Microkernel development: from project to implementation

Technical Notes

Rodrigo Maximiano Antunes de Almeida
rmaalmeida@gmail.com

Universidade Federal de Itajubá

Summary

This is the technical notes about the ESC talk: “Microkernel development: from project to implementation” given by Rodrigo Maximiano Antunes de Almeida, from Unifei.

The talk consists of the design and implementation of a microkernel. The project will be developed on ISO-C without the standard libraries, in such a way that the code to be presented to the participants can be easily ported to any architecture and used as a basis for future projects. It will also be addressed the standardization of the procedures and hardware requisitions, encapsulated as drivers. Upon completion, the participants will have a better understanding about the kernel its advantages and restrictions.

This documents presents in deeper details some of the topics that will be covered in the speech, mainly the theoretical ones.

Index

1 Developing an embedded system

An embedded system is a system designed for a specific propose that has a microcontroller or a microprocessor as the central piece. The microcontroller has a software built exclusively to support the functionalities underlined in the embedded product. In order to build an embedded system, we must account for both hardware and software issues.

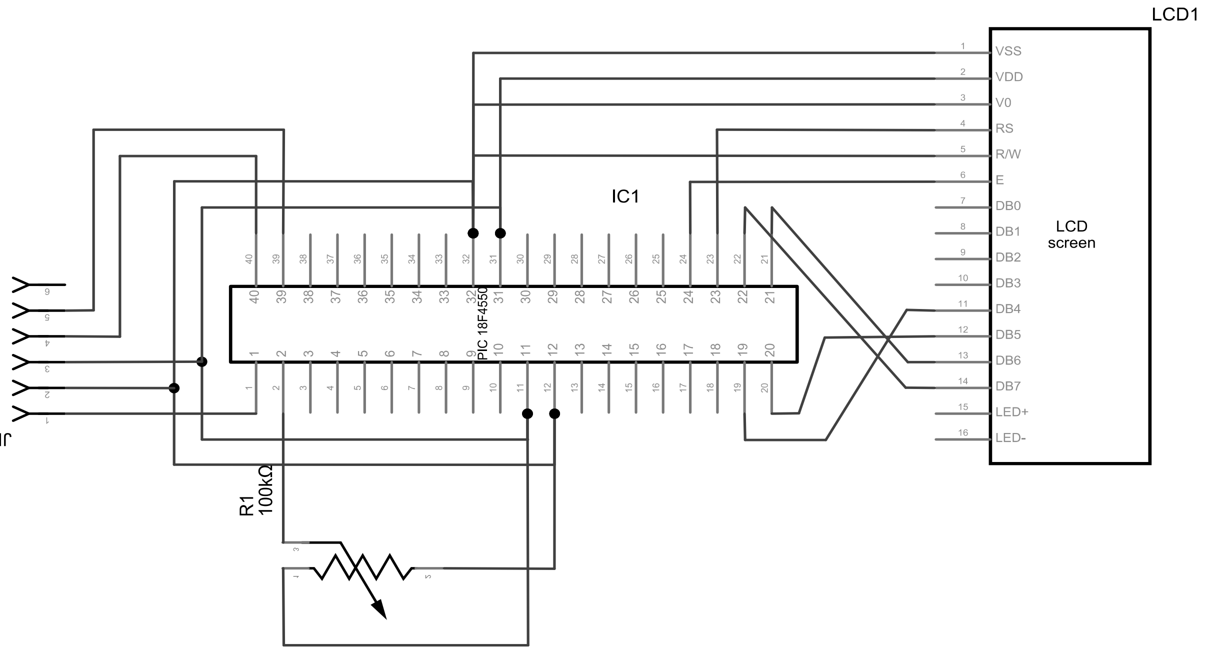
The hardware part is composed by a microcontroller and other circuits that are responsible to make the interface between the microcontroller and the user or other microcontroller. The document that describes the way these components are connected is called a schematic.

In this document we’re going to build a simple board with a microcontroller with a lcd. The way each of them must be connected depends on each component. In general their manufactures send an application note showing the default connection.

