficient code
    - There might be different versions of algorithms, that use memory in different ways
    - Find the right trade-off between code efficiency and memory occupation
  - Exploit hardware optimizations
    - e.g. ARM Thumb instructions
- Memory access is costly in therms of energy

ES Programming - Performance

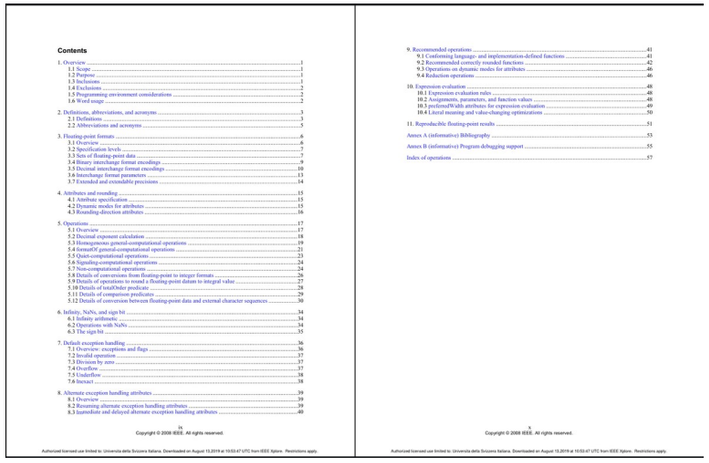
- Know your hardware
  - No. of cores
  - Pipeline
  - Memory system (caches and organization)
  - Accelerators
    - e.g., for cryptography

ES Programming - Performance

- Know your algorithm
  - Event or data driven?
  - Which system resources does it use?
    - e.g., floating point multiplications vs. integer sums
  - Can it be implemented in a different way?
- Hardware / software interactions
  - e.g., my hardware does not have a floating point unit, but my algorithm uses floats

Floating Point Representation of Numbers

- Floating-point arithmetic use formulaic representation of real numbers
  - An approximation so as to support a trade-off between range and precision
  - Can be used to represent numbers of different orders of magnitude by using a fixed number of digits
- IEEE 754-2008: a standard specifies interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments


Contents

1. Overview 1
1.1 Scope 1
1.2 Purpose 1
1.3 Inclusions 1
1.4 Exclusions 2
1.5 Programming environment considerations 2
1.6 Word usage 2
2. Definitions, abbreviations, and acronyms 3
2.1 Definitions 3
2.2 Abbreviations and acronyms 5
3. Floating-point formats 6
3.1 Overview 6
3.2 Specification levels 7
3.3 Sets of floating-point data 7
3.4 Binary interchange format encodings 9
3.5 Decimal interchange format encodings 10
3.6 Interchange format parameters 13
3.7 Extended and extendable precisions 14
4. Attributes and rounding 15
4.1 Attribute specification 15
4.2 Dynamic modes for attributes 15
4.3 Rounding-direction attributes 16
5. Operations 17
5.1 Overview 17
5.2 Decimal exponent calculation 18
5.3 Homogeneous general-computational operations 19
5.4 formatOf general-computational operations 21
5.5 Quiet-computational operations 23
5.6 Signaling-computational operations 24
5.7 Decimal re-encoding operations 24
5.8 Details of conversions from floating-point to integer formats 26
5.9 Details of operations to round a floating-point datum to integral value 27
5.10 Details of totalOrder predicate 28
5.11 Details of comparison predicates 29
5.12 Details of conversion between floating-point data and external character sequences 30
6. Infinity, NaNs, and sign bit 34
6.1 Infinity arithmetic 34
6.2 Operations with NaNs 34
6.3 The sign bit 35
7. Default exception handling 36
7.1 Overview: exceptions and flags 36
7.2 Invalid operation 37
7.3 Division by zero 37
7.4 Overflow 37
7.5 Underflow 38
7.6 Inexact 38
8. Alternate exception handling attributes 39
8.1 Overview 39
8.2 Resuming alternate exception handling attributes 39
8.3 Immediate and delayed alternate exception handling attributes 40
9. Recommended operations 41
9.1 Conforming language- and implementation-defined functions 41
9.2 Recommended correctly rounded functions 42
9.3 Operations on dynamic modes for attributes 46
9.4 Reduction operations 46
10. Expression evaluation 48
10.1 Expression evaluation rules 48
10.2 Assignments, parameters, and function values 48
10.3 preferredWidth attributes for expression evaluation 49
10.4 Literal meaning and value-changing optimizations 50
11. Reproducible floating-point results 51
Annex A (informative) Bibliography 53
Annex B (informative) Program debugging support 55
Index of operations 57

