A Real-Time Operating System for the EFM32HG

Vlad Dumitru-Popescu
Keble College
University of Oxford

Supervisor: Alex Rogers

Third Year Project Report
Honour School of Computer Science - Part B
Trinity 2020

Abstract

Embedded applications run on naked hardware and rely on precise assumptions about the physical implementation of peripherals. Code is not compatible between two peripherals with different structure, but otherwise the same function.

We provide a real-time operating system for the EFM32HG that separates the application code from the hardware, and which also offers multi-tasking and services to processes through system calls.

The ability to compile applications independently of drivers and the operating system paves the way for smaller, more portable applications that can be easily shared.

Acknowledgements

I am greatly indebted to my supervisor, Professor Alex Rogers, for his incredible guidance throughout the project, and for sparking my passion for embedded development. He has been very available, and has always given me excellent advice with regards to both technical and design decisions.

I would also like to thank Professor Mike Spivey for his thorough Digital Systems course, and for his Phos operating system which served as heavy inspiration for this project.

Contents

1 Introduction

Embedded systems are microcontrollers with a dedicated function within a larger mechanical or electrical system. They generally control the physical operations of the machine they are embedded within, and their uses are wide-ranging, from plane radars and car radios to home automation systems. Ninety-eight percent of all microprocessors manufactured are used within embedded systems. They generally offer a small amount of processing power and memory, and thus make for an interesting challenge to program.

1.1 Motivation

Code for embedded systems traditionally runs 'on the metal', exposing the programmer to the full complexity of the underlying hardware. Changes to the peripherals and the wiring of the board may make existing applications obsolete, even though the physical services the board offers remain identical. We aim to provide a means for improving the portability of embedded applications by building a framework which decouples hardware services from the application code.

To better illustrate this goal, let us imagine a programmer working on a simple embedded device consisting of a microcontroller, a thermometer peripheral and an USART peripheral. The programmer is writing an application that periodically asks for the temperature, computes the variance in the measured values every hour, and outputs the result over a serial connection.

In most embedded systems, interaction with peripherals is done through exposed registers which are mapped onto the address space of the microcontroller. To read the temperature, the programmer would first have to initialise the thermometer by configuring its registers to produce an interrupt every second, then set up a handler for the interrupt which reads the temperature value out of another register when triggered. The readings would then be buffered and processed once per hour, and finally transmitted out of the system through USART.

Suppose now that the particular thermometer peripheral used by the device is taken off the market and a new version is offered in its place. The new thermometer's readings are much more precise and it even supports outputting in different temperature scales! The lowest order bit of the output register would be used to differentiate between a Celsius value and a Fahrenheit value. To obtain the actual reading, one must shift the remaining bits right by one. As the thermometer manufacturer is American, it is set to output in Fahrenheit by default. The device manufacturer is satisfied with this new peripheral and includes it in all its products going forward, carefully mapping the registers to the same addresses to maintain compatibility with existing programs.

The programmer also receives a copy, and eagerly flashes his old application on the device. Much to his surprise, it does not work - it reads a temperature above boiling despite being located in his basement. To top it all off, it crashes barely a minute into running. The programmer is furious - what had he done wrong?

Of course, the answer was nothing. The programmer's code was perfectly correct, but made assumptions that did not hold anymore for the new iteration of the embedded device. It assumed that all temperature readings were in Celsius, whereas the new peripheral defaults to Fahrenheit, and never specified the temperature scale to be used in the initialisation of the thermometer.

Code being made obsolete by changes to the hardware is a major problem in the field of embedded development. An operating system addresses this problem by introducing a *separation of concerns*. There are three agents at play in making an application portable. These are the device manufacturer, the kernel, and the application writer. Distinguishing their responsibilities clearly is of prime importance to creating portable and efficient applications.

The device manufacturer aims to establish an interface to the programmer for working with their system, which is independent from hardware and can therefore remain fixed even if some peripherals change. The application programmer, in aiming to make their application portable, strictly abides by the interface provided by the manufacturer. The kernel can best be envisioned as the plumbing that holds all of this together, permitting the device manufacturer to plug in their interface on one side, the programmer to plug in their application on the other, and enabling the two to talk through sharing messages.

The work of the device manufacturer culminates in a *driver* that is shipped together with the kernel on their boards. The application programmer can work away with the API, blissfully unaware of any changes that the hardware may incur. When the thermometer peripheral is replaced, the manufacturer must change the driver and ship the recompiled kernel on their new boards, but this makes no difference to the programmer - their code runs just the same.

Several operating systems, including Mike Spivey's *Phos*, provide a framework for achieving such a separation, but require the applications to be compiled together with the kernel. This exposes the programmer to unnecessary complexity and increases the size of the resulting binaries significantly, thus hindering the sharing of applications. We aim to provide a solution that allows for separate compilation of the applications and kernel to negate these drawbacks. Additionally, such a solution would be very easy to install by the device manufacturers on the boards as firmware.

Finally, it is oftentimes a good idea to divide applications into smaller, independent applications. Again referring to the example above, the programmer might realise that the code that calculates the temperature variance is quite complicated and easy to get wrong. There might be complex algorithms involved whose naive implementation by our him has resulted in a bug ridden and overly slow code. The programmer might thus want to delegate the responsibility of calculating variances to a different, specialised application, which he can pull off the internet and integrate with his existing code. The advantage of splitting up applications and compiling them separately is an increase in maintainability and modularity. For example, whenever an improved version of the calculator program appears, the programmer can replace his existing copy in memory with the new one without making any changes to the rest of the code.

1.2 Requirements

To be considered successful, our solution needs to satisfy the following requirements:

- Separate application code from the hardware.
- Separate application code from the kernel.
- Allow separate applications to live in memory and interact.
- Achieve real-time performance.
- Provide example drivers for some EFM32HG-SK peripherals.
- Provide tools to develop applications for the operating system.
- Work on the EFM32HG Starter Kit:
  - Fit within 64 kB of flash memory.
  - Use at most 8 kB of SRAM.

1.3 Contribution

To address these requirements, this project presents hackOS - a preemptive, multi-tasking, real-time operating system for embedded devices. It can be divided up into five main parts:

- The kernel.
- The drivers.
- The system tasks.
- Board support.
- The developer tools.

The kernel is the core of the operating system, providing a platform for processes to run on, hardware abstraction, and services through system calls. Its main responsibilities are booting the system, scheduling and running processes, providing access to kernel modules through system calls, enabling inter-process communication via message passing, handling exceptions and implementing a mechanism to bind peripheral interrupts to handler processes.

The drivers are pieces of code that provide an interface to access hardware components, abstracting away the gory details of their implementation. We distinguish between the *low level drivers*, which are very closely tied to the particular architecture of their device or otherwise critical to the operating system and thus need to be compiled alongside the kernel, and the *high level drivers*, which rely on their machine-level counterparts to provide an end-to-end encapsulation of the components they are handling. The first class is static and cannot be modified without editing the kernel source itself. Of these we provide three - the USB CDC, Serial I/O and Temperature Sensor drivers. The higher level drivers run as processes, communicating with the low level drivers and other processes through message passing. They are more concerned with the abstract functionality of devices, and so do not generally need to be tailored to any specific one. High lever drivers allow device manufacturers to provide an abstract interface to the components, and thus are essential in separating application code from the hardware. We provide three high level drivers as the led, gpio and tempsens system tasks.

System tasks are processes that are critical to the kernel, but are otherwise removed enough from it so as to be semi-independent. They are compiled along with the kernel and thus share its address space, but are provided with their own stack and a slot in the process table. They run as regular processes, able to make system calls and share information with any other process. System tasks always start first on boot, and hackOS provides a simple interface to add or change them. We provide five system tasks - led, gpio, tempsens, idle and loader.

Board support is a set of files specific to the SoC on which the operating system is installed that initialises the hardware prior to boot time, defines the basic exception handlers specific to it, and populates the vector table. We have made an effort to limit most of the work associated with porting hackOS to another platform to the board support module, but only provide an implementation for the EFM32HG starter kit.

The developer tools are a collection of header files, Makefiles and scripts that provide a platform to develop hackOS applications on. They provide startup services and system calls to the applications, and a way to interface with the loader through USB.

The full code of hackOS can be found online at github.com/depevlad/hackOS.