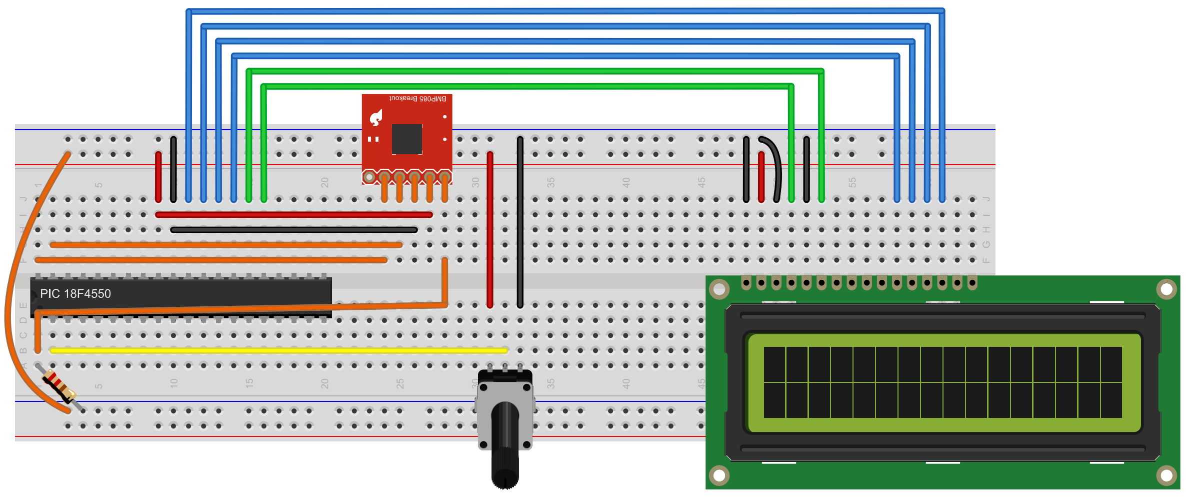
For the system we’re going to use a pic18f4550 as the microcontroller unit, a HD77480 as LCD output interface and a regular potentiometer as input information.

1.1 System connections

For the required system the connections will be made as in the following schematic



This system can be connected using a protoboard as the image below.


PIC 18F4550
BMP085 Breakout

2 System programming

In order to correctly program the system, a dedicated programmer will be used. This programmer need some initial code to execute correctly. These codes are responsible to make the initial setup on the system.

```
#pragma config MCLRE=ON             // Master Clear disable
#pragma config FOSC=INTOSC_XT       // Internal oscilator
#pragma config WDT=OFF              // Watchdog disbled
#pragma config LVP=OFF              // Low voltage programming disabled
#pragma config DEBUG=OFF
#pragma config XINST=OFF            // Don't use new processor instructions
```

2.1 *Making access to the hardware*

All terminals are mapped in the RAM area. In order to access one terminal we need to first find the terminal address and than make a pointer to this address. The PORTD for example is connected in the 0xF83 address.

```
void main (void){
    char *ptr;
    //pointing to the port D
    ptr = 0xF83;
    //changing all outputs to high
    *ptr = 0xFF;
}
```

Another need to make the peripheral usable, the circuit direction must be defined as input or output. This is achieved by the TRIS register. The code below can blink all the leds connected on the PORTD. In order to simplify the port access a #define structure can be used to hide the pointer accesses.

```
#define PORTD    (*(volatile __near unsigned char*)0xF83)
#define TRISD    (*(volatile __near unsigned char*)0xF95)
void main(void) {
        TRISD = 0x00;
        for(;;){
            PORTD ^= 0xFF;
        }
}
```

2.2 *Accessing individual bits*

The address used is generally composed of eight bits, each one mapped to one terminal. In order to make individual accessing we need to use some binary manipulation, using bitwise operations. These operations can be made in a pseudo-function using the #define preprocessor

```
//using define
#define BitSet(arg,bit) ((arg) |=  (1<<bit))
#define BitClr(arg,bit) ((arg) &= ~(1<<bit))
#define BitFlp(arg,bit) ((arg) ^=  (1<<bit))
#define BitTst(arg,bit) ((arg) &   (1<<bit))
```

2.3 LCD communication

The LCD communication will be made using only 4 bits of data. This enforces us to build a library witch can run all the steps required by the LCD. The LCD data will be accessed using the PORTD bits 4, 5, 6 and 7. The enable and register select pin will be accessed by the PORTC 6 and 7.