https://en.wikipedia.org/wiki/Floating-point_arithmetic

https://ieeexplore.ieee.org/document/4610935

Floating Point Representation of Numbers

- Numbers have, in general, an approximate representation
  - To a fixed number of significant digits (the significand)
    - Determines the maximum precision
  - Scaled using an exponent in some fixed base
    - Normally two, ten, or sixteen
    - Determines the range
- Single precision, "float" type in C:
  - 32 bits, 24-bit significand
- Double precision, "double" type in the C:
  - 64 bits, 53-bits significand

$$\text{significand} \times \text{base}^{\text{exponent}},$$

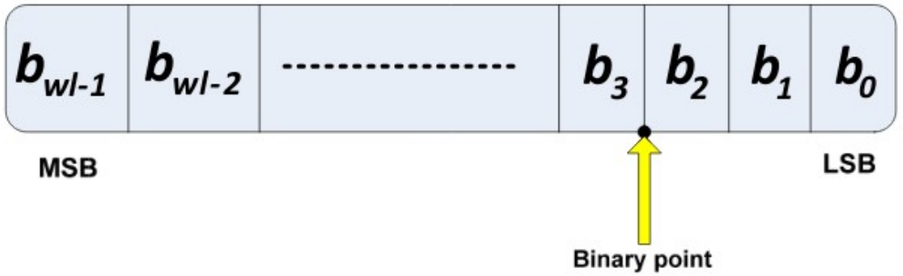
$$1.2345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{\overbrace{-4}^{\text{exponent}}}}_{\text{base}}.$$

https://en.wikipedia.org/wiki/Floating-point_arithmetic

https://ieeexplore.ieee.org/document/4610935

Fixed Point Representation of Numbers

- Our microprocessor / microcontroller may not have a floating point unit
- The FPU is energy hungry
- Working with floats may be slower
- Saves memory in some cases



- Floating point to Fixed-point conversion
  - May be problematic when representing numbers with very different orders of magnitude
  - Can introduce rounding errors that might have an impact on the algorithms
    - There are ways to mitigate this effect, by adjusting algorithms (e.g., quantization-aware training for machine learning algorithms)

http://ee.sharif.edu/~digitalvlsi/Docs/Fixed-Point.pdf

ES Programming - I/O

- Input/Output is one of the key aspects of embedded systems
  - Some devices (sensors and actuators) have no complete drivers or no drivers at all
    - e.g., an analog temperature sensor
  - We may need to map the I/O (calibrate)
    - e.g, we may read 545 for our temperature sensor, this does not mean that the temperature is 545°C
  - Different I/O buses, we may need to manage them
  - How do we manage the I/O?
    - Polling vs. interrupts

Software Libraries

- Software libraries: a collection of non-volatile resources used by computer programs
  - They define classes, functions, constants, and variables defined with the purpose of reusing code
  - They might include:
    - Common functions (e.g., conversion between integers and strings)
    - Functions for managing specific devices/subsystems (e.g., libraries for the I^2C bus)
- The use of libraries greatly simplifies writing embedded applications
  - We can use the library provided by the producer of our sensor instead of writing all the code from scratch
    - Provided that a specific library works on the platform of choice …
- Libraries increase portability
  - They may provide a common interface to different implementations
    - e.g., the function read_sensor() may be implemented differently on different platforms; still, the code using this function will not require any modification to be ported from one platform to the other
- This concept is used in modern IDEs that support multiple platforms