1.4 Outline

In the background, we go over knowledge that one should have before dissecting the operating system. This is followed by a description of the implementation in the hackOS: A Real-Time Operating System section. Following this, the evaluation section explains how the operating system was tested. In the conclusion, we go over a summary of the operating system, my reflections on writing it, and lay the groundwork for future improvements. The Appendix contains some code that, while not crucial to understanding the project, I found interesting enough to put on display. Finally, the Bibliography contains a list of sources which inspired hackOS and guided its implementation.

2 Background

2.1 Preliminaries

To start understanding hackOS, we must begin with some fundamental definitions from the field of embedded development:

Device	A piece of hardware with a dedicated function.
Chip / IC	A set of electronic circuits on one single, small piece of silicon.
Peripheral	An external device which provides input and output to the computer.
Microcontroller / MCU	A CPU, together with basic I/O and memory, on a single chip.
Embedded System	An MCU with a dedicated function within a larger electrical system.
System on a chip (SoC)	An MCU together with more advanced peripherals.

Thus, the EFM32HG starter kit on which this operating system was designed is a SoC. It features an EFM32HG microcontroller and several additional peripherals like AES, USART and an LCD display. The microcontroller is made up of an ARM Cortex-M0+ CPU, I/O pins and a small (8kB) amount of SRAM. The starter kit itself is not an embedded system, but the microcontroller could be were it hooked up to the radio of a car.

Moving forward, it is helpful to introduce the concepts of program and process. A *program* is a group of instructions to carry out a specific task, while a *process* is a program in execution. As such, the process contains additional information like the current instruction and stack pointer.

An *operating system* is a piece of software that runs directly on the hardware, and serves three main purposes. It manages the hardware, providing a layer of abstraction on top of it and allowing processes to communicate with it using an interface that is independent of the physical details of the implementation. It manages system resources, for example chopping up memory into blocks and handing them to processes only when needed. Finally, it provides services to the processes that run on it. These services can include sleep, connecting to hardware, or messaging other processes.

2.2 Interrupts & Exceptions

An exception is an input signal to the CPU that some task requires immediate attention. They can be triggered by both hardware and software and serve as the main way to implement asynchronous operations in computer systems. Exceptions can be further divided into internal exceptions and interrupts. The former pertain to the CPU itself and serve various purposes. The ones that interest us for operating systems are:

Exception	Purpose
SVC (Supervisor Call)	Used by applications to request access to system services
HardFault	Indicates failure of hardware
PendSV	Used to request a context switch
SysTick	Periodic, configurable exception for system maintenance

Interrupts are exceptions triggered by peripherals. They are programmable, and their meaning is specific to the system in which they are implemented. For example, the EFM32HG starter kit uses interrupt line 2 for USB, and interrupt lines 1 and 6 for GPIO.

Exception handlers are pieces of code that are attached to an exception and service it when it occurs. To follow the exception mechanism, imagine an application that is displaying some image on the LCD. As it is running, the programmer presses a button on the board. This triggers an interrupt, which the CPU receives on one of its lines, and which prompts it to immediately stop what it was doing. It stores its state at the moment it was halted in RAM and looks at a pre-defined position in memory for the appropriate handler. It then starts executing the instructions in the handler, and in this case toggles an LED. After finishing, the CPU automatically restores its state from memory and resumes execution where it left off. To the application that was interrupted, this is transparent. It knows nothing about what happened while it was interrupted, or even that it was interrupted at all.

The section of memory where the addresses of the handler functions are kept is called the *vector table*. It is typically located at the very start of memory and may also contain other information about the application.

2.3 ARM Cortex M0-Plus

The ARM Cortex M0+ is a 32-bit microprocessor based on the ARMv6M architecture that runs at 25MHz. It uses the Thumb instruction set, which is a version of the ARM instruction set that is compactified to fit within 16 bits. It features 16 general purpose registers, R0-R12 along with PC, LR and PSR. Through the use of Thread mode, the Cortex M0+ offers hardware support for maintaining two stacks simultaneously. The main stack pointer (MSP) is intended for use within the kernel and exception handlers, while the process stack pointer (PSP) is meant to be used by running applications. The Cortex-M0+ supports up to 32 programmable peripheral interrupts.

To not sabotage the execution of an interrupted program, the CPU is responsible for ensuring that its state (defined as the totality of registers) immediately after an exception is identical to the state immediately before. To this end, when entering an exception, the Cortex-M0+ automatically saves the caller-saved registers on the active stack before executing the handler. According to the ARM ABI, these are registers R0-R3, R12, LR, PC and PSR. The remaining registers, by definition of being callee-saved, are always preserved during the execution of a function. Since a handler is ultimately just a function, any correctly compiled handler is guaranteed not to disrupt the state.

2.4 The EFM32HG Starter Kit

hackOS was developed and tested on the EFM32HG starter kit sold by Silicon Labs. The kit features an EFM32HG microcontroller along with 8kB of RAM, 64kB of flash memory, two microUSB ports, two push buttons and programmable LEDs and an LCD display. It features several additional peripherals like general-purpose IO (GPIO), DMA, timers, a watchdog and USART.

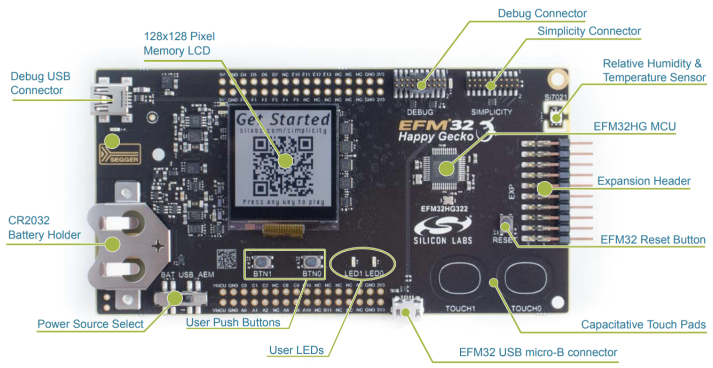


Figure 1: The EFM32HG Starter Kit

2.5 Real-Time Operating Systems

A *real-time operating system (RTOS)* is an operating system intended to serve real-time applications - those that process data as it comes in. Real-time operating systems offer a guarantee that the time to process an input stimulus is less than the time lapsed until the next stimulus of the same type occurs. RTOS can be *event-driven*, which switch tasks only when an event of a higher priority needs servicing, or *time-sharing*, which switch tasks on a regular clock interrupt.

There are several real-time operating systems for embedded devices on the market, including *mbed-OS*, *uC-OS3* and Mike Spivey's *Phos*. The former offer support for a wide variety of boards, but they crucially do not offer access to the hardware through system calls, but through implementations of the hardware abstraction layer. This leaves application code reliant on the exact specifics of the hardware for which the HAL was written, and thus breaks portability if some components of the board are replaced. Phos does make use of system calls, but it suffers from the drawback of requiring application code to be compiled together with the kernel. hackOS fully decouples hardware services from the application code, and the application code from the kernel, therefore allowing for maximum code portability.

3 hackOS: A Real-Time Operating System for the EFM32HG

We explore the detailed design of the operating system components in the order in which they were laid out in the introduction.

3.1 Kernel

As the core of the operating system, the kernel provides a platform for processes to run on, hardware abstraction, and access to services through system calls. Its functionality can roughly be broken down into processes management and scheduling, context switching, the system call API, exception handling and message passing.

3.1.1 Processes

Processes are fundamental components of the operating system. They represent programs in execution, and as such are augmented with various data, such as a name, state, and a pointer to the process queued to run after the current. Processes run on their own stack, provided by the kernel at boot time and fixed in size. They make system calls to request services, which can include sleep, sending messages to each other, and yielding to their peers. They may serve as interrupt handlers by binding to a specific interrupt line. They are arranged in the process table, a kernel data structure.

```c
#define MAX_PROC_NAME 16
#define NUM_PROCS     16
#define IDLE_ID 0

struct Process {
        char      pName[MAX_PROC_NAME];  /* Process name. */
        uint8_t   pId;                   /* Process ID. */
        ProcState pState;                /* Process state. */
        uint32_t *pSp;                   /* Process current stack pointer. */
        void     *pStk;                  /* Pointer to the bottom of the process stack. */
        uint32_t  pStkSize;              /* Process stack size. */
        uint32_t  pSleepTime;            /* Remaining sleep time. */
        uint32_t  pPending;              /* Contains the logical OR of pending interrupts. */
        uint8_t   pAccept;               /* Who the process can receive messages from. */
        uint8_t   pPriority;             /* Process priority. */
        Message  *pMsg;                  /* Pointer to message. */
        Process  *pWaiting;              /* Processes waiting to send to this one. */
        Process  *pNext;                 /* Process that is scheduled after this one. */
};
```

Processes are identified by the operating system through their unique ID, while their name serves as an easy handle for the programmer. They administer their stack using the pSp, pStk and pStkSize fields. A process can be in one of multiple states - dead, ready to run, sending or receiving a message, sleeping or idling.

```c
typedef enum {
        DEAD,
        READY,
        SENDING,
        RECEIVING,
        SLEEPING,
        IDLING
} ProcState;
```

Processes are arranged into the *process table*, which is loaded at boot time by the kernel and used by it to keep track of their execution. In addition to this, the kernel keeps pointers to the idle process, and to the current running process. The hackOS process table supports up to 16 individual processes.

```
Process pTable[NUM_PROCS];
Process *crtProc;
Process *idleProc;
```

The idle process is a fallback for when no other processes are running. It does not do anything on its own, but instead loops and yields to another process as soon as it is able to. It is provided as a system task and discussed in Section 4.

