Embedded Xinu on the ARM 32F4 Discovery Board

Ethan Weber
Marquette University
615 N 11th St Milwaukee
Milwaukee, Wisconsin
webere@mscs.mu.edu

ABSTRACT

Xinu is a small embedded operating system which exists on the MIPS routers in the Marquette Systems Lab. It serves as an educational system in teaching several courses at Marquette University. Porting the operating system over to an ARM 32F4 Discovery board serves to expand the educational experience offered by Xinu. The ARM board has added capabilities including audio I/0 and motion detection via an accelerometer. In order to achieve this goal, several key links must be made between the operating system and the hardware. These include a process manager, memory manager, and user command interface. This is done by creating USART communications with the user. In addition, a null process is made with respect to a context switch. Next, an interrupt controller is created to swap in and out of processes according to a schedule. Currently, further testing must be applied to the USART communications and the context switch. In addition, an interrupt controller needs to be implemented. Once completed, communication efforts can begin with the audio I/0 and accelerometer. In conclusion, a completed Xinu transition to the ARM board will increase the educational experience through the embedded Xinu operating system.

Categories and Subject Descriptors

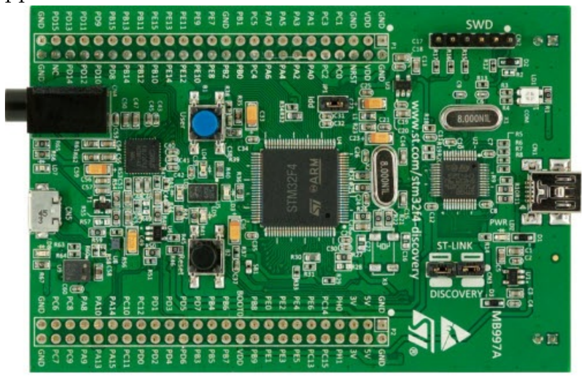
[]: **Operating Systems: Embedded Systems:** Embedded Xinu

General Terms

Embedded Operating System, Xinu, ARM 32F4 Discovery

1. INTRODUCTION

The Combination of Embedded Xinu on the ARM 32F4 Discovery Board will serve as a gateway to bigger and better things Xinu. The completion of this project demands replacement code for all platform specific code (Assembly Language) implemented in Xinu. Additional management will be needed to support operation on various devices including the audio I/O and the accelerometer. This paper will go through the steps to complete this goal, as well as, highlight different features the ARM Discovery board has to support various functions.



STM32F4 High-Performance Discovery Board
STM32F407VGT6 µCU
32-bit ARM Cortex-M4F core: 1 MB Flash + 192 KB RAM
[10]

First an overview of Xinu and the discovery board will be had along with reasons/benefits for combining the two. Then basic input and output will be surveyed and weighed according to complexity. Next a context switch along with interrupts will be discussed. Finally, peripheral devices will be over-viewed along with future possibilities and conclusions on the research.

2. EMBEDDED XINU ON THE ARM 32F4 DISCOVERY BOARD

Before diving into working directly with Xinu and ARM, it is important to have a solid base in both C programming and Xinu. A great way to do this is to follow the coursework for embedded Xinu on the MIPS routers in the Marquette University Operating Systems course. Assignments one and two provide hands on work with C and learning its basics. Next is the serial communications for Xinu on the MIPS routers. The Xinu used for this assignment is an incomplete stripped down version of Xinu. It is geared towards step by step creating the program code needed to implement Xinu on the MIPS routers. The next assignment deals with the context switch. This is great to illustrate how processors use registers and what needs to be done in order to switch to another process. Also, it shows how there are particular standards that need to be followed when swapping data in these registers. This is a great lead into working with Xinu and ARM. In fact, it is the same steps that need to be taken. Comparing ARM to MIPS should then reveal what needs to be changed in order for the context switch to work with ARM rather than MIPS.

2.1 Motivation

There are many reasons for porting Xinu to the ARM 32F4 Discovery board; first and foremost it diversifies the educational experience offered by Xinu. By adding hardware that runs on a completely different assembly language, students can compare and contrast the differences in the two. This provides further insight into what it takes to link an operating system to hardware.

In addition, the ARM 32F4 Discovery board provides new peripheral devices that students can experiment with. These devices include an accelerometer and audio input and output. These extra bells and whistles may create increased interest in embedded Xinu. Also, it could lead to more creative projects that build off of Xinu. Overall, the embedded Xinu universe is better off with the increased diversity provided by the ARM board.