The LCD protocol requires some delays. As the delay time does not need to be precise we can use a simple for loop to achieve the times.

```
void delayMicroseconds(int ms) {
    int i;
    for (; ms > 0; ms--) {
        for (i = 0; i < 30; i++);
    }
}
```

Another step in the communication is the data clock. This is achieved using the enable pin.

```
void pulseEnablePin() {
    BitClr(PORTC, EN);
    delayMicroseconds(1);
    // send a pulse to enable
    BitSet(PORTC, EN);
    delayMicroseconds(1);
    BitClr(PORTC, EN);
}
```

To send a byte, it is needed to send first the most significant 4 bits and than the last 4 significant bits.

```
void pushNibble(int value, int rs) {
    PORTD = (value) << 4;
    if (rs) {
        BitSet(PORTC, RS);
    } else {
        BitClr(PORTC, RS);
    }
    pulseEnablePin();
}

void pushByte(int value, int rs) {
    int val_lower = value & 0x0F;
    int val_upper = value >> 4;
    pushNibble(val_upper, rs);
    pushNibble(val_lower, rs);
}
```

The difference between a command or a data is the value on register select pin.

```
void lcdCommand(int value) {
    pushByte(value, 0);
    delayMicroseconds(40);
}

void lcdChar(char value) {
    pushByte((unsigned int) value, 1);
    delayMicroseconds(2);
}
```

The initialization requires a strict protocol of delays and different operations.

```
#define PORTC   (*(volatile __near unsigned char*)0xF82)
#define PORTD   (*(volatile __near unsigned char*)0xF83)
#define TRISC   (*(volatile __near unsigned char*)0xF94)
#define TRISD   (*(volatile __near unsigned char*)0xF95)

void lcdInit() {
    BitClr(TRISC, EN);
    BitClr(TRISC, RS);
    TRISD = 0x0f;
    delayMicroseconds(50);
    commandWriteNibble(0x03);
    delayMicroseconds(5);
    commandWriteNibble(0x03);
    delayMicroseconds(100);
    commandWriteNibble(0x03);
    delayMicroseconds(5);
    commandWriteNibble(0x02);
    delayMicroseconds(10);
    //configura o display
    lcdCommand(0x28); //8bits, 2 linhas, 5x8
    lcdCommand(0x06); //modo incremental
    lcdCommand(0x0c); //display e cursor on, com blink
    lcdCommand(0x03); //zera tudo
    lcdCommand(0x80); //posição inicial
    lcdCommand(0x01); //limpar display
    delayMicroseconds(2);
}
```

To print a regular string one must just watch for the string end '\0'.

```
void lcdString(char msg[]) {
    unsigned char i = 0; //fancy int.  avoids compiler warning when comparing i
with strlen()'s uint8_t
    while (msg[i]) {
        lcdChar(msg[i]);
        i++;
    }
}
```

The LCD can build customized characters. The characters must be send to a specific address to the LCD internal memory.

```c
void lcdDefconLogo(void) {
    int i;
    unsigned char defcon[] = {
        0x0, 0x1, 0x3, 0x3, 0x3, 0x3, 0x1, 0x4,
        0xe, 0x1f, 0x4, 0x4, 0x1f, 0xe, 0x11, 0x1f,
        0x0, 0x10, 0x18, 0x18, 0x18, 0x18, 0x10, 0x4,
        0xc, 0x3, 0x0, 0x0, 0x0, 0x3, 0xc, 0x4,
        0x0, 0x0, 0x1b, 0x4, 0x1b, 0x0, 0x0, 0x0,
        0x6, 0x18, 0x0, 0x0, 0x0, 0x18, 0x6, 0x2
    };
    lcdCommand(0x40);
    for (i = 0; i < 8 * 6; i++) {
        lcdChar(defcon[i]);
    }
}
```

2.4 *Analog reading*

The reading on the analog pins require an initial setup from the AD converter with a latter loop to wait for conversion time.

```c
#define TRISA   (*(volatile __near unsigned char*)0xF92)
#define ADCON2  (*(volatile __near unsigned char*)0xFC0)
#define ADCON1  (*(volatile __near unsigned char*)0xFC1)
#define ADCON0  (*(volatile __near unsigned char*)0xFC2)
#define ADRESL  (*(volatile __near unsigned char*)0xFC3)
#define ADRESH  (*(volatile __near unsigned char*)0xFC4)

void adInit(void) {
    BitSet(TRISA, 0); //pin setup
    ADCON0 = 0b00000001; //channel select
    ADCON1 = 0b00001110; //ref = source
    ADCON2 = 0b10101010; //t_conv = 12 TAD
}

unsigned int adRead(void) {
    unsigned int ADvalue;
    BitSet(ADCON0, 1); //start conversion
    while (BitTst(ADCON0, 1)); //wait
    ADvalue = ADRESH; //read result
    ADvalue <<= 8;
    ADvalue += ADRESL;
    return ADvalue;
}
```

To use the AD it is needed only to initialize it before using. In the example below we're going to light the led whenever the value goes above a threshold.

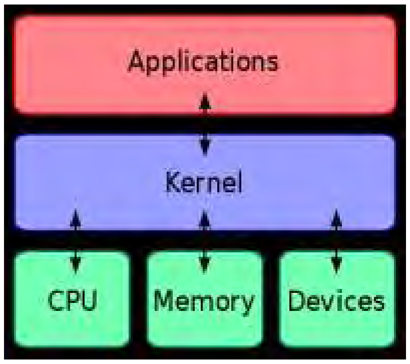
```
#define PORTD    (*(volatile __near unsigned char*)0xF83)
#define TRISD    (*(volatile __near unsigned char*)0xF95)
void main(void) {
    TRISD = 0x00;
    adInit();
    for(;;){
       //threshold on half the scale
       if(adRead()>512){
            PORTD = 0xFF;
       }else{
           PORTD = 0x00;
       }
    }
}
```

3 First embedded firmware

```
void main(void) {
    OSCCON = 0x73;
    lcdInit();
    lcdDefconLogo();
    lcdCommand(0x80);
    lcdChar(0);
    lcdChar(1);
    lcdChar(2);
    lcdString(" Defcon");
    lcdCommand(0xC0);
    lcdChar(3);
    lcdChar(4);
    lcdChar(5);
    lcdString("mBed workshop");
    adInit();

    for (;;) {
        lcdCommand(0x8B);
        lcdChar((adRead() / 1000) % 10 + 48);
        lcdChar((adRead() / 100) % 10 + 48);
        lcdChar((adRead() / 10) % 10 + 48);
        lcdChar((adRead() / 1) % 10 + 48);
    }
}
```

4 What is a kernel?

In computer science the kernel is the software part of the system responsible to implement the interface and manage the hardware and the application. The most critical hardware resources to be managed are the processor, the memory and I/O drivers.



Another task that is commonly provided by the kernel is the process management. This is even more important in embedded context, when, in general, the processes have strict time requirements.

When there is no kernel, all the responsibility of organizing the processes, hardware or application processes, is over the programmer.

5 Kernel components

In general a kernel has three main responsibilities:

1) Manage and coordinate the processes execution using “some criteria”

The “some criterea” can be the maximum execution time, the function's priorities, the event criticity, the programmed execution sequence among others. It is this criteria which distinguishes the preemptive kernel (which each process has a maximum time to be executed, if the time runs out the next process is started and when it finishes, or its time expire, the first process is recalled from the exact point were it was interrupt) from the cooperative (which each process will be executed until it ends and after it the next is called). As it is responsible to manage the process, it should have functions that allow the inclusion of a new process or the remove of an older one.

As each process uses, internally, some amount of memory for its variables, the kernel should handle it to. It is the second kernel responsibility.

2) Manage the free memory and coordinate the processes access to it

The kernel should also be capable to inform to the process when a malloc() function could not be accomplished.

Aside the memory, the processes may also need to get access to the I/O resources from the computer/microcontroler as serial ports, lcd displays, keyboards among others. The responsible to allow/deny the access from the processes to the hardware devices is the kernel. This is the third kernel responsibility:

3) Intermediate the communication between the hardware drivers and the processes

The kernel should provide an API which the processes can safely access the information available in the hardware, both to read and to write.

6 Kernel Project

6.1 *Why to develop our own kernel?*

Having your own kernel can improve home design while still giving the developers full control over the source.

With the single-loop architecture, you need to re-test almost everything every time you reuse code. When the kernel is full tested, there is no problem in reuse. Even the applications have a better reuse rate as the kernel keeps the hardware abstraction layer even if the chip is changed.

When planning to use a kernel in your new system development, always consider all alternatives, both paid and free.

Even if the home design option was choosen start with a free project as basis. Both OpenRTOS and BRTOS are really small (BRTOS has only 7 code files, and only one is hardware dependent) and their licenses are more permissive (you can close the source code). Another great source information is the linux kernel (www.kernel.org). There are as much as 10k lines added daily!

6.2 *Alternatives*

There are lots of options for moving from a kernel-less to a kernel design. Paid solutions have some benefits, mainly for the technical support. Below are presented some options with their descriptions.

Windows Embedded Compact® is the Windows version for small computers and embedded systems. It is a modular real-time operating system with a kernel that can run in under 1 MB of memory. It is available for ARM, MIPS, SuperH and x86 processor architectures. The source code is available to modifications for the developer.

VxWorks® is a real-time operating system. It has been ported and optmize to embedded systems, including x86 family, MIPS, PowerPC, Freescale ColdFire, Intel i960, SPARC, SH-4 and ARM. In its smallest option (fully static) have a footprint of just 36k.

X RTOS®: this kernel is mainly aimed in *Deeply Embedded Systems* with severe temporal restriction and computer resources. It supports ARM and Power PC processors.

uClinux is a derivative of Linux kernel intended for microcontrollers without Memory Management Units (MMUs). As an operating system it includes Linux kernel as well as a collection of user applications, libraries and tool chains. The kernel can be compiled to a footprint of just 2k.

FreeRTOS kernel consists of only three or four C files (there are a few assembler functions included where needed). SafeRTOS is based on the FreeRTOS code base but has been updated, documented, tested and audited to enable its use in IEC 61508 safety related applications.

BRTOS is a lightweight preemptive real time operating system designed for low end microcontrollers. It supports a preemptive scheduler, semaphores, mutexes, message queues and mailboxes. It is written mostly in C language, with little assembly code. There are ports to Coldfire V1, HCS08, RX600, MSP430, ATMEGA328/128 and Microchip PIC18. It can be compiled to just 2KB of program memory and about 100bytes of RAM.

6.3 *Monolithic kernel versus microkernel*

The main difference between these architectures are the amount of functions that are implemented inside the kernel space. By keeping a minimalistic approach, microkernels tend to use less resources from the cpu. Becouse the device drivers are now in the user space, microkernels tend to be less susceptible from drivers crash. It is also easier to maintain an microkernel because its small source code size, generally under 10.000

lines. As Jochen Liedtke stated:

"A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality."

6.4 Kernel design decisions:

The designer should consider some points before star the development of the kernel:

I/O devices management: How should the kernel implement the device interface? Inside the kernel? Using devices drivers? Will it use a separated driver controller or it will be implicit in kernel activities? The direct access (application<>driver) will be possible? In which case? In case of hot-plug devices how the kernel will load the drive dynamically?

Process management: The kernel context switch will be cooperative or preemptive? How the process can communicate with each other? Will a message queue be implemented? Should it have an shared memory? How to control its access? Will semaphores be available? Is there any need to implement process priority check?

System safety: Is there any hardware safety item to be used (watchdog, protected memory)? Will hierarchical protection be used? If so, does the CPU support MMU or the design can go on with the slowdown of software protection checking? The system should try to close and restart an unanswered process automatically?

Decide carefully, some of these decisions cannot be changed without a complete source code rewrite, other ones can be delayed until latter in the project. Bring the hardware responsible to help in this definition, some of the decisions are very hardware dependent.

6.5 This course decisions

In this course we will present a simple non-preemptive, cooperative microkernel, without memory management using an device driver controller to isolate the devices drivers from the kernel. The processes will be scheduled based on their execution frequency necessities.

7 Concepts

Kernel development require some deep knowledge in programming and hardware/software issues. Some of these points will be presented ahead.

7.1 Function pointers

In some situation we want our program to choose which function to execute, for example an image editor: use the function blur or the function sharpen at some image. Declaring both function:

```
image Blur(image nImg){
 // Function implementation
}
image Sharpen(image nImg){
 // Function implementation
}
```

We can build the image editor engine as:

```
image imageEditorEngine(image nImg, int option){
 image temp;
 switch(option){
     case 1:
         temp = Sharpen(nImg);
 break;
 case 2:
     temp = Blur(nImg);
 break;
 }
 return temp;
}
```

Its clear that we need to change the engine code if we need to add some more features. In general, changing the code means more tests and more bugs.

Another option is to made the engine a little bit more generic by using function pointers.

```
//declaracao do tipo ponteiro para função
typedef image (*ptrFunc)(image nImg);

//chamado pelo editor de imagens
image imageEditorEngine(ptrFunc function, image nImg){
 image temp;
 temp = (*function)(nImg);
 return temp;
}
```

From the code we could note that the function receives now an pointer to function as parameter. This way we do not need to worry about adding features, the main code will be kept intact. One of the drawbacks is that all functions now must have the same signature, it means, they must receive the same type parameters in the same order and the return variable must be of the same type.

Using function pointers concept we can than use the Blur and Sharpen functions in an easier way:

```
//...
 image nImage = getCameraImage();
 nImage = imageEditorEngine(Blur, nImagem);
 nImage = imageEditorEngine(Sharpen, nImagem);
//...
```

The functions are passed as if they were variables.

By essentially being an pointer, we must dereference the variable before using the function:

```
temp = (*function)(nImg);
```

We can also store the function passed as parameter as a conventional variable. This way we can call that function latter in the program (only the pointer is stored, no code is actually copied).

The syntax function pointer declaration is somehow complex. Normally we use a typedef to make things clear.

7.2 First example

In this first example we will build the main part of our kernel. It should have a way to store which functions are needed to be executed and in which order. To accomplish this we will use an vector of function pointers:

```
//pointer function declaration
typedef void(*ptrFunc)(void);
//process pool
static ptrFunc pool[4];
```

Our processes will be of ptrFunc type, i.e. they do not receive any parameters and do not return anything.

Each process will be represented by a function. Here are three examples

```
static void tst1(void) { printf("Process 1\n");}
static void tst2(void) { printf("Process 2\n");}
static void tst3(void) { printf("Process 3\n");}
```

These processes just print their name on the console/default output.

The kernel itself has three functions, one to initialize itself, one to add new process to the process pool, and one to start the execution. As it is supposed that the kernel never wears off, we build an infinite loop inside its execution function.

```
//kernel variables
static ptrFunc pool[4];
static int end;
//protótipos das funções do kernel
static void kernelInit(void);
static void kernelAddProc(ptrFunc newFunc);
static void kernelLoop(void);
//funções do kernel
static void kernelInit(void){
 end = 0;
}
static void kernelAddProc(ptrFunc newFunc){
 if (end <4){
     pool[end] = newFunc;
     end++;
 }
}
static void kernelLoop(void){
 int i;
 for(i=0; i<end;i++){
     (*pool[i])();
 }
}
```

In this first example the kernel only execute the functions that are given to it, in the order which they were called. There is no other control. The process pool size is defined statically.

To use the kernel we should follow three steps: initialize the kernel, add the desired processes, and execute the kernel.

```
void main(void){
 kernelInit();
 kernelAddProc(tst1);
 kernelAddProc(tst2);
 kernelAddProc(tst3);
 kernelLoop();
}
```

7.3 Structs

Structs are composed variables. With them we can group lots of information and work with them as if they were one single variable. They can be compared with vectors, but each position can store a different type variable. Here is an example:

```
typedef struct{
 unsigned short int age;
 char name[51];
 float weight;
}people; // struct declaration

void main(void){
 struct people myself = {26, "Rodrigo", 70.5};

 //using each variable from the struct
 printf("Age:    %d\n", myself.age);
 printf("Name:   %s\n", myself.name);
 printf("Weight: %f\n", myself.weight);

 return 0;
}
```

To build a functional kernel, we need to aggregate more information about each process. We will make it through a struct. For now just the function pointer is enough. As more information is needed (as process ID or priority) we will add to the process struct.

```
//function pointer declaration
typedef char(*ptrFunc)(void);

//process struct
typedef struct {
 ptrFunc function;
} process;
```

We should note that now every process must return a char. We will use it as returning condition indicating success or failure.

7.4 Circular buffers

Buffers are memory spaces with the propose of storing temporary data. Circular buffers can be implemented using a normal vector with two indexes, one indicating the list start and other indicating the list end.

The main problem with this implementation is to define when the vector is full or empty, as in both cases the start and the end index are pointing to the same place.

There are at least 4 alternatives on how to resolve this problem. In order to keep the system simplicity we will keep the last slot always open, in this case if (start==end) the list is empty.

Below there is an example on how to cycle through all the vector an infinite number of times:

```
#define CB_SIZE 10
int circular_buffer[CB_SIZE];
int index=0;
for(;;){
 //do anything with the buffer
 circular_buffer[index] = index;
 //increment the index
 index = (index+1)%CB_SIZE;
}
```

To add one element to the buffer (avoiding overflow) we can implement a function like this:

```
#define CB_SIZE 10
int circular_buffer[CB_SIZE];
int start=0;
int end =0;

char AddBuff(int newData){
 //check if there is space to add any number
 if ( ((end+1)%CB_SIZE) != start){
     circular_buffer[end] = newData;
     end = (end+1)%CB_SIZE;
     return SUCCESS;
 }
 return FAIL;
}
```

7.5 Second Example

As presented there is four important changes in this version: the process pool now is implemented as a circular buffer,the process is now represented as a struct, composed of the process id and the process function pointer and all functions return an error/success code.

The last change is that now the process can inform the kernel that it wants to be rescheduled. The kernel then re-adds the process to the process pool

```
//return code
#define SUCCESS   0
#define FAIL      1
#define REPEAT    2

//kernel information
#define POOL_SIZE 4
process pool[SLOT_SIZE];
char start;
char end;

//kernel functions
char kernelInit(void);
char kernelAddProc(process newProc);
void KernelLoop(void);
```

The biggest change in kernel usage is that now we need to pass an process struct to AddProc function instead of only the function pointer. Note that each process function returns if they are successfully finished or if it wants to be rescheduled.

```
char tst1(void){
 printf("Process 1\n");
 return REPEAT;
}

char tst2(void){
 printf("Process 2\n");
 return SUCCESS;
}

char tst3(void){
 printf("Process 3\n");
 return REPEAT;
}


void main(void){
 //declaring the processes
 process p1 = {tst1};
 process p2 = {tst2};
 process p3 = {tst3};
 kernelInit();

 //now is possible to test if the process was added successfully
 if (kernelAddProc(p1) == SUCCESS){
     printf("1st process added\n");
 }
 if (kernelAddProc(p2) == SUCCESS){
     printf("2nd process added\n");
 }
 if (kernelAddProc(p3) == SUCCESS){
     printf("3rd process added\n");
 }

 kernelLoop();
}
```

The kernel function Execute are the one with most changes. Now it needs to check if the executed function wants to be rescheduled and act as specified.

```
void kernelLoop(void){
 int i=0;
 for(;;){
     //Do we have any process to execute?
     if (start != end){
         printf("Ite. %d, Slot. %d: ", i, start);
         //execute the first function and
         //check if there is need to reschedule
         if ( (*(pool[start].Func))() == REPEAT){
         //rescheduling
         kernelAddProc(pool[start]);
         }
         //prepare to get the next process
         start = (start+1)%POOL_SIZE;
         //just for debug
         i++;
     }
 }
}
```

The AddProc() function have to check if there is at least two slots available in the buffer (remember that the last position is required to be free all times) and insert the process.

```
char kernelAddProc(process newProc){
 //checking for free space
 if ( ((end+1)%SLOT_SIZE) != start){
     pool[end] = newProc;
     end = (end+1)%POOL_SIZE;
     return SUCCESS;
 }
 return FAIL;
}
```

The initialization routine only set start and end variables to the first position

```
char kernelInit(void){
 start = 0;
 end = 0;
 return SUCCESS;
}
```

Here is presented the output of the main program for the first 10 iterations.

```
-----------------------------
1st process added
2nd process added
3rd process added
Ite. 0, Slot. 0: Process 1
Ite. 1, Slot. 1: Process 2
Ite. 2, Slot. 2: Process 3
Ite. 3, Slot. 3: Process 1
Ite. 4, Slot. 0: Process 3
Ite. 5, Slot. 1: Process 1
Ite. 6, Slot. 2: Process 3
Ite. 7, Slot. 3: Process 1
Ite. 8, Slot. 0: Process 3
Ite. 9, Slot. 1: Process 1
...
-----------------------------
```

Note that only process 1 and 3 are repeating, as expected. Note also that the pool is cycling through slots

0, 1, 2 and 3 naturally. For the user the process pool seems “infinite” as long as there is no more functions than slots.

7.6 Temporal conditions

In the majority part of embedded systems, we need to guarantee that a function will be executed in a certain frequency. Some systems may even fail if these deadlines are not met.

There are at least 3 conditions that need to be satisfied in order to implement temporal conditions on the kernel:

1. There must be a tick event that occurs with a precise frequency
2. The kernel must be informed of the execution frequency needed for each process.
3. The sum of process duration must “fit” within the processor available time.

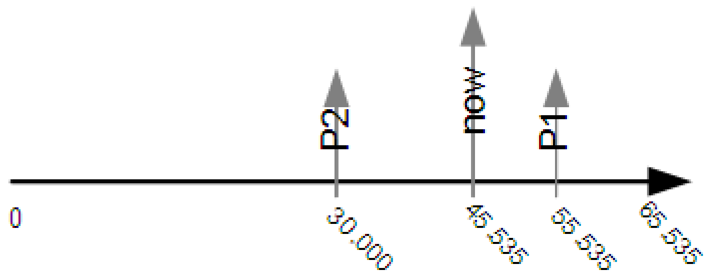
The first condition can be easily satisfied if there is an available internal timer that can generate an interrupt. This is true for the overwhelming majority of microcontrollers. There is no need for a dedicate interrupt routine.

For the second condition we just need to add the desired information in the process struct. We added two integers, the first indicate the period which the frequency should be recalled (if it returns REPEAT code). The second is an internal variable, in which the kernel store the remaining time before call function.

```
//process struct
typedef struct {
  ptrFunc function;
  int period;
  int start;
} process;
```

The third condition depends entirely on the system itself. Suppose a system which the function UpdateDisplays() need to be called in 5ms interval. