ENCE463

Embedded Software Engineering

Lecture 1: Introduction

ENCE463

The team:

Steve Weddell (Course coordinator & Lecturer in Term-3)
steve.weddell@canterbury.ac.nz (office: von Haast 470)

Mike Shurety (Lecturer in Term-4)
cosmik.debris@elec.canterbury.ac.nz (office: von Haast 449A)

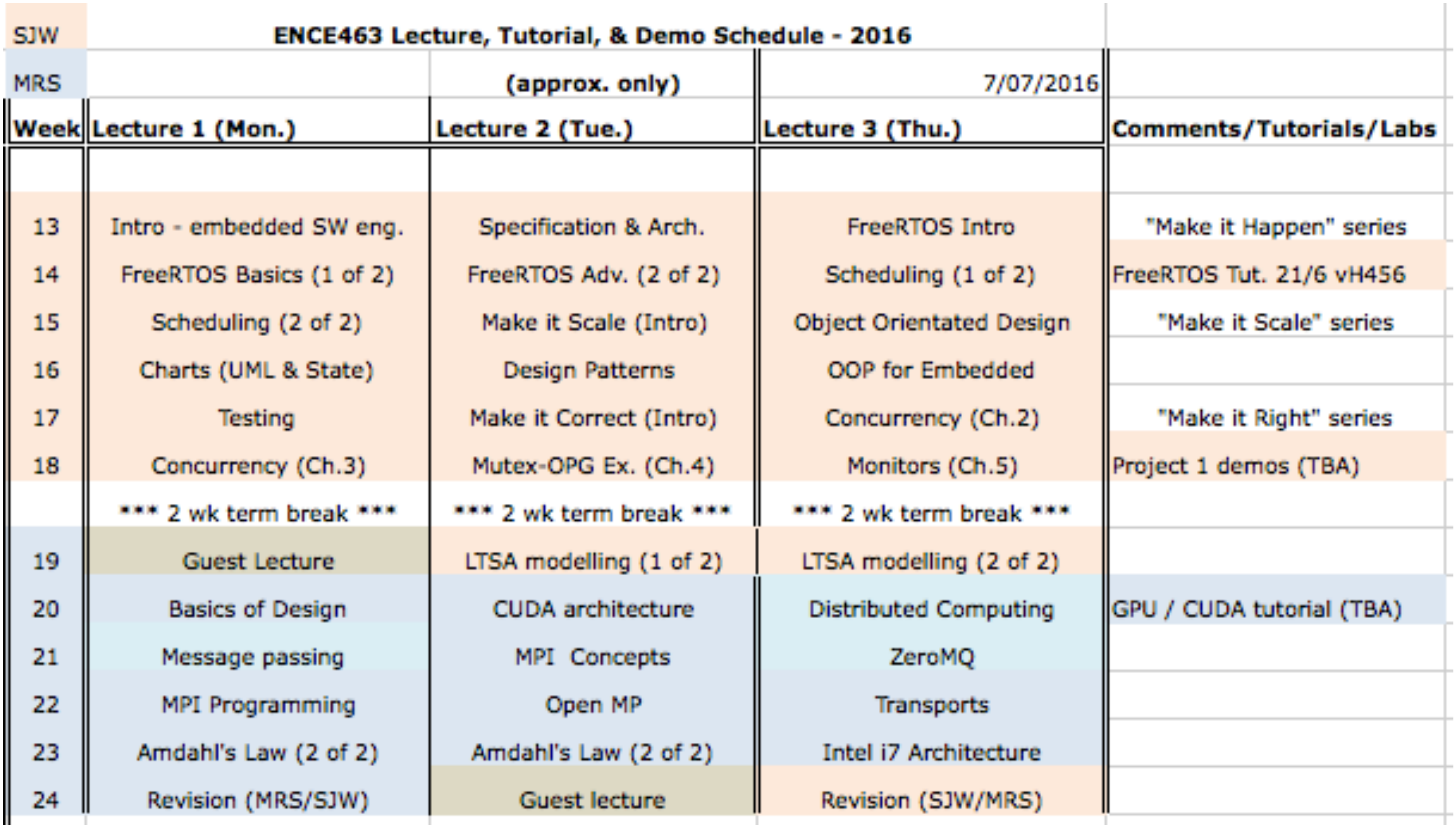
Dave van Leeuwen (Support)
dave@elec.canterbury.ac.nz (office: von Haast 455)

Assessment:

Individual Project (Term-3): 30% (several deliverables, final ones due 19th Aug.) Team Project (Term-4): 20% (report due 7th Oct.)

Final Exam : 50% (2.5 hours)

ENCE463



SJW	ENCE463 Lecture, Tutorial, & Demo Schedule - 2016			
MRS		(approx. only)	7/07/2016	
Week	**Lecture 1 (Mon.)**	**Lecture 2 (Tue.)**	**Lecture 3 (Thu.)**	**Comments/Tutorials/Labs**
13	Intro - embedded SW eng.	Specification & Arch.	FreeRTOS Intro	"Make it Happen" series
14	FreeRTOS Basics (1 of 2)	FreeRTOS Adv. (2 of 2)	Scheduling (1 of 2)	FreeRTOS Tut. 21/6 vH456
15	Scheduling (2 of 2)	Make it Scale (Intro)	Object Orientated Design	"Make it Scale" series
16	Charts (UML & State)	Design Patterns	OOP for Embedded	
17	Testing	Make it Correct (Intro)	Concurrency (Ch.2)	"Make it Right" series
18	Concurrency (Ch.3)	Mutex-OPG Ex. (Ch.4)	Monitors (Ch.5)	Project 1 demos (TBA)
	*** 2 wk term break ***	*** 2 wk term break ***	*** 2 wk term break ***	
19	Guest Lecture	LTSA modelling (1 of 2)	LTSA modelling (2 of 2)	
20	Basics of Design	CUDA architecture	Distributed Computing	GPU / CUDA tutorial (TBA)
21	Message passing	MPI Concepts	ZeroMQ	
22	MPI Programming	Open MP	Transports	
23	Amdahl's Law (2 of 2)	Amdahl's Law (2 of 2)	Intel i7 Architecture	
24	Revision (MRS/SJW)	Guest lecture	Revision (SJW/MRS)	

Previous courses covered the basics of how to program

ENCE361 covered developing embedded software:

- The embedded tool-chain
- The basics of embedded debugging and testing
- The concept of data abstraction
- How to interact with a variety of peripherals at a low level
- Interrupts and latency
- Simple real-time operating systems
- Simple concurrency (+ from ENEL373 and ENCE360)

This course

This course goes beyond simply knowing how to program to look at software engineering for embedded systems. Software engineering in general is about the application of a *systematic*, *disciplined*, *quantifiable* approach to the *development*, *operation* and *maintenance* of software.

Term-4 Project (coordinated by Mike)

Team (2 member) assesment for Term-4

Background
The Cochrane warp drive was first flown by Zefram Cochrane in 2163, although adopted by Starfleet it had a number of temporal instabilities. The Alcubierre drive and its modified version became the standard in the 23rd century. It had solved the instability problems but consumed vast amounts of negative energy to maintain the spacetime displacement bubble. Work done by Chris Van Den Broeck modified the equations and drastically reduced the energy required.
The new drive depends crucially on the ability to calculate, in real-time, the 3D Fast Fourier Transform (FFT) of the position wave function of the neutralino in the plasma conduit, to give its momentum flux. The neutralino is a supersymmetric Marjorana fermion and is the constituent of Dark Matter.

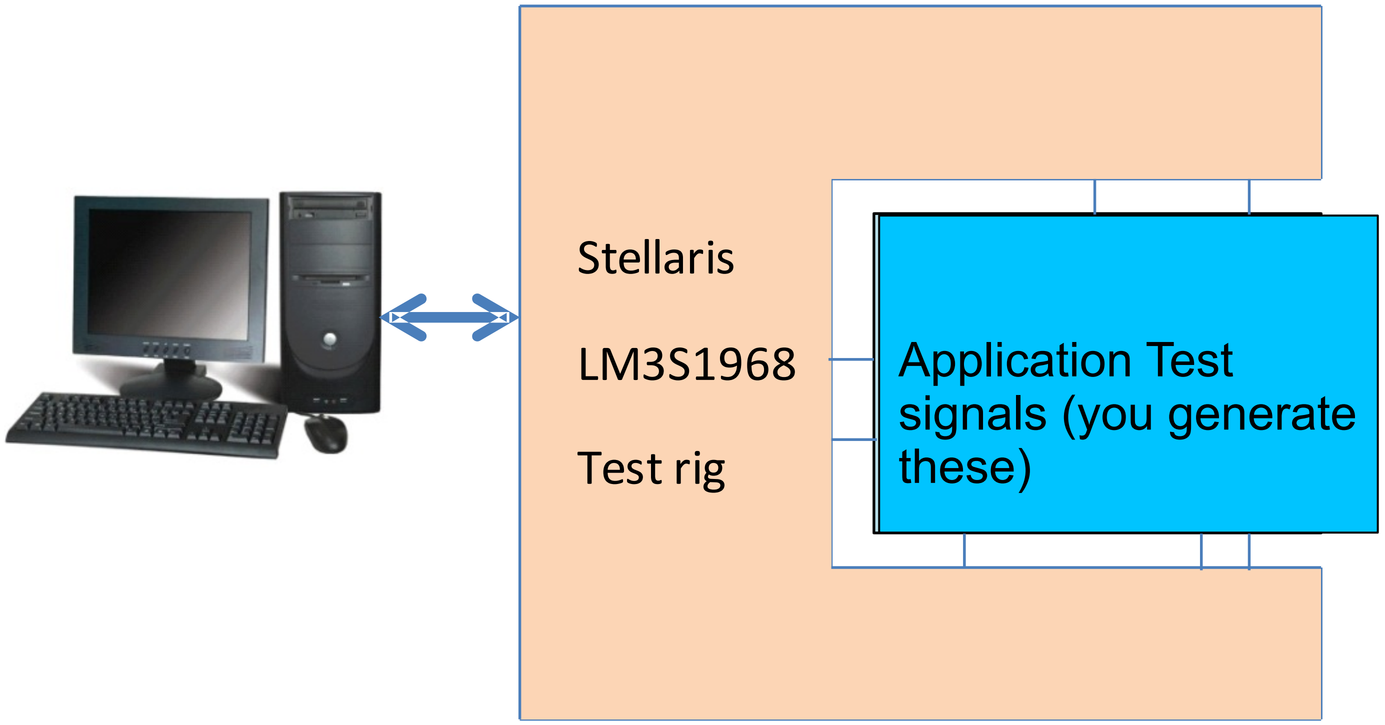
The Project
Your company makes the controllers and writes the embedded software to control the Van Den Broeck drive. You have been given the task to evaluate the 3D FFT on a GPU and compare it to the same FFT on a standard CPU.

Term-3 Project (coordinated by Steve)

Stellaris LM3S1968 boards + host PC (Windows or Linux)

FreeRTOS, C (or C++) on the Stellaris

Individual assessment



A very rough draft of the project specification:

Students are to design and implement a program for a real-time application on the Stellaris LM3S1968 board and using FreeRTOS (real time operating system). For your application, you may choose from the brief idea outlines given below, or come up with an idea of your own. In some cases this might be related to your ENEL400/ ENMT400 project. You will be working on your own in Project #1, but you are encouraged to discuss your designs with classmates. The program can be written in either C or C++ (FreeRTOS itself is written in C and has a C-callable API).

Examples…

At least two analogue signals from function generators are sampled; specific properties of those signals are analysed and the results displayed (note: the analogue range for the Stellaris ADC is 0 - 3V). The program is implemented such that the maximum sampling rate of the ADC is exercised.

ABS braking control system. Inputs are digital pulses from wheel rotation sensors on each of the 4 wheels, plus analogue signals representing the steering wheel and brake pedal positions. Any wheel rotating at 90% or less than its expected speed is deemed to be starting to lock up and the ABS must detect this (and apply pulsed braking to that wheel). Assume 50 pulses per wheel revolution, 500 mm wheel diameter and a maximum road speed of 180 kph: $50 \times 180 \times 1000 / 3600 / 0.5 / \pi \approx 1591$ pulses / sec for each of the 4 wheels.

FreeRTOS

"FreeRTOS is a market leading real time operating system (or RTOS) from Real Time Engineers Ltd. that supports 34 architectures and receives 103000 downloads a year. It is professionally developed, strictly quality controlled, robust, supported, and free to use in commercial products without any requirement to expose your proprietary source code. Why would you choose anything else?" - www.freertos.org

Features:
FreeRTOS is designed to be simple and easy to use: Only 3 source files that are common to all RTOS ports, and one microcontroller specific source file are required, and its API is designed to be simple and intuitive.
Supports:
Tasks and co-routines
Tasks are implemented as C functions
Fixed priority pre-emptive scheduling (based on ticks from SysTick)
Optional cooperative scheduling
Queues
Binary and counting semaphores
Mutexes and recursive mutexes
Stack overflow detection

Why do we care about ***engineering*** embedded software?

The lawyers are coming!

The quality of a lot of embedded software is abysmal. And lawyers are on to it. If you don't want your source code to show up in court, you better get your act together.

By Michael Barr   Embedded.com   (2009)

Listing 1   A single line of uninteligible mystery code.

```
y = (x + 305)/146097 * 400 + (x + 305) % 146097 / 36524 * 100 * (x + 305)
    % 146097 % 36524 / 1461 * 4 + (305) % 146097 % 36524 % 1461 / 365;
```

Why do we care about ***engineering*** embedded software?

Listing 1   A single line of uninteligible mystery code.

```
y = (x + 305)/146097 * 400 + (x + 305) % 146097 / 36524 * 100 * (x + 305)
    % 146097 % 36524 / 1461 * 4 + (305) % 146097 % 36524 % 1461 / 365;
```

"The original listing had no comments on or around this line to help. I eventually learned that this code computes the year, with accounting for extra days in leap years, given the number of days since a known reference date (such as January 1, 1970). But I note that we still don't know if it works in all cases, despite it being present in an FDA-regulated medical device."

Michael Barr   Embedded.com   (2009)

Why do we care about ***engineering*** embedded software?

“Consider the case of the *Alcotest 7110*. After a two-year legal fight, seven defendants in New Jersey drunk driving cases successfully won the right to have their experts review the source code for the Alcotest firmware. Expert reports evaluating the quality of the firmware source code were produced:

• Of the available 12 bits of ADC precision, just 4 bits (most-significant) are used in the actual calculation. This sorts each raw blood-alcohol reading into one of 16 buckets.
• Out of range A/D readings are forced to the high or low limit. This must happen with at least 32 consecutive readings before any flags are raised.
• There is no feedback mechanism for the software to ensure that actuated devices, such as an air pump and infrared sensor, are actually on or off when they are supposed to be.
• The software first averages the initial two readings. It then averages the third reading with that average, then the fourth reading is averaged in, and so on. Thus the finalreading has a weight of 0.5 in the "average", the one before that 0.25, and so on.
• Out of range averages are forced to the high or low limit, too.
• Static analysis with `lint` produced over 19,000 warnings about the code (that's about three errors for every five lines of source code).

What would you infer about the reliability of a defendant's blood-alcohol reading if you were on that jury?”

Michael Barr Embedded.com (2009)

What makes an embedded system different from a desktop system?

- Interaction with the physical world implies a need for meeting real-time deadlines. Total data throughput is less important than meeting deadlines. More importantly, we need to understand the physical processes that the software is interacting with, and how the physical processes and the software will work together.

- Interaction with the physical world is often through application-specific sensors or actuators, rather than a standard mouse, keyboard, display or network port.

- The software must run on a non-desktop processor, often on a custom PCB. Insight into the internal state of the software is more difficult to come by. Standard operating system services may not be available (or appropriate).

- The application may require high reliability. Systems often must work without fail or with minimal interference. What can go wrong? How can we think through the possibilities? How can we recover?

- The software must interact simultaneously with multiple external processes that may produce continuous signals or sporadic events. This implies that some kind of concurrency will be required.

- **Embedded systems are generally resource-constrained and cost-constrained**.

What we're trying to avoid:

Avoiding the Top 43 Embedded Software Risks
Philip Koopman (Carnegie Mellon University)
Embedded Systems Conference, Silicon Valley, 2011

Koopman's identified software risk areas included:

#3. No written requirements
#4. Requirements omit extra-functional aspects
#5. Requirements with poor measurability
#6. No defined software architecture
#7. Poor code modularity
#8. Too many global variables
#10. Design skipped or is created after code is written
#11. Flowcharts are used in place of statecharts
#15. No real time schedule analysis
#19. No test plan
#31. High requirements churn
#32. No version control

Avoiding the Top 43 Embedded Software Risks
Philip Koopman (Carnegie Mellon University)
Embedded Systems Conference, Silicon Valley, 2011

Koopman's advice includes:

2. Define requirements in a measurable, traceable way. If you can’t measure a requirement, you don’t know if you've met it.
3. Document your software architecture. Make your code modular and avoid global variables.
7. Do real time scheduling analysis and use a 3rd party RTOS if you are doing preemptive task switching.
8. Pay attention to concurrency to avoid tricky timing bugs. Home-brew methods are risky.
11. Ensure that problems found both in test and at runtime are identified and tracked.
12. Explicitly specify and plan for performance, dependability, security and safety up-front. Most embedded systems involve all these and it is painful to try to add them at the end of the project.
16. Leave plenty of slack resources. Use as little assembly language as possible (zero assembly code is a good amount).

What things matter in embedded software development?

Jack's Top Ten (Identifying the 10 most important issues in embedded software development)

Jack Ganssle *(jack@ganssle.com)*

December 1, 2006

“Embedded systems defy conventional test techniques. How do you build automatic tests for a system which has buttons some human must push and an LCD someone has to watch? A small number of companies use virtualization. Some build test harnesses to simulate I/O. But any rigorous test program is expensive.

“Programming is a human process subject to human imperfections. The usual tools, which are usually not used, can capture and correct all sorts of unexpected circumstances. These tools include checking pointer values; range checking data passed to functions; using asserts and exception handlers, and checking outputs (I have a collection of amusing pictures of embedded systems displaying insane results, like an outdoor thermometer showing 505 degrees, and a parking meter demanding $8 million in quarters).

“For very good reasons of efficiency, C does not check, well, pretty much anything. It's up to us to add those [checks] that are needed.”

Final words (in this lecture anyway!)

It's worth emphasizing (and constantly bearing in mind) that one of the essential features of software, and one of its key contributors of value in a system design, is **that it is soft**. That is, software can be changed and adapted to meet changing customer needs or to respond to our evolving understanding of the problem space. How can we design our systems to best take advantage of this fact?

Outside of the books and articles cited in this course, one of the best references for embedded software engineers is the articles at http://www.embedded.com/