2.2 Embedded Xinu

"Embedded Xinu is an ongoing research and implementation project in the area of Operating Systems and Embedded Systems." Xinu was created by Dr. Doug Comer while the Embedded Xinu project was created by Dr. Dennis Brylow. It is currently used in the Marquette University Systems Laboratory as the focus for their Operating Systems course. The Embedded Xinu project is in the process of expanding to other University laboratories across the United States. The idea is to provide a vehicle for students to learn about operating systems and get their hands dirty at the same time. Xinu runs on the WRT54GL routers so that it can be implemented in other university at a low cost. Instructions to install these systems can be found on the Embedded Xinu wiki page. Further information and help can be found by contacting Dr. Dennis Brylow.

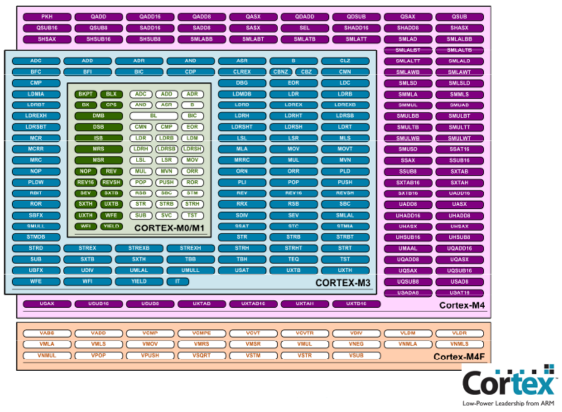
2.3 ARM 32F4 Discovery

The ARM 32F4 Discovery board is a small development board meant for research and development. It comes setup for windows, but a tool chain can be found for Linux distributions atjethomson.wordpress.com. The instruction on the web page are sufficient for setting up the development environment. After following the instructions multiple sample projects should be available from the download. The key features of the ARM 32F4 Discovery include:

- "STM32F407VGT6 micro controller featuring 32-bit ARM Cortex-M4F core, 1 MB Flash, 192 KB RAM in an LQFP100 package
- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone ST-LINK/V2 (with SWD connector for programming and debugging)
- Board power supply: through USB bus or from an external 5 V supply voltage
- External application power supply: 3 V and 5 V
- LIS302DL, ST MEMS motion sensor, 3-axis digital output accelerometer
- MP45DT02, ST MEMS audio sensor, omni-directional digital microphone
- CS43L22, audio DAC with integrated class D speaker driver
- Eight LEDs:
  - LD1 (red/green) for USB communication
  - LD2 (red) for 3.3 V power on
  - Four user LEDs, LD3 (orange), LD4 (green), LD5 (red) and LD6 (blue)
  - 2 USB OTG LEDs LD7 (green) VBus and LD8 (red) over-current
- Two push buttons (user and reset)
- USB OTG FS with micro-AB connector
- Extension header for all LQFP100 I/Os for quick connection to prototyping board and easy probing"[10]

2.3.1 *Thumb Assembly Language*

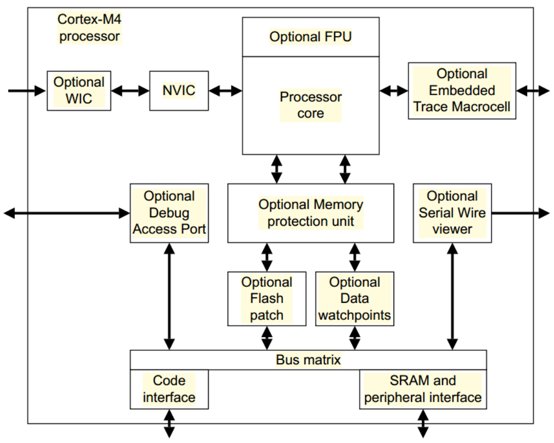
The board uses the ARMv7-M Thumb2 instruction set. Thumb was created as a 16bit version of the RISC compliant ARM architecture and was originally created for increased code density perfect for small devices. Thumb2 is an implementation that supports a few 32 bit instructions, as well. This extends the functionality available to the developer on their ARM device; however, the majority of code is still 16 bit, thus maintaining a high code density. The depth of the M series instruction set also changes from M0 processors to M4F processors. As shown in the figure below, the M4F processor is a super set of all other M series processors. This means it can run any code compiled for all other M series processors.