3.1.2 Scheduler

The scheduler can be thought of as the core of the operating system, being the part which dynamically decides which programs to run. It sorts processes into one of three priority levels and provides a queue for each level, allowing programs to wait for their turn at execution. Level 0 is considered the highest priority and is reserved for handler processes. Level 1 contains regular processes, while level 2 contains only the idle process.

```
#define MAX_PRIORITY 3

typedef struct {
        Process *qHead;
        Process *qTail;
} procQueue;

procQueue readyQueue[MAX_PRIORITY];
```

Only processes which are ready to run (and idle) are placed into one of the scheduler queues. Importantly, the current process is not part of any queue. There are three main events that can happen inside the handler - either a process is made ready to run, rescheduled with a different priority, or the current process is updated. We will go over each of these situations in order.

When a process is rescheduled, it is taken out of its current queue and inserted into the maximum priority queue. Only ready processes may be rescheduled. A regular program generally needs to be rescheduled after becoming a handler through connecting to an IRQ line. As we would like the interrupts to be serviced as soon as possible, we allow handlers to preempt other processes.

```
void rescheduleProc(Process *p) {
        procQueue *q = &readyQueue[p->pPriority];

        /* Delete p from its queue. */
        if (q->qHead != NULL) {
                Process *prev = NULL;
                while (q->qHead != p) {
                        prev = q->qHead;
                        q->qHead = q->qHead->pNext;
                }
                if (prev != NULL)
                        prev->pNext = p->pNext;
        }

        p->pPriority = 0;
        makeReady(p);
}
```

To prepare a process to run, one must invoke the *makeReady* function. This happens when processes are started at boot time, when they wake up or have finished sending / receiving messages. The scheduler marks the process as ready and inserts it at the end of its priority queue.

```
void makeReady(Process *p) {
        if (p != idleProc) {
                p->pState = READY;
                p->pNext  = NULL;
                procQueue *q = &readyQueue[p->pPriority];
                if (q->qHead == NULL)
                        q->qHead = p;
                else
                        q->qTail->pNext = p;
                q->qTail = p;
        }
}
```

Finally, the scheduler can pick a new process to execute by calling the *chooseProcess* function. This function scans the queues in decreasing order of priority, dequeues the first available process and marks it as currently running. If the queues are all empty, the idle process becomes current.

```
void chooseProcess(void) {
        for (uint8_t pLevel = 0; pLevel < MAX_PRIORITY; pLevel += 1) {
                procQueue *q = &readyQueue[pLevel];
                if (q->qHead != NULL) {
                        crtProc  = q->qHead;
                        q->qHead = q->qHead->pNext;
                        return;
                }
        }
        crtProc = idleProc;
}
```

When starting the scheduler, before handing control to any of the processes we must ensure that the Cortex-M0+ is running in Thread mode in order to use separate kernel and process stacks. This is achieved through the *enterThreadMode* assembly routine.

```
/* Set PSP to @pSp and enter thread mode. */
void __attribute((naked)) enterThreadMode(uint32_t *pSp) {
        __asm__ volatile (
                "msr psp, r0      \n"
                "mov r0, #0x2     \n"
                "msr control, r0  \n"
                "isb              \n"
                "bx lr            \n"
        );
}
```

The following code starts the scheduler by initialising thread mode, setting the idle as the current process and ceding control to it:

```
void startScheduler(void) {
        crtProc = idleProc;
        enterThreadMode(crtProc->pSp);
        sysYield();
        idle();
}
```

3.1.3 Context Switching

A context switch is a transfer of control to a different body of code. It might happen in one of two situations:

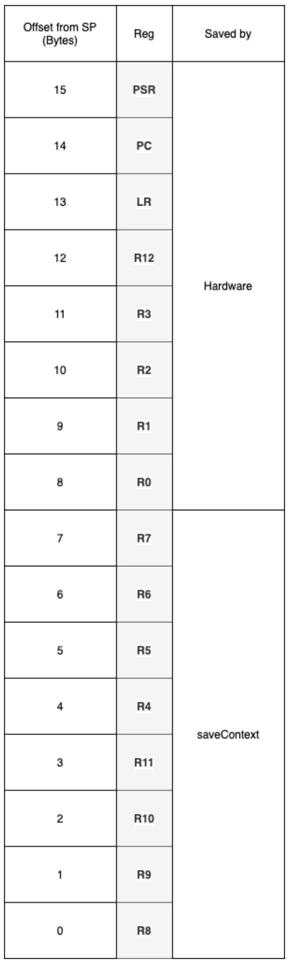
- Exception handling.
- Multitasking.

In the first case, control is temporarily ceded to the handler registered with the incoming exception. As described in the Background section, the Cortex-M0+ automatically saves caller-saved registers on the process stack when the exception is raised then invokes the body of the handler. As a properly compiled function, the exception handler is guaranteed to preserve the eight remaining callee-saved registers, and it restores the hardware saved registers from the stack on termination. Thus, the state of the interrupted process is kept consistent across the exception.

In the second case, no peripheral interrupts have happened but the running process may be put to sleep, wait for a message from another process or request to interrupt its execution itself by either yielding or using the PendSV system trap. The scheduler picks a new process to execute and places the incumbent process into a queue to be resumed later. Control is not given back directly to the interrupted program, so although the handler will preserve the callee-saved registers, there is no guarantee that other processes won't clobber them by the time the interrupted process is resumed. We must therefore augment the saving process by manually storing the eight remaining registers on the process stack before transferring control to the other process. We achieve this through the following assembly routine:

```
void __attribute((naked)) saveContext() {
        __asm__ volatile(
                ".syntax unified   \n"
                "mrs r0, psp       \n"
                "subs r0, #16      \n"
                "stm r0!, {r4-r7}  \n"
                "mov r4, r8        \n"
                "mov r5, r9        \n"
                "mov r6, r10       \n"
                "mov r7, r11       \n"
                "subs r0, #32      \n"
                "stm r0!, {r4-r7}  \n"
                "subs r0, #16      \n"
                "bx lr             \n"
        );
}
```

We call the set of all 16 general purpose registers saved on the process stack an *exception frame*. Exception frames are always 64 bytes long and hold the exact state of the CPU at the time a process was interrupted.



Offset from SP (Bytes)	Reg	Saved by
15	PSR	Hardware
14	PC	
13	LR	
12	R12	
11	R3	
10	R2	
9	R1	
8	R0	
7	R7	saveContext
6	R6	
5	R5	
4	R4	
3	R11	
2	R10	
1	R9	
0	R8	

Figure 2: Exception Frame Layout

Thus, at each point in time, all processes which are not currently running contain an exception frame on their stack, at the location pointed to by their stack pointer. When a process is resumed, restoration of their state begins with the manually saved registers. The following assembly routine performs this:

```
void __attribute__((naked)) restoreContext() {
        __asm__ volatile(
                ".syntax unified   \n"
                "ldm r0!, {r4-r7}  \n"
                "mov r8, r4        \n"
                "mov r9, r5        \n"
                "mov r10, r6       \n"
        "mov r11, r7       \n"
                "ldm r0!, {r4-r7}  \n"
                "msr psp, r0       \n"
                "bx lr            \n"
        );
}
```

Then, the SP register is changed to point to the upper half of the exception frame and the exception handler exits. The CPU automatically restores the eight remaining registers from its active stack pointer during the exit, and execution of the halted program is free to resume as if nothing had happened.