Software Libraries Example: Arduino

```
void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as output
}


void loop() {
  digitalWrite(13, HIGH); // sets the digital pin 13 on
  delay(1000);            // waits for a second
  digitalWrite(13, LOW);  // sets the digital pin 13 off
  delay(1000);            // waits for a second
}
```

```
void digitalWrite(uint8_t pin, uint8_t val)
        uint8_t timer = digitalPinToTimer(pin);
        uint8_t bit = digitalPinToBitMask(pin);
        uint8_t port = digitalPinToPort(pin);
        volatile uint8_t *out;
        if (port == NOT_A_PIN) return;

        // If the pin that support PWM output, we need to turn
it off
        // before doing a digital write.
        if (timer != NOT_ON_TIMER) turnOffPWM(timer);
        out = portOutputRegister(port);
        uint8_t oldSREG = SREG;
        cli();
        if (val == LOW) {
                *out &= ~bit;
        } else {
                *out |= bit;
        }

        SREG = oldSREG;
}
```

The Infinite Loop

- Most embedded systems run only one piece of software
  - If the software stops running, the hardware is rendered useless
- Embedded programs almost always have an infinite loop
- Typically, this loop surrounds a significant part of the program's functionality

```
#include <Arduino.h>
int main(void)
{
    init();
#if defined(USBCON)
    USBDevice.attach();
#endif

    setup();

    for (;;) {
        loop();
        if (serialEventRun) serialEventRun();
    }

    return 0;
}
```

Programming Embedded Systems, 2nd Edition, Anthony Massa, Michael Barr
https://www.oreilly.com/library/view/programming-embedded-systems/0596009836/ch01.html

Cross-compilation

- Cross compilation is about building code for a **target** architecture, by using a machine with a different architecture as a **host**
  - e.g., development and project build is performed on a PC with an x86 architecture; the generated machine code is for an ARM v7
  - User friendly platforms hide cross-compilation to the programmer
- Required as, in most of the cases, we cannot build directly on the target architecture
  - It might be problematic to program on a machine with no display, no keyboard and no mouse, and that might even not be able to run a compiler (or it would be too slow)

Cross-compilation

- Not only the compiler, but a whole toolchain is required
  - A set of programming tools that is used to perform a complex software development task or to create a software product
    - Compiler
    - Linker
    - Libraries
    - Other utilities
  - The GNU Toolchain is an example of a cross-compilation capable toolchain
- Depending on the type of system considered, you may need to build just your application or your a full operating system
  - If there is an OS, somebody, at some point, needs to build it for your platform

Cross-compilation

- Obtaining a cross-compilation toolchain (gcc):
  - Who builds the toolchain? Let’s suppose that our host architecture is x86 and our target is ARM v7
    - We start from a working toolchain for x86 (hot toolchain, HT)
    - By using he HT, we build the cross-toolchain (CT) (compiler+linker+utilities)
    - By using the CT we build some basic libraries for the target (BLT)
    - By using the HT and the BLT we build the full libraries
  - When you need to build it, there are plenty of things that can go wrong
    - Different versions of libraries installed on the host
    - Different (minor/major) releases of the compiler
    - May be difficult if not impossible to build if the host toolchain and the cross one have significant release mismatch
      - e.g., build a gcc 2.97 based cross-toolchain with gcc 5.0

Optimizing Compilers

- A compiler that tries to minimize or maximize some attributes of an executable computer program
- Common requirements are to minimize one or more of the following
  - Execution time
  - Memory requirement
  - Energy
- Compiler optimization is generally implemented by means of a sequence of code transformations
  - Algorithms which take a program and transform it to produce a semantically equivalent output program that uses fewer resources and/or executes faster
- Optimized code may make debugging more difficult

https://en.wikipedia.org/wiki/Optimizing_compiler

Optimizing Compilers Principles

- Optimize the common case: the common case may have unique properties that allow a fast path at the expense of a slow path. If the fast path is taken most often, the result is better overall performance
- Avoid redundancy: reuse results that are already computed and store them for later use, instead of recomputing them
- Less code: remove unnecessary computations and intermediate values
  - Less work for the CPU, cache, and memory usually results in faster execution
- Fewer jumps by using straight line code, also called branch-free code: less complicated code
  - Conditional or unconditional branches interfere with the prefetching of instructions
  - Using inlining or loop unrolling can reduce branching, at the cost of increasing binary file size by the length of the repeated code

https://en.wikipedia.org/wiki/Optimizing_compiler#Common_themes

Optimizing Compilers Principles

- Locality: code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference
- Exploit the memory hierarchy: accesses to memory are increasingly more expensive for each level of the memory hierarchy, so place the most commonly used items in registers first, then caches, then main memory, before going to disk.
- Parallelize: reorder operations to allow multiple computations to happen in parallel, either at the instruction, memory, or thread level.
- Strength reduction: replace complex or difficult or expensive operations with simpler ones
  - For example, replacing division by a constant with multiplication by its reciprocal

https://en.wikipedia.org/wiki/Optimizing_compiler#Common_themes

Optimizing Compiler Techniques

- Loop optimizations
  - For example:
    - Loop unrolling
    - Loop interchange
- Data-flow optimizations
  - For example:
    - Common subexpression elimination: in the expression (a + b) - (a + b)/4, the "common subexpression" “(a + b)” is computed only once
    - Constant folding and propagation: replacing expressions consisting of constants with their final value at compile time
      - e.g., 3 + 5 is replaced with 8 at compile time
    - Dead store elimination: removal of assignments to variables that are not subsequently used
      - e.g., because of a subsequent assignment that overwrites the first value.

https://en.wikipedia.org/wiki/Optimizing_compiler#Specific_techniques

Optimizing Compiler Techniques

- Code generator optimizations:
  - For example:
    - Register allocation: the most frequently used variables should be kept in processor registers for fastest access
    - Cache locking: some cache blocks are locked to avoid being replaced
      - Requires hardware support
    - Instruction scheduling: instruction scheduling is an important optimization for modern pipelined processors, which avoids stalls or bubbles in the pipeline by clustering instructions with no dependencies together, while being careful to preserve the original semantics
- …

https://en.wikipedia.org/wiki/Optimizing_compiler#Specific_techniques

Optimizing Compilers – Example: Loop Unrolling

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

```
int x;
for (x = 0; x < 100; x += 5 )
{
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}
```

- Code semantic is preserved
- Code size is increased
- More pressure on CPU registers
- Number of branches is decreased
  - Less conditional stalls / possible wrong branch predictions

https://en.wikipedia.org/wiki/Loop_unrolling#Simple_manual_example_in_C

Optimizing Compilers – Example: Loop Interchange

```
int x, y;
for (x = 0; x < 100; x++)
  for (y = 0; y < 50; y++)
    a[x,y] = x + y
```

```
int x, y;
for (y = 0; y < 50; y++)
  for (x = 0; x < 100; x++)
    a[x,y] = x + y
```

- Code semantic is preserved
- No change in code size
- Might have a big impact on performance:
  - Depending on how arrays are saved in memory (by columns or by rows) one version or the other is best
  - Exploits cache locality and lower pressure on the write-back mechanism

Optimizing Compilers in Practice

- In GCC use the -Olevel flags:
  - -O0 Reduce compilation time and make debugging produce the expected results. This is the default.
  - -O1 Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function
    - The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
  - -O2 Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff
  - -O3 Optimize yet more.
  - -Os Optimize for size. -Os enables all -O2 optimizations except those that often increase code size
  - -Ofast Disregard strict standards compliance
    - enables all -O3 optimizations. It also enables optimizations that are not valid for all standard-compliant programs.
- Alternatively, you can turn on/off specific optimization techniques

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Debugging

- Debugging is more complex for embedded devices:
  - Difficult to interact with the device
    - No display or keyboard
    - Sometimes no network connection (that provides external access)
  - Not only software is relevant, but also the interaction with the environment
    - Difficult to reproduce stimuli that come from the environment
  - In some cases, diagnostics is performed by means of leds or by connecting probes of an oscilloscope on output pins
- Debugging real-time applications (or any time dependent application) is inherently more difficult
  - Not only the application is involved, but also the operating system and the interaction with other applications

Application Debug

- Some embedded code can be debugged on the host machine
  - i.e., if the code can also be built on the host machine, correctness of the algorithms can be checked
- There exist emulators that provide the ability to test software applications to a certain extent
- Debug on device is possible if the device provides a suitable debug connector and/or by means of debug boards

Debugging on Device

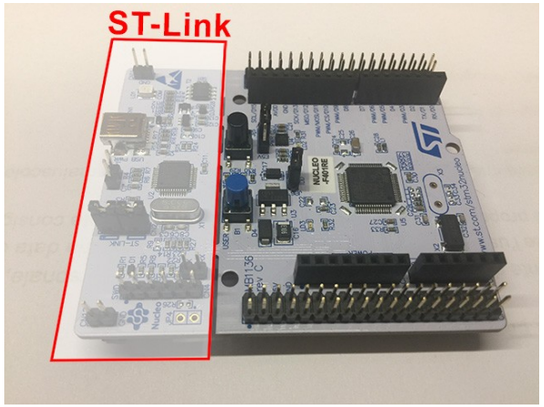
- Software on device can be debugged in different ways (can be combined)
  - By printing variables over a serial connection
  - LEDs and other visualization methods
  - Debugger
  - Oscilloscope
  - Logic analyzer

Debugging by printing variables

- It is a simple and often inefficient debug method
- If the target device has a serial (USB) connection, it is often possible to use it to print the contents of variables and visualize it on the host PC
- It requires changes in the code
- It might introduce delays and interactions that are not in the original code
  - E.g., it might not cope well with real-time applications
- Use of LEDs is similar, but on-device LEDs are used to visualize messages (e.g., the led is green when a certain point of a function is reached)

Debugger

- Debuggers are devices that provide the ability to connect to the target device
- Most of the development boards already have on-board debuggers
  - Other devices might need an external debugger instead
- Debuggers are suited to the target board/microprocessor
  - They exploit the debug options provided



https://www.maffucci.it/2017/12/07/stm-32-nucleo-supporto-arduino-core/

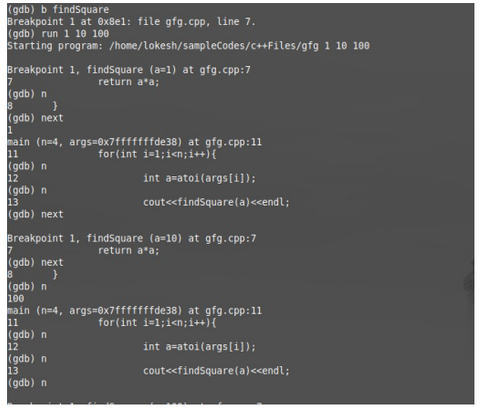
Debuggers – GNU GDB

- GDB is a software debugger that can be used to debug software on the host machine as well as on device
  - Client-server architecture:
    - Server: target device
    - Client: host device
- Open source
- Can be used
  - To insert brekapoints
  - To execute steb-by step
  - To monitor variables
  - …
- Text-based but often integrated into graphic IDEs
- Even though very powerful, not suitable to debug timing problems

https://www.gnu.org/software/gdb/

https://medium.com/@amit.kulkarni/gdb-basics-bf3407593285

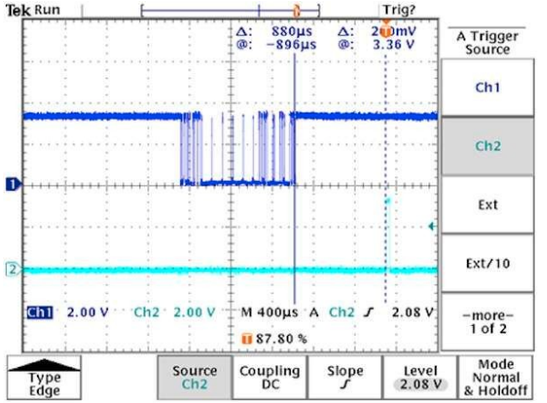
```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at example.c:4
(gdb) run
Starting program: /home/akulkarni/projects/gdb-basics/factorial

Breakpoint 1, factorial (n=5, a=1) at example.c:4
4               printf("Value of n is %d\n",n);
(gdb) condition 1 n==2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at example.c:4
        stop only if n==2
        breakpoint already hit 1 time
(gdb) continue
Continuing.
Value of n is 5
Value of n is 4
Value of n is 3

Breakpoint 1, factorial (n=2, a=60) at example.c:4
4               printf("Value of n is %d\n",n);
```



https://www.geeksforgeeks.org/gdb-command-in-linux-with-examples/

Oscilloscope

- A type of electronic test instrument that graphically displays varying signal voltages
  - The displayed waveform can then be analyzed for properties such as amplitude, frequency, rise time, time interval, distortion, and others
- Digital oscilloscopes offer the storage capability needed for investigating aperiodic or periodic events
- Measure the levels on selected pins
  - Oscilloscopes have limited number of channels
  - Usually, you pulse 1-4 pins on specific points of your software, as pass points
  - e.g., monitor pins that control an actuator or monitor pins used for communicating with other devices

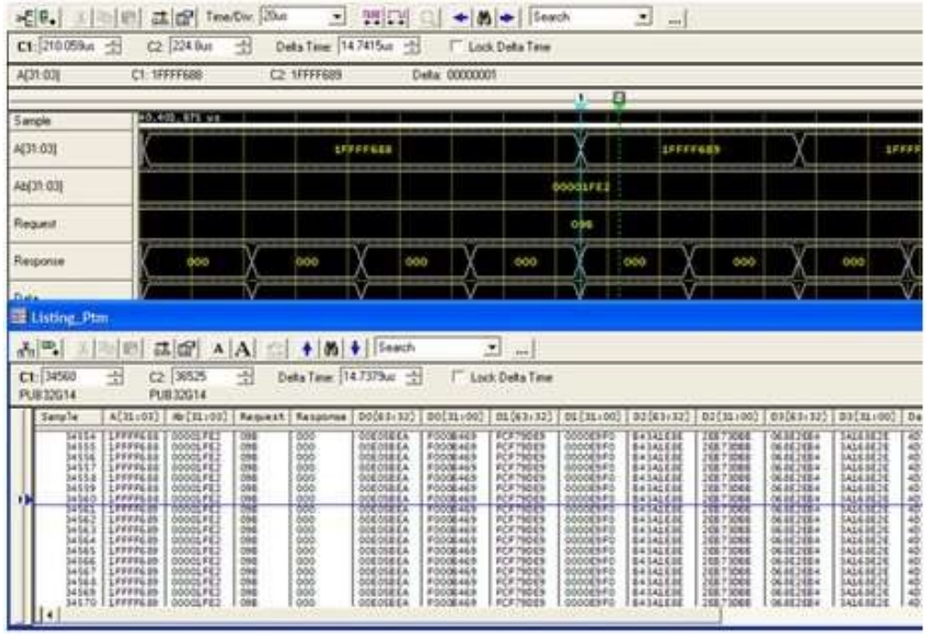


https://peaktech-rce.com/en/oscilloscopes-two-channels/577-peaktech-1337-digital-oscilloscope-8-bits-7-2-channels-100-mhz-2-gss-total.html

https://www.edn.com/how-to-debug-embedded-systems/

Logic Analyzer

- Electronic instrument that captures and displays multiple signals from a digital system or digital circuit
  - May convert the captured data into timing diagrams, protocol decodes, state machine traces, assembly language, or may correlate assembly with source-level software
  - They have advanced triggering capabilities, and are useful when a user needs to see the timing relationships between many signals in a digital system



https://www.edn.com/how-to-debug-embedded-systems/

Simulation and Data Logging

- To obtain reproducibility and/or to start testing parts of the system
  - Some parts can be simulated (e.g., the inputs from sensors, or the outputs to actuators)
  - Data can be collected by means of sensors, logged and replayed by means of a software module

In Short...

- Peculiarities of embedded systems programming
- The infinite loop
- Dealing with floating point numbers
- Cross compilation
- Optimizing compilers
- Debugging embedded code