[5]

2.3.2 *Processor - Cortex M4*

"The Cortex-M4 processor is a low-power processor that features low gate count, low interrupt latency, and low-cost debug. The Cortex-M4F is a processor with the same capability as the Cortex-M4 processor, and it includes floating point arithmetic functionality. Both processors are intended for deeply embedded applications that require fast interrupt response features." [10](1.1 r-manual) This processor includes a core, Nested Vector Interrupt Controller (NVIC), Memory Protection Unit (MPU), and a Floating Point Unit (FPU).


Cortex-M4 processor
Optional FPU
Processor core
Optional WIC
NVIC
Optional Embedded Trace Macrocell
Optional Debug Access Port
Optional Memory protection unit
Optional Serial Wire viewer
Optional Flash patch
Optional Data watchpoints
Bus matrix
Code interface
SRAM and peripheral interface

[10]

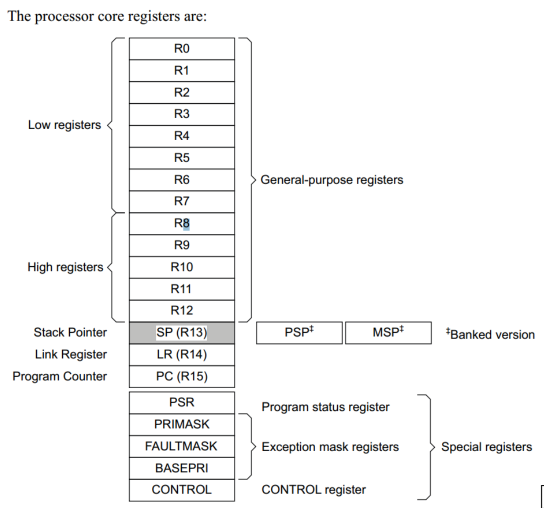
2.3.3 *Floating Point Unit*

The FPU will add floating point operations to Xinu. Previous implementations of Xinu do not support floating point. This FPU provides:

- "32bit instructions for single-precision (C float) data-processing operations.
- Combined Multiply and Accumulate instructions for increased precision (Fused MAC).
- Hardware support for conversion , addition, subtraction, multiplication with optional accumulate, division, and square-root.
- Hardware support for denormals and all IEEE rounding modes.
- 32 dedicated 32-bit single precision registers, also addressable as 16 double-word registers.
- Decoupled three stage pipeline." [10]

2.3.4 *Registers*

There are 17 32 bit registers for the Cortex M4. They are referred to as r0-r15 and PSR. They can also be referred to by their alias. R0-r3 (also a1-a4) are the argument registers. They are used to the arguments passed by processes. The registers r4-r11 (v1-v8) are all volatile registers and do not need to be saved in memory during a context switch. Register r12 (IP) is a special register reserved for Intra Procedure. R13 (SP) is the stack pointer register. R14 (LR) is the link register. R15 (PC) is the program counter. The PSR register is a special register called the program status register. It is diagrammed below along with all of the other registers. Low registers are accessible by all instructions and the high registers are only available to 32 bit instructions.



[10]

2.3.5 *Nested Vector Interrupt Controller*

The NVIC is a special on chip interrupt controller meant to allow for fast interrupt processing. Its features are listed below.

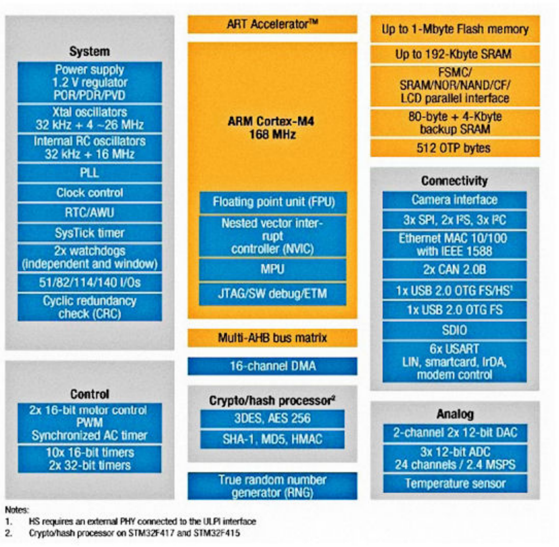
- "External interrupts, configurable from 1 to 240
- Bits of priority, configurable from 3 to 8.
- dynamic re-prioritization of interrupts.
- Priority grouping. this enables selection of preempting interrupt levels and non preempting interrupt levels.
- Support for tail-chaining and late arrival of interrupts. This enables back-to-back interrupt processing without the overhead of state saving and restoration between interrupts.
- Processor state automatically saved on interrupt entry, and restored on interrupt exit, with no instruction overhead.
- Optional Wake-up Controller (WIC), providing ultra-low power sleep mode support."[10]

2.3.6 *Memory Protection Unit*

The MPU is an optional unit meant for setting up access permissions and privileges.