Processes may request to be swapped out at any time by invoking either the YIELD system call or by triggering the PendSV trap. In the first case, the context switch is carried out immediately through an SVC exception, while the in the latter the switch is delayed until the process is ready to run. The trap is therefore useful to ensure that there is no overlap between context switches and exception handlers.

```
/* Trigger the PendSV exception. */
void __attribute((noinline)) requestContextSwitch(void) {
    SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk;
}
```

3.1.4 System Calls

hackOS provides nine system calls that offer various services to the processes that call them. To make a system call, a process must execute the SVC instruction (which stands for Supervisor Call). The lower byte of this instruction can be set to any value and is used to specify which service the program is requesting. System calls may receive up to two additional arguments - an integer in R0, and a message pointer in R1.

```
typedef enum {
        SYS_YIELD,              /* Cede control to the next scheduled process. */
        SYS_EXIT,               /* Stop process. */
        SYS_SEND,               /* Send a message #1 to process #0. */
        SYS_SLEEP,              /* Sleep for #0 milliseconds. */
        SYS_PRINT,              /* Output the string pointed to by #0 over serial. */
        SYS_RECEIVE,            /* Receive a message from source #0 into address #1. */
        SYS_CONNECT_IRQ,        /* Register process as a handler for IRQ #0. */
        SYS_DISCONNECT_IRQ,     /* Free the process as a handler for IRQ #0. */
        SYS_ACKNOWLEDGE_IRQ,    /* Signal to the OS that IRQn #0 has been dealt with. */
} SystemCalls;
```

The kernel provides system call stubs for system tasks:

```
#define NOINLINE __attribute((noinline))
#define SYSCALL(n) __asm__ volatile ("svc %0" :: "i"(n));
```

```
void NOINLINE sysExit(void)                                   { SYSCALL(SYS_EXIT); }
void NOINLINE sysYield(void)                                  { SYSCALL(SYS_YIELD); }
void NOINLINE sysSleep(void)                                  { SYSCALL(SYS_SLEEP); }
void NOINLINE sysConnectIrq(uint8_t IRQn)                     { SYSCALL(SYS_CONNECT_IRQ); }
void NOINLINE sysDisconnectIrq(uint8_t IRQn)                  { SYSCALL(SYS_DISCONNECT_IRQ); }
void NOINLINE sysAcknowledgeIrq(uint8_t IRQn)                 { SYSCALL(SYS_ACKNOWLEDGE_IRQ); }
void NOINLINE sysSend(uint8_t dst, Message *msg)              { SYSCALL(SYS_SEND); }
void NOINLINE sysReceive(uint8_t idSrc, Message *msg) { SYSCALL(SYS_RECEIVE); }
```

The system call handler is invoked by the SVC handler after saving the full exception frame of the calling process on the stack. This is necessary since the current program may change as a result of the system call (as in the case of SYS_YIELD). It receives just one parameter - a pointer to the bottom of the exception frame. It fetches the system call number and the arguments, if any, from the exception frame, and takes the appropriate action. It returns the stack pointer of the new current process.

```
#define INSTR_SIZE 2 /* Set for Thumb mode. */

uint32_t *syscallHandler(uint32_t *pSp) {
        uint8_t  svcId = ((uint8_t*)(pSp[SAVED_PC]))[-(INSTR_SIZE)];
        uint32_t arg   = pSp[SAVED_R0];
        Message *msg   = (Message*)pSp[SAVED_R1];
        crtProc->pSp = pSp;

        switch(svcId) {
                case SYS_EXIT:
                        crtProc->pState = DEAD;
                        chooseProcess();
                        break;

        ...

                default:
                        kPrintf("Invalid SVC %d!\n", svcId);
                        break;
        }

        return crtProc->pSp;
}
```

3.1.5 Message Passing

Message passing is the mechanism provided by the operating systems for communication between processes, or between a process and hardware.

The *data* of a message is four bytes long. Processes are allowed to use these four bytes to store either an integer, a pointer or four individual bytes. By passing allowing messages to contain a pointer, hackOS effectively permits arbitrarily sized messages.

```
typedef union {
        uint32_t mInt;
        void    *mPtr;
        struct {
                uint8_t mBw;
                uint8_t mBx;
                uint8_t mBy;
                uint8_t mBz;
        };
} mData;
```

A *message* consists of a piece of data, together with a programmable type and the ID of the sending process. Type 0xff is reserved for communications to and from the hardware.

```
#define M_HARDWARE 0xff

typedef struct {
        uint8_t mType;
        uint8_t mSender;
        mData   mDatum;
} Message;
```

Sending and receiving messages is handled through the SYS_SEND and SYS_RECEIVE system calls. Processes may specify where they who they want to receive message from through the pAccept field. Processes that try to send a message which is not immediately acknowledged, or to receive one when none has been transmitted, are marked to be in the SENDING and RECEIVING states respectively, and taken out of the scheduler.

When sending a message, the operating system first checks if the destination process is currently listening for any messages. If it is, it immediately accepts the message and is scheduled to run. Otherwise, the sender joins a queue of other processes waiting to send to the destination.

```
void sendMessage(uint8_t dst, Message *msg) {
        uint8_t idSrc = crtProc->pId;
        msg->mSender  = idSrc;
        Process *pDst = &pTable[dst];

        if (pDst->pState == RECEIVING &&
           (pDst->pAccept == ACCEPT_ANY || pDst->pAccept == idSrc)) {
                /* Receiver is waiting for us. */
                *(pDst->pMsg) = *msg;
                makeReady(pDst);
        } else {
                /* Sender must wait and join the receiver's queue. */
                crtProc->pState = SENDING;
                crtProc->pMsg   = msg;
                crtProc->pNext  = NULL;

                if (pDst->pWaiting == NULL) {
                        pDst->pWaiting = crtProc;
                } else {
                        /* Add sender to the back of the queue. */
                        Process *q = pDst->pWaiting;
                        while (q->pNext != NULL) q = q->pNext;
                        q->pNext = crtProc;
                }

                chooseProcess();
        }
}
```

When receiving a message, a process selects the first sender in its waiting list and accepts its transmission. The sending process is then added back into the scheduler. If no messages are pending, the process marks itself as RECEIVING and yields. Processes which handle interrupts first check for any messages from the hardware before examining the waiting list.

```
void receiveMessage(uint8_t idSrc, Message *msg) {
        /* See if any interrupts are pending. */
        if (crtProc->pPending && (idSrc == ACCEPT_ANY || idSrc == ACCEPT_HW)) {
                crtProc->pPending = 0;
                msg->mSender = ACCEPT_HW;
                msg->mType   = M_HARDWARE;
        }

        /* If not, pick the first suitable process from the queue. */
        Process *pPrev = NULL;
        for (Process *pSrc = crtProc->pWaiting; pSrc != NULL; pSrc = pSrc->pNext) {
                if (idSrc == ACCEPT_ANY || idSrc == pSrc->pId) {
                        /* Receive from this process. */
                        if (pPrev == NULL) {
                                /* First process in the queue was suitable. */
                                pSrc->pWaiting = pSrc->pNext;
                } else {
                        pPrev->pNext = pSrc->pNext;
                }

                *msg = *(pSrc->pMsg);
                msg->mSender = pSrc->pId;

                makeReady(pSrc);

                return;
            }
            pPrev = pSrc;
        }

        /* Wait. */
        crtProc->pState   = RECEIVING;
        crtProc->pAccept  = idSrc;
        crtProc->pMsg     = msg;
        chooseProcess();
}
```

3.1.6 Memory

The memory module provides an interface for interacting with the flash. On the EFM32HG, the flash is divided up into 64 pages of 1024 bytes. 48 of these are reserved for the kernel, and the remaining 16 can be populated with applications.

```
#define USER_START_PAGE 48
#define USER_END_PAGE   64
#define PAGE_SIZE     1024
```

We provide a function *writeFlashPage* to allow for writing a flash page. As the page that is being written to may contain the text of the method itself, the function is run from RAM.

```
bool __attribute((section(".ram"))) writeFlashPage(uint32_t* dst, uint32_t* src) {
    MSC_Status_TypeDef status;
    CORE_DECLARE_IRQ_STATE;
    CORE_ENTER_ATOMIC();
    MSC->LOCK = MSC_UNLOCK_CODE;
    status = MSC_ErasePage(dst);
    if (status == mscReturnOk) {
            status = MSC_WriteWord(dst, src, FLASH_PAGE_SIZE);
    }
    MSC->LOCK = 0;
    CORE_EXIT_ATOMIC();
    return (status == mscReturnOk);
}
```

Programs are stored and run from the flash. Their start addresses must be aligned to the page size. Furthermore, programs can only span whole numbers of pages. For example, a program can start at addresses 0xe000 and take up all of page 56 if it is 479 bytes long, or start at address 0xe800 and take up all of pages 58 and 59 if it is 1113 bytes long. The first 32 bytes are not executable; instead they contain information about the program and run-time requirements.

Bytes	Function	Example
0-3	Magic Number	0x01234567
4-7	Entry Point	0xe479
8-23	Name	blink
24-25	RAM Space	256
26-27	Stack Space	256
28-31	Reserved	-

The magic number is a four byte constant that identifies the start of a program. When the operating system boots, it scans all the pages and loads programs from those whose first word is the magic number. The name is used to identify the program to humans, and is provided at load time (it is not encoded in the binary itself). The RAM and stack space fields indicate to the operating system how much space it should allocate for the program's stack, as well as the .data and .bss sections. Finally, the last four bytes are reserved for future use.

3.1.7 Interrupts & Exceptions

This module implements handlers for the SVC, HardFault, SysTick and PendSV exceptions, as well as some of the peripherals supported by the EFM32HG starter kit.

To account for the possibility of the current process changing during their handling, the SVC and PendSV exception handlers save the full exception frame on the process stack, as explained in the Context Switch section. They also restore the callee-saved registers on exit. This functionality can be applied onto the handlers by the *__wrap* macro:

```
#define __wrap(f)                          \
        __asm__ volatile("push {lr, r0}");    \
        saveContext();                        \
        f();                                  \
        restoreContext();                     \
        __asm__ volatile("pop {r0, pc}");     \

void __attribute((naked)) SVC_Handler()    { __wrap(syscallHandler)   }
void __attribute((naked)) PendSV_Handler() { __wrap(cxtSwitchHandler) }
void HardFault_Handler(void)                 { kPrintf("HardFault!\n"); }
```

SysTick exceptions are configured to happen every millisecond. The SysTick handler scans for any sleeping processes, and decrements their remaining sleep time. When this time reaches 0, the processes are woken up.

```
void SysTick_Handler(void) {
        for (uint8_t i = 0; i < NUM_PROCS; i++) {
                if (pTable[i].pState == SLEEPING) {
                        if (pTable[i].pSleepTime == 0) {
                                /* Wake up. */
                                makeReady(&pTable[i]);
                        }
                        pTable[i].pSleepTime -= 1;
                }
        }
}
```

hackOS allows processes to bind themselves to interrupt lines as handlers, and receive signals from the hardware through messages. This can be accomplished through the SYS_CONNECT and SYS_DISCONNECT system calls.

```
/* Bind the current process to interrupt line IRQn. */
void irqConnect(uint8_t IRQn) {
        uint8_t pId = crtProc->pId;
        if (irqHandler[IRQn] == 0) {
                irqHandler[IRQn] = pId;
                NVIC_EnableIRQ(IRQn);
        }
}

/* Release the IRQn interrupt line. */
void irqDisconnect(uint8_t IRQn) {
        uint8_t pId = crtProc->pId;
        if (irqHandler[IRQn] == pId) {
                irqHandler[IRQn] = 0;
                NVIC_DisableIRQ(IRQn);
        }
}
```

Interrupt messages must be acknowledged by the handler processes once their cause has been dealt with. By default, no interrupt can trigger unless it is bound to a process. All signals that arrive before the acknowledgement are discarded.

The vector table interrupt handler methods wrap their signals in a message, and invoke the *toIRQHandler* method to forward this method to the registered handler process.

```c
/* Low level handler method for the odd GPIO interrupt. */
void GPIO_ODD_IRQHandler(void) {
        uint32_t iFlags = GPIO_IntGet() & GPIO_ODD_MASK;
        GPIO_IntClear(iFlags);
        Message msg = {
                .mSender = GPIO_ODD_IRQn,
                .mType   = M_HARDWARE,
                .mDatum  = iFlags
        };
        toIRQHandler(GPIO_ODD_IRQn, msg);
}

/* Forwards a message from the hardware to the registered handler of interrupt @IRQn. */
void toIRQHandler(uint8_t IRQn, Message msg) {
         if (IRQn < NUM_IRQS) {
                uint8_t idHandler = irqHandler[IRQn];
                if (idHandler != 0) {
                        NVIC_DisableIRQ(IRQn);
                        Process *pHandler = &pTable[idHandler];
                        /* Send an interrupt message to the handler. */
                        if (pHandler->pState == RECEIVING &&
                             (pHandler->pAccept == ACCEPT_ANY ||
                              pHandler->pAccept == ACCEPT_HW)) {
                                /* The handler is waiting for a message from the hardware. */
                                *(pHandler->pMsg) = msg;
                                rescheduleProc(pHandler);
                                requestContextSwitch();
                        } else {
                                /* Mark the interrupt as pending. */
                                pHandler->pPending |= (1 << IRQn);
                        }
                } else {
                        kPrintf("Bad IRQn %d!\n", IRQn);
                }
        }
}
```

3.1.8 Booting

At boot time, the kernel relies on two external methods which should be linked together with it. They are *boardInit*, which performs sets up the hardware, peripherals and system exceptions, and *useLoaderMode*, which specifies which mode boot should occur in. In the case of the EFM32HG starter kit, loader mode is set to true if push button 0 is pressed during reset. These functions are provided in the *Board Support* module.

The kernel keeps a system task table which is defined at compile time.

```
typedef void (*fPtr) (void);

typedef struct {
        char    *tName;   /* Name of the system task. */
        fPtr     tBody;   /* Pointer to the body of the system task. */
        uint16_t tSize;   /* System task stack space, in bytes. */
} Task;

#define makeTask(_tName, _tBody, _tSize) {     \
                .tName = _tName,                \
                .tBody = _tBody,                \
                .tSize = _tSize                 \
}                                              \

Task sysTasks[] = {
        makeTask("idle", &idle, 128),
        makeTask("loader", &loader, 512),
        makeTask("gpio", &gpioHandler, 256),
        makeTask("leds", &ledHandler, 128)
};
```

Every boot starts with clearing the process table. The stack for each system task is allocated incrementally starting from the top of the kernel stack.

```
memset(pTable, 0, sizeof(NUM_PROCS * sizeof(Process)));
```

Loader mode is a feature of hackOS that allows starting up with just the loader task running. It gives programmers an easy way to load applications onto the board. If any applications crash on startup, the system will reboot forever. Loader mode also serves as a solution to this issue by allowing developers to overwrite these applications. To enter loader mode on the EFM32HG starter kit, hold PB1 as the system is starting up.

```
if (loaderMode) {
        /* Only start idle and loader. */
        for (uint8_t i = 0; i < 2; i++) startTask(i);
        kPrintf("Booted in loader mode.\n");
        startScheduler();
        return;
}
```

If, on the other hand, the boot is regular, then all system tasks *except* the loader are started. Following this, all applications are started.

```
uint8_t nSysTasks = sizeof(sysTasks) / sizeof(Task);
for (uint8_t i = 0; i < nSysTasks; i++) {
        startTask(i);
        kPrintf("Started task %s\n", sysTasks[i].tName);
}

crtId = nSysTasks;
for (uint8_t i = USER_START_PAGE; i < USER_END_PAGE; i++) {
        uint8_t *pStart = i * PAGE_SIZE;
        if (*((uint32_t*)pStart) == MAGIC_NR)
            startProgram(pStart);
}
```

When starting a process, we distinguish between the idle process and other processes. We fill the appropriate process table entry and create a dummy exception frame on the stack. This is mostly zeroed out, but most importantly the stacked PSR, PC and LR are set to values specific to each process. Using this mechanism, we can start a process as if it were being 'resumed' from a context switch that happened at the very start of its execution.

```
void startProc(uint8_t pId, char *name, void (*pBody)(void),
               void *pStk, uint32_t stkSize) {
        if (pId == 0) {
                /* Start idle task. */
                idleProc = &pTable[IDLE_ID];
                fillPTable(idleProc, 0, "idle", pStk, stkSize);
                idleProc->pState    = IDLING;
                idleProc->pPriority = MAX_PRIORITY;
        } else {
                Process *p = &pTable[pId];

                if (p->pState != DEAD)
                        kPrintf("pId for process already taken! (%d)\n", pId);

                fillPTable(p, pId, name, pStk, stkSize);

                /* Create a fake exception frame on the process stack. */
                uint32_t *new_pSp = p->pSp - 16;
                memset(new_pSp, 0, 64);
                new_pSp[SAVED_PSR] = INIT_PSR;
                new_pSp[SAVED_PC]  = (uint32_t)pBody;
                new_pSp[SAVED_LR]  = (uint32_t)exit;
                p->pSp = new_pSp;

                makeReady(p);
        }
}
```