- "Eight memory regions.
- Sub Region Disable (SRD), enabling efficient use of memory regions.
- The ability to enable a background region that implements the default memory map attributes."[10]

2.3.7 *Periphial Devices and more*



Above is a block diagram of the stm32f4 discovery. As seen in the diagram, the board supports additional connectivity including a camera interface, 6x USART, SDIO, and Ethernet to name a few. These are not on the board itself, but they can be added via two 50 pin connectors available on either side of the board. The stm32f4 user manual shows which pins are necessary for each type of add on. On the board itself exist audio I/O, an accelerometer, and two usb plug ins. These do not need any additional hardware add-ons.

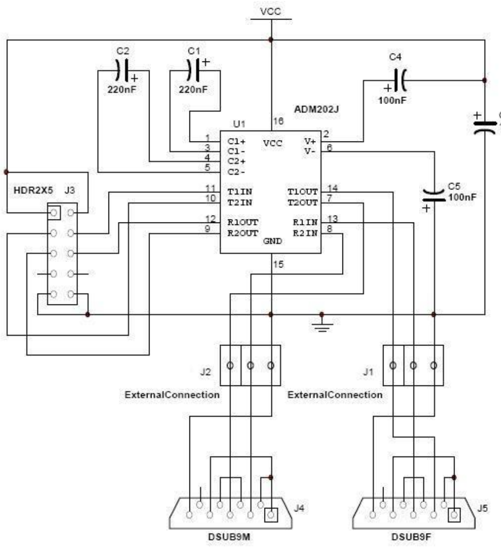
2.4 Basic I/O

Input on the ARM 32F4 Discovery board is initially set up to do be done over usb. After compiling any project it is then pushed onto the board via the command flash write [your-project.bin] 0x8000000. This places the program into the necessary memory location to boot from. Once the board is power cycled, the program will take effect. It then follows that basic I/O features can be done over usb. However, the MIPS routers and Xinu are already configured to use USART for basic I/O. Rather than research into usb standards and protocols, a USART connection was made with the arm discovery board. Once properly built, USART communication with the ARM 32F4 Discovery can be created.

2.4.1 USART

USART stands for Universal Synchronous/Asynchronous Receiver/Transmitter. It supports both asynchronous and synchronous communication. To achieve this a serial driver needs to be written. This involves opening the serial port and sending characters back and forth through the port. Synchronous communication is the first step to complete as it does not deal with interrupts. Once interrupts are implemented, the driver would then be considered asynchronous. Once completed the driver will serve as a command interface between the Xinu operating system and the I/O device connected via the serial port.

2.4.2 Building a Serial Port on the Discovery Board



The first step is to create a transceiver board that converts the 3.3 V signals from the ARM 32F4 Discovery board to that of of the RS-232 serial communication that is being adapted to the board. This same process needs to be done for the MIPS routers already installed at the Marquette University Systems Lab. Detailed documentation on how to perform this task can be found on the Xinu wiki[3]. The ARM board only needs support for one serial communication, so once the transceiver board is built, only 4 of the of the pin holes need to be used: one for transceiver, receive, power and ground. In accordance to the above figure, this could be either T1OUT and R1IN or T2OUT and R2IN. Both VCC and GND need to be connected, as well.

The TxOUT needs to be connected to the Rx Pin on the ARM board, and the RxIN needs to be connected to the Tx Pin. For the code below, the USART2 connection was used on the ARM board. This maps to the PA2 (Tx) and PA3 (Rx) pins on the board. Detailed description of the pins are found in the ARM 32F4 Discovery User Manual if a different USART setup is desired.

2.4.3 Communicating with the Serial Port

To communicate via the serial port on the ARM board, the device must first be properly configured to use the serial port. The default fault mode is not configured to use the serial port. After that, it must also be configured to listen to the port as well as send characters back over the port. A working example can be found at this link[2]. The code on this website has several dependencies that can be found in the Libraries/STM32F4xx_StdPeriph_Driver/inc/ stm32f4xx_usart.h directory downloaded with the tool chain for the board. The uart_puts("Init complete!"); line did not compile, so it is commented out as seen below. Otherwise, the function of the code is to initialize the usart and then forever send the character 'h' over the port in intervals determined by the delay set.

```
int main(void)
        {
        init_usart();
        //uart_puts("Init complete!");
        while(1){ USART_SendData(USART2, 'h');
                        }
        }
```

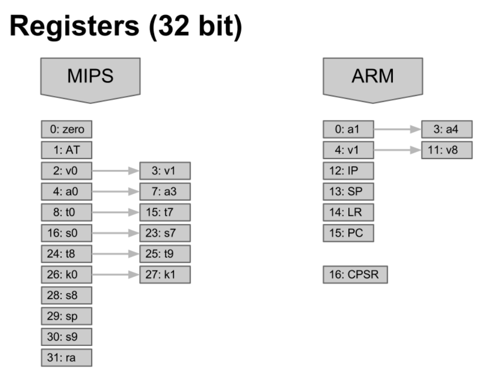
To function as a component of the Xinu operating system, the USART port would not constantly send the character 'h' but instead would send when the transceive buffer on the ARM device has characters to send. This would be executed via a listener to that register.

2.5 Context Switch

A context switch serves to change processes being executed by the processor. Information and data about a current process is held within the processor registers. When a process is changed, the important information in these registers need to be stored in memory(RAM) and then new information and data swapped in their place. This needs to be done quickly and in an orderly manner so nothing is misplaced. The implementation of a context switch is thus dependent upon the processor it is meant for. The processor registers for the MIPS routers is quite a bit different from that on the ARM board. These differences need to be accounted for a successful Embedded Xinu port over to the ARM 32F4 Discovery.

2.5.1 MIPS vs ARM

The MIPS architecture has 32 registers compared to 17 registers in the ARM architecture. These registers each contain 32 bits of data. In MIPS there are 10 temporary registers that can be used unconditionally. Alternatively, there are 10 Saved registers that must be copied and replaced before and after use. Other registers include 4 for arguments, 2 for a varied length return value (32 or 64 bit), a return address, a stack pointer, 2 kernel scratch registers, a temporary assembly register, and a zero register.



In comparison, ARM designates 13 registers as general purpose, a stack pointer, a return address(link register), program counter, and a program status register. According to the standard, the first 4 general purpose registers are used for arguments. The next 8 registers are volatile registers, or temporary registers. The last register is used as an Intra Procedure register.

In addition, to the processor registers, there exist registers for the Floating Point Unit. There are 32 floating point registers. These registers are also 32 bit and are divided into saved and temporary registers by standard. The first 16 are callee save, while the last 16 are temporary registers. They can also be referenced as 64 bit and 128 bit registers for larger operations. This is done by the aliases d0 (r0 + r1) - d15 and q0 (d0 + d1) -q7 that stand for double word and quad word.

2.5.2 ARM code and Assembler Directives

A few things to note when writing code for the ARM Discovery board is that there are some platform specific assembler directives. Assembler directives are not assembly code themselves so they won't be found in an instruction set for an assembly language. Rather, directives tell the assembler how to assemble the code. Often times these directives can be issued as options in the command issued to the compiler such as gcc. However, they are also available to be hard coded into the Assembly code itself. As mentioned above, there are general assembler directives and architecture specific ones. The ARM specific ones can be found at this link[1].

The coding process for making the context switch is a bit easier. Rather than explicitly managing the memory space for register information to be put onto the stack. ARM has support for push and pop commands in the Thumb assembly language. If the FPU registers are the target of the command, then vpush and vpop are used. The context switch then needs only to keep track of the order in which data is pushed and popped.

2.6 Interrupts

Most modern day operating systems are interrupt driven. This means that they execute down a main program until it is completed or it is interrupted. The interrupt tells the computer to change course and continue execution on some other code. These interrupts serve as a way to create multitasking. The processor can work on several processes seemingly at once by rapidly switching them in and out.

2.6.1 NVIC

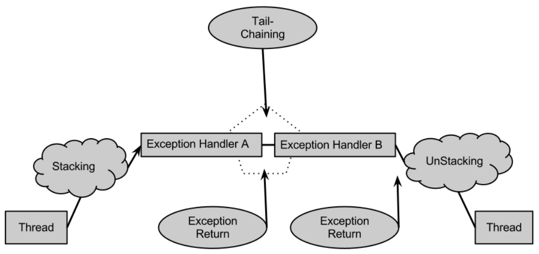
The Nested Vector Interrupt Controller serves as a center for handling interrupts. When an interrupt is signaled, the processor consults the NVIC in how to handle the interrupt. The NVIC supports up to 240 interrupts.

2.6.2 Priority

The NVIC supports 256 levels of priority and group priority. This means it can differentiate between 256 ordered types of interrupts and can execute them in order from most urgent to least urgent. Additionally, some interrupts can be grouped together to have the same priority, in which case they would be executed on a first in first out bases. Furthermore, priorities of the interrupts can be changed dynamically, allowing for more user control. All of this may not mean much for Xinu right away, but future work could lead to the ARM board embedded with Xinu being a multitasking machine.

2.6.3 Tail-Chaining

The NVIC supports tail-chaining which is the execution of one interrupt immediately after another. As illustrated below, an interrupt signal is given and the main thread of execution is put onto the stack. Then the handler for the interrupt is given to the processor. At this point another interrupt has been given. Rather than unstacking the the main thread of execution back onto the processor and then stacking it again to handle the interrupt, tail-chaining allows for the next interrupt to be handled immediately. Overall, this saves time and energy on the ARM board and allows for a more fluid thread of execution.



3. CONCLUSIONS

In Conclusion, there is a lot of background and research needed to understand the ARM 32F4 Discovery, Xinu, Operating Systems, Assembly Language, C, and how they all connect together. The bulk of this research has gone into learning these topics and how they function. Since the end result of the research is by large this paper, it should serve as a concise and compact point of reference for those intending to do future work on the ARM 32F4 Discovery. This paper reviews the ARM board and several of its important technical features. It also overviews the assembly language required to speak to the board and some about assembly directives. In addition, it mentions how to go about creating serial communication, context switching, and interrupts.

3.1 Future Work

So far I have research a lot about the board itself, the assembly language, and found content and examples to help out the porting of Embedded Xinu to the ARM 32F4 Discovery. Future work will be using the information to adapt the serial communications to Xinu. Also, the context switch and interrupts need to be created. Once those are done, accelerometer and audio I/O support can be added. Then future extensions could be made through the boards support for Ethernet, SDIO, and camera interface. Finally, a model of the approach to port Embedded Xinu to the ARM 32F4 Discovery board could be crafted for the classroom environment.

3.2 Related Work

One related project for the ARM 32F4 Discovery board can be found at www.emcu.it. This project goes over each step on how to put their Real-Time Operating System onto the ARM board. It goes over several helpful topics including a scheduler for interrupts and a context switch.

3.3 Acknowledgments

I would like to recognize Dr. Dennis Brylow in his help as a mentor and as a leader for this REU program. Also, I would like to thank NSF and Marquette University for their financial and resource support throughout the ten weeks of the program. I would like to give additional thanks to Teddy Sudol, Alex Brecherer, and Mike Ziwosky for their support in the Systems Lab at Marquette University.

4. REFERENCES

[1] Anonymous. Arm directives - using as. http://sourceware.org/binutils/docs/as/ARM-Directives.html, jul 2012.

[2] Blog, E. E. Stm32f4 discovery usart example. http://torrentula.to.funpic.de/2012/05/20/stm32f4-discovery-usart-example/, jul 2012.

[3] Brylow, D. Main page. http://sources.redhat.com/binutils/docs-2.12/as.info/Pseudo-Ops.html, jul 2012.

[4] Corliss, G. Mu coen 4820 operating systems and networks. http://www.eng.mu.edu/corlissg/OpSys.12Sp/, jul 2012.

[5] emcu.it. Freertos on stm32f4-discovery. http://www.emcu.it/STM32F4xx/STM32F4xx.html, jul 2012.

[6] Flynn, I. M. *Understandin Operating Systems.* Course Technology, Cambridge, MA, 2006.

[7] Noergaard, T. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers.* Elsevier/Newnes, Amsterdam, 2005.

[8] redhat.com. Assemblerdirectives. http://sources.redhat.com/binutils/docs-2.12/as.info/Pseudo-Ops.html, jul 2012.

[9] Scott, M. L. *Programming Language Pragmatics.* Morgan Kaufmann, San Francisco, 2009.

[10] STMicroelectronics. Stm32f4discovery. http://www.st.com/internet/evalboard/product/252419.jsp, jul 2012.