Filling the process table is straightforward:

```
void fillPTable(Process *p, uint8_t pId, char *name, void *pStk, uint32_t stkSize) {
        p->pId        = pId;
        p->pState     = READY;
        p->pSp        = (uint32_t*)(pStk + stkSize);
        p->pStk       = pStk;
        p->pStkSize   = stkSize;
        p->pSleepTime = 0;
        p->pWaiting   = NULL;
        p->pPending   = 0;
        p->pPriority  = 0;
        p->pNext      = NULL;
        p->pAccept    = ACCEPT_ANY;
        p->pMsg       = NULL;
        p->pName[MAX_PROC_NAME - 1] = 0;
        strncpy(p->pName, name, MAX_PROC_NAME - 1);
        /* Clear the stack. */
        memset(pStk, 0, stkSize);
}
```

When starting a program from memory, we load the appropriate values from the flash into variables and then invoke *startProcess*.

```
void startProgram(uint8_t *pStart) {
        uint8_t *addrBody     = pStart +  4;
        uint8_t *addrName     = pStart +  8;
        uint8_t *addrDataSize = pStart + 24;
        uint8_t *addrStkSize  = pStart + 26;

    /* Get stack start and size. */
        void *stkStart   = (void*)((uint8_t*)__break + *((uint16_t*)addrDataSize));
        uint16_t stkSize = *((uint16_t*)addrStkSize);
        startProc(
                crtId,
                (char*)addrName,
                *((void (*) (void))addrBody),
                stkStart,
                stkSize
        );

    /* Update break, next available process ID. */
        __break = (void*)((uint8_t*)stkStart + stkSize);
        crtId++;
}
```

Starting a task is a thin wrapper around starting a process:

```
void startTask(uint8_t tId) {
        startProc(
                tId,
                sysTasks[tId].tName,
                sysTasks[tId].tBody,
                __break,
                sysTasks[tId].tSize
        );
        __break += sysTasks[tId].tSize;
}
```

Finally, once the program table has been populated with system tasks and user processes, we start the scheduler.

```
startScheduler();
```

3.2 Drivers

The drivers are pieces of code that provide an interface to access hardware and abstract away the details of its implementation. They can be broken down into the hardware-dependent low level drivers, which we will study in this section, and the more abstract high level drivers, which we present as system tasks. Low-level drivers are always compiled together with the kernel.

3.2.1 USB

The USB driver is based on the *emusb* middleware provided by Silicon Labs. It interfaces with the other parts of the kernel through two global variables - a global variable *gloXferred* that is set to the number of bytes read in the last session, and a byte array *usbRxBuffer* which contains the payload. We present the important parts of its implementation here:

```c
#define CDC_USB_RX_BUF_SIZ 128

uint8_t usbRxBuffer[CDC_USB_RX_BUF_SIZ] __attribute((aligned(4)));
volatile uint32_t gloXferred;

/* Call this function when receiving a frame. */
int receiveData(USB_Status_TypeDef usbStatus, uint32_t xferred, uint32_t remaining) {
        if (usbStatus == USB_STATUS_OK && xferred > 0) {
                gloXferred = xferred;
                USBD_Read(CDC_EP_DATA_OUT, (void*)usbRxBuffer,
                          CDC_USB_RX_BUF_SIZ, receiveData);
        }
        return USB_STATUS_OK;
}
```

3.2.2 Serial I/O

The serial I/O driver is the simplest. It provides a single function *kPrintf* which takes a format followed by a variable number of arguments, and outputs the formatted string over a USART connection.

```c
#define SERIAL_BUF_SIZE 128

char serialOutBuffer[SERIAL_BUF_SIZE];

void serialOut(char *message) {
    for (uint32_t i = 0; i < strlen(message); i++ ) {
        USART_Tx(USART1, message[i]);
    }
}

void kPrintf(char *fmt, ...) {
        va_list args;
        va_start(args, fmt);
        /* Pass the va_list to sprintf. */
        vsprintf(serialOutBuffer, fmt, args);
        serialOut(serialOutBuffer);
}
```

3.2.3 Temperature Sensor

We provide a modified version of the Silicon Labs driver. It uses the analogue thermometer on the chip and the ADC peripheral to provide reliable temperature measurements inside the CPU. It provides just one function, *measureTemperature*, which reports the current measurement in Celsius.

```
volatile bool conversionDone = false;

int32_t measureTemperature(void) {
  /* Start ADC conversion and wait until it is complete */
  conversionDone = false;
  ADC_Start(ADC0, adcStartSingle);
  while (!conversionDone) { /* Wait. */ }
  /* Calculate temperature. */
  return convertToCelsius(ADC_DataSingleGet(ADC0));
}
```

3.3 System Tasks

System tasks are processes that are critical to the kernel, but otherwise removed enough from it so as to be semi-independent. They share an address space with the kernel, but have their own stack and are otherwise managed as regular processes.

The first three system tasks are high level drivers. They wrap the functionality of their low level counterparts in a message interface, thus permitting other processes to communicate with the hardware without being compiled together with the drivers.

3.3.1 High-Level GPIO Driver

The high level GPIO driver interfaces with the hardware and other processes through messages, and provides two basic functionalities - it allows processes to register callback functions for button presses, and dispatches these callbacks when the buttons are pressed.

```
#define PB_0_PIN 9
#define PB_1_PIN 10

#define IS_SET(x, pos) ((x & (1 << pos)) != 0)

typedef void (*pbCallback) (void);

pbCallback pbCallbacks[2];
Message msg;

void gpioHandler(void) {
        sysConnectIrq(GPIO_EVEN_IRQn);
        sysConnectIrq(GPIO_ODD_IRQn);
        while (1) {
                /* Get an interrupt message and acknowledge it. */
                sysReceive(ACCEPT_ANY, &msg);
                /* Either dispatch a callback... */
                if (msg.mType == M_HARDWARE) {
                        sysAcknowledgeIrq(msg.mSender);
                        uint32_t iFlags = msg.mDatum.mInt;
                        if (IS_SET(iFlags, PB_0_PIN)) {
                                if (pbCallbacks[0] != NULL) {
                                        pbCallbacks[0]();
                                }
                        }
                        if (IS_SET(iFlags, PB_1_PIN)) {
                                if (pbCallbacks[1] != NULL) {
                                        pbCallbacks[1]();
                                }
                        }
                }
                /* Or register one. */
                else {
                        uint8_t pbID = msg.mType;
                        pbCallback f = msg.mDatum.mPtr;
                        pbCallbacks[pbID] = f;
                }
        }
}
```

3.3.2 High-Level LED Driver

The high level LED driver receives messages from other processes and acts upon the LEDs on the board using information from these messages. For example, processes can send a message with type 0 and mInt set to the index of the LED to toggle it.

```
#define NUM_LEDS 2

#define LEDS_PORT gpioPortF
#define LED_0_PIN 4
#define LED_1_PIN 5

Message msg;

void ledToggle(uint8_t ledId) {
        uint8_t pinId = LED_0_PIN;
        if (ledId == 1) pinId = LED_1_PIN;
        GPIO_PinOutToggle(LEDS_PORT, pinId);
}
```

```
void ledHandler(void) {
        /* Initialise the pins. */
        GPIO_PinModeSet(LEDS_PORT, LED_0_PIN, gpioModePushPull, 0);
        GPIO_PinModeSet(LEDS_PORT, LED_1_PIN, gpioModePushPull, 0);
        while (1) {
                sysReceive(ACCEPT_ANY, &msg);
                /* Toggle a LED. */
                if (msg.mType == 0) {
                        uint8_t ledId = msg.mDatum.mInt;
                        assert(0 <= ledId && ledId < NUM_LEDS);
                        ledToggle(ledId);
                }
        }
}
```

3.3.3 High-Level Temperature Driver

The high level temperature driver treats all incoming messages of type 0 as a request to read the temperature on the chip. It uses its low level counterpart to get this reading, and sends it back to the process which asked.

```
Message msg;

void tempSensor(void) {
        while (1) {
                sysReceive(ACCEPT_ANY, &msg);
                if (msg.mType == 0) {
                        /* Read temperature and send it back. */
                        msg.mDatum.mInt = measureTemperature();
                        sysSend(msg.mSender, &msg);
                }
        }
}
```

The final two system tasks are not drivers, but processes which help the execution of the operating system. These are the idle task, and the loader.

3.3.4 Idle

The idle task loops and yields whenever it has the opportunity. It is a fallback for when no other processes are running.

```
void idle(void) {
        while (1) {
                /* Wait for an event or interrupt. */
                __asm__ volatile ("wfe");
                requestContextSwitch();
        }
}
```

3.3.5 Loader

The loader is responsible for putting programs into the flash. To this extent it uses the USB CDC driver. The loader loops until it receives a special USB frame that denotes the start of a binary transmission. The special frame is 32 bytes long, starts with the program magic number, and contains all the run-time information that will be stored in the flash alongside the program. It also contains the page at which the binary must be loaded.

```
void loader() {
        while (1) {
                if (gloXferred > 0) {
                        uint32_t pattern = *((uint32_t*)(usbRxBuffer));
                        if (pattern == MAGIC_NR) {
                                /* Start of a binary. The current frame contains:
                                 *   0-3   : MAGIC NR
                                 *   4-7   : ENTRY
                                 *   8-23  : NAME
                                 *   24-25: RAM SPACE
                                 *   26-27: STACK SPACE
                                 *   28-28: PAGE ID
                                 */
                                uint8_t pageId = usbRxBuffer[28];
                                loadBinary(pageId);
                        }
                        gloXferred = 0;
            }
        }
```

The *loadBinary* function assembles USB frames into a buffer, which it then copies into flash. To end a transmission, the host must send a frame that begins with the magic number.

```
void loadBinary(uint8_t pageId) {
        uint32_t *addr = (uint32_t*)(pageId * FLASH_PAGE_SIZE);
        gloXferred = 0;

        memset(mscBuffer, 0, sizeof(mscBuffer));
        memcpy(mscBuffer, usbRxBuffer, 28);
        uint32_t buffPos = 32;
        while (1) {
                if (gloXferred > 0) {
                        if (*((uint32_t*)usbRxBuffer) == MAGIC_NR) {
                                /* This marks the end of the binary. */
                                if (buffPos > 0) {
                                        writeFlashPage(addr, (uint32_t*)mscBuffer);
                                }
                                return;
                        }
                        else {
                                for (uint8_t i = 0; i < gloXferred; i++) {
                                        mscBuffer[buffPos] = usbRxBuffer[i];
                                        buffPos += 1;
                                        if (buffPos == FLASH_PAGE_SIZE) {
                                                writeFlashPage(addr, (uint32_t*)mscBuffer);
                                                addr += (FLASH_PAGE_SIZE / 4);
                                                buffPos = 0;
                                        }
                                }
                                gloXferred = 0;
                        }
                }
        }
}
```

3.4 Board Support

The board support module contains platform-specific code that initialises the hardware that the operating system runs on, as well as creating the vector table and defining the usable exception handlers. Board support is divided into three files. Most of the work in porting hackOS to a different platform is limited to this folder.

startup.c initialises clocks, drivers, peripherals and interrupts, and implements the *boardInit* and *useLoaderMode* functions. We provide this file in the Appendix.

3.5 Developer Tools

The developer tools are a collection of files that provide a platform for building hackOS applications. They may be divided into a library of header files, default make and linker scripts, and a script to transfer applications to the board through USB.

3.5.1 Library

We provide two header files that supply the message data type and the system call API to programs.

os_msg.h

```
#define uint8_t  unsigned char
#define uint32_t unsigned

#define ACCEPT_ANY 127
#define ACCEPT_HW  126
#define M_HARDWARE 0xff

typedef union {
        uint32_t mInt;
        void    *mPtr;
        struct {
                uint8_t mBw;
                uint8_t mBx;
                uint8_t mBy;
                uint8_t mBz;
        };
} mData;

typedef struct {
        uint8_t mType;
        uint8_t mSender;
        mData   mDatum;
} Message;
```

os_syscalls.h

```
#include "os_msg.h"

typedef enum {
    SYS_YIELD,
    SYS_EXIT,
    SYS_SEND,
    SYS_SLEEP,
    SYS_PRINT,
    SYS_RECEIVE,
    SYS_CONNECT_IRQ,
    SYS_DISCONNECT_IRQ,
    SYS_ACKNOWLEDGE_IRQ,
} SystemCalls;

#define SYSCALL(n) __asm__ volatile ("svc %0" :: "i"(n));
#define NOINLINE __attribute((noinline))

void NOINLINE sysExit(void)                             { SYSCALL(SYS_EXIT); }
void NOINLINE sysYield(void)                            { SYSCALL(SYS_YIELD); }
void NOINLINE sysSleep(uint32_t mSec)                   { SYSCALL(SYS_SLEEP); }
void NOINLINE sysPrint(char *fmt)                       { SYSCALL(SYS_PRINT); }
void NOINLINE sysConnectIrq(uint8_t IRQn)               { SYSCALL(SYS_CONNECT_IRQ); }
void NOINLINE sysDisconnectIrq(uint8_t IRQn)            { SYSCALL(SYS_DISCONNECT_IRQ); }
void NOINLINE sysAcknowledgeIrq(uint8_t IRQn)           { SYSCALL(SYS_ACKNOWLEDGE_IRQ); }
void NOINLINE sysSend(uint8_t dst, Message *msg)        { SYSCALL(SYS_SEND); }
void NOINLINE sysReceive(uint8_t idSrc, Message *msg) { SYSCALL(SYS_RECEIVE); }
```

We also provide a startup file that initialises the program before calling main. It copies the .data segment, zeroes out .bss, calls *main* and places the entry point of the program at the start of its vector table.

startup.c

```
extern unsigned __data_start[],
                __data_end[],
                __bss_start[],
                __bss_end[],
                __etext[],
                main();

/* __reset - the system starts here. */
void __reset(void) {
    unsigned *p, *q;
    /* Copy data segment and zero out bss. */
    p = __data_start;
    q = __etext;
    while (p < __data_end) *p++ = *q++;
    p = __bss_start;
    while (p < __bss_end) *p++ = 0;
    main();
};

void *__vectors[] __attribute((section(".vectors"))) = { __reset };
```

3.5.2 Make and Linker Scripts

We provide a default Makefile and linker script to compile applications for hackOS. We instruct the compiler not to link the executable with *stdlib* to conserve space. Moreover, the application writer must define the exact places where their application will sit in flash, and where its RAM starts, in the linker script. These files can be found on the project Github.

3.5.3 USB Script

We provide a simple Python script to transfer applications to the board over USB. It takes as parameters the name of the binary to be sent, the index of the flash page where it will be placed, and the desired data and stack sizes. The script is also provided in the Appendix.

4 Evaluation & Testing

4.1 Evaluation

The operating system has succeeded in fitting within the desired memory limits. It takes up 40kB of flash, and up to 6kB of RAM when running. In addition to this, the *Board Support* module ensures compatibility with the EFM32HG Starter Kit, abstracting away the details of the hardware.

Our operating system achieves its goal of separating application code from the hardware through its nine system calls and its message passing system. Through the system call mechanism, it also achieves separation of the application code from the kernel, thus allowing the two to be compiled separately. It provides several low and high level drivers for the starter kit, including USB, Serial I/O, GPIO, LED and Temperature Sensor drivers. We achieved real-time performance through the pre-emption of lower priority processes by interrupt handlers. Through the memory module, boot mechanism and message passing, we have also achieved our goal of allowing multiple applications to run concurrently, and to interact.

Finally, we have provided a simple and clean toolset to write and load applications into hackOS. These contain header files for system calls and the message passing mechanism, application startup code, standard Makefiles and linker scripts, and a Python script for loading programs into memory.

4.2 Testing

4.2.1 Applications

We provide three applications that test the functionality of the operating system.

4.2.2 Blink

Blink toggles LED 0 every second. It tests the message passing system, the high level LED driver and the sleep mechanism.

```c
#include "../lib/inc/os_syscalls.h"

Message sendMsg;

/* ID of the LED handler process. */
const int ledID = 3;

void main(void) {
    /* Craft a message that toggles LED 0. */
    sendMsg.mType    = 0;
    sendMsg.mDatum.mInt = 0;

    while (1) {
        sysSleep(1000);
        sysSend(ledID, &sendMsg);
    }
}
```

4.2.3 Temperature

Temperature is an implementation of the example application described in the Motivation section. It reads the temperature every five seconds and outputs it over a serial connection. In doing so, it also tests the temperature drivers and the sleep mechanism. Note that the application developer never interacts with the thermometer directly, but instead through the temperature sensor task implemented by the device vendor. As such, the application will keep working even if the peripheral is changed.

```c
#include "../lib/inc/os_syscalls.h"

/* Temperature sensor is process 4. */
const int tempSensID = 4;

Message msg;

/* Use our own print function, since stdlib is unavailable.  */
void printInt(int x) {
    ...
}

int main() {
        while (1) {
            sysSend(tempSensID, &msg);
            sysReceive(tempSensID, &msg);
            sysPrint("Temperature: ");
            printInt(msg.mDatum.mInt);
            sysSleep(5000);
        }
        return 0;
}
```

4.2.4 Led

Led registers a callback that toggles LED 1 for when button 1 is pressed and then exits. It interfaces with the high level GPIO driver, as well as the high level LED driver. It demonstrates a more complex interaction between the drivers and system tasks that make up hackOS.

```
#include "../lib/inc/os_syscalls.h"

Message msg;

/* ID of GPIO and LED handler processes. */
const int gpioID = 2;
const int ledID  = 3;

void pbCallback(void) {
    /* Toggle LED 1. */
    msg.mType = 0;
    msg.mDatum.mInt = 1;
    sysSend(ledID, &msg);
}

int main(void) {
    /* Register button 1 callback. */
    msg.mType = 1;
    msg.mDatum.mPtr = &pb1_Callback;
    sysSend(gpioID, &msg);
    sysExit();
    return 0;
}
```

4.2.5 Stress Testing

We have conducted several tests with the applications above, and ensured the stability of the system throughout:

- Loaded each application into memory and ran it for four hours.
- Loaded each pair of applications into memory, and ran them concurrently for four hours.

- Loaded all three applications into memory, and ran them concurrently for four hours.

5 Conclusion

5.1 Summary

We have succeeded in achieving the requirements of this project. We provided hackOS, a pre-emptive, multi-tasking, real-time operating system for the EFM32HG MCU. It can run in normal or loader mode and supports up to sixteen processes across three priority levels. It contains various drivers for the EFM32HG starter kit and a platform to develop applications on. It provides several services to processes which include debug output, sleep, and registering as a handler for a specific interrupt. It implements a message passing system for inter-process communication, and enhances application portability by providing separation between the application code and the hardware.

5.2 Reflection

I got a lot of enjoyment out of this project. As my first incursion into the fields of embedded programming and operating system design, I had to do a lot of background reading. I progressed slowly at the beginning, but picked up speed along the way and am satisfied with how the project turned out.

I particularly enjoyed the low-level coding and interacting with hardware. Figuring out the delicate balance between the Cortex-M0+ and interrupt handlers in context switching was quite challenging, but also equally rewarding.

I particularly disliked hunting the bugs that kept showing up during the middle parts of development. They were hard to track down as the operating system wasn't yet integrated, and I also often lacked insight about which pieces could go wrong. I would often spend several days fixing one bug, only to run into another immediately after. Most of the bugs turned out to have a very simple cause - which I think is a great lesson for future troubles!

I am slightly disappointed that I couldn't finish making the application code relocatable, but was glad to learn about the ways this could be done nonetheless.

On the whole, this project gave me a completely new perspective on how computers work, and the complex interplay between applications and the operating system. It also sharpened my C skills quite a bit, and definitely laid the groundwork for many more skills that I'm sure will be important in my career.

5.3 Future Work

One possible direction for future work is making application code position independent. This would allow applications to be compiled and shared without regard for the exact place where they must end up in memory on each board. Applications achieve position independence mainly through the use of global offset tables, which are loaded into memory at a fixed offset from the text and provide a layer of indirection when accessing global variables.

This could be achieved by moving the global offset table to the beginning of the binary, then shifting the addresses inside at load time by the offset at which the application is placed in memory. This would probably work, although it must be noted that the global offset table is more complex in reality and a good understanding of linkers and executable formats is essential for this task.

Alternatively, the duty of shifting its global offset table could be left to the application itself during startup, which has the advantage of requiring no changes to the operating system. However, the application would have to modify the same flash page on which it is stored, which could introduce problems.

In theory, the operating system should be very easy to port to the Tiny Gecko and Zero Gecko microcontrollers, since they both use the Cortex-M0+ microprocessor. Future work could include porting the operating system to the whole array of Gecko MCUs, though this would undoubtedly be difficult as the specifics of the CPU would be different.

6 Appendix

We provide a Python script to interface with the hackOS loader through an USB connection. To send a binary, specify its name, the page on which to load it, and the desired RAM and stack space as arguments.

```python
#!/usr/bin/python3

import serial
import sys

# Output is little endian.
def toBytes(x):
    if x == 0:
        return bytearray([0])
    bytesList = []
    while x > 0:
        bytesList.append(x % 256)
        x //= 256
    return bytearray(bytesList)

# Marks the start of a program.
MAGIC_NR = int(0x01234567)
```

```
# Configure for your own system.
serialPort = serial.Serial('/dev/cu.usbmodem14101')

def serialOut(message):
    serialPort.write(message)
    # Flush the connection.
    serialPort.close()
    serialPort.open()

#   Layout of a program in flash:
#       0-3  : MAGIC NR
#       4-7  : ENTRY
#       8-23 : NAME
#       24-25: RAM SPACE
#       26-27: STK SPACE
#       28-31: RESERVED

#  ------------------------------

#   Layout of a header:
#       0-3  : MAGIC NR
#       4-7  : ENTRY
#       8-23 : NAME
#       24-25: RAM SPACE
#       26-27: STK SPACE
#       28-28: PAGE ID
#       29-31: RESERVED

# Binaries must be preceded by a header.
def buildHeader(name, entry, ramSpace, stkSpace, pageId):
    print(toBytes(pageId))
    return (
        toBytes(MAGIC_NR) +
        entry +
        bytearray(name.ljust(16), 'ascii') +
        toBytes(ramSpace) +
        toBytes(stkSpace) +
        toBytes(pageId)
    )

def sendBinary(name, ramSpace, stkSpace, pageId):
    path = name + '/' + name + '.bin'
    with open(path, 'rb') as f:
        contents  = f.read()
        length    = len(contents)
        blockSize = 32

        # Tell the loader to expect a binary.
        entry  = contents[0 : 4] # Entry point.
        header = buildHeader(name, entry, ramSpace, stkSpace, pageId)
        print(entry)
        print(header)
        serialOut(header)

        # Send binary.
        cntBlocks = length // blockSize
        if length % blockSize != 0:
            cntBlocks += 1
        for i in range(cntBlocks):
            left  = i * blockSize
            right = min(length, left + blockSize)
            message = contents[left : right]
            print(message.hex())
            serialOut(message)

        # Signal to the loader that we are done.
        serialOut(toBytes(MAGIC_NR))

name     = sys.argv[1]
pageId   = int(sys.argv[2])
ramSpace = int(sys.argv[3])
stkSpace = int(sys.argv[4])
sendBinary(name, ramSpace, stkSpace, pageId)
```

7 Bibliography

References

[1] Operating Systems: Design and Implementation, A. Tanenbaum
(ISBN 0131429388)

[2] Assemblers and Loaders, D. Salomon
(ISBN 0130525642)

[3] The Definitive Guide to ARM® Cortex®-M0 and Cortex-M0+ Processors, J. Yiu (ISBN 0128032774)

[4] ARM Cortex-M0+ Technical Reference Manual
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0484c/index.html`

[5] The Digital Systems Practicals,
`http://spivey.oriel.ox.ac.uk/hg/digisys/file/tip`

[6] EFM32HG SK User Manual,
`https://www.silabs.com/documents/public/user-guides/ug255-stk3400-user-guide.pdf`

[7] EFM32HG Reference Manual,
`https://www.silabs.com/documents/public/reference-manuals/efm32hg-rm.pdf`

[8] Exception Handler,
`https://www.sciencedirect.com/topics/engineering/exception-handler`

[9] Exception Return Mechanism,
`sciencedirect.com/topics/engineering/exception-return-mechanism`

[10] Handler Mode,
`https://www.sciencedirect.com/topics/engineering/handler-mode`

[11] Interrupted Program,
`https://www.sciencedirect.com/topics/engineering/interrupted-program`

[12] Thread Mode,
`https://www.sciencedirect.com/topics/engineering/thread-mode`

[13] Demystifying Microcontroller GPIO Settings,
`https://embeddedartistry.com/blog/2018/06/04/demystifying-microcontroller-gpio-settings/`

[14] Using LD, the GNU Linker,
`http://www.scoberlin.de/content/media/http/informatik/gcc_docs/ld_3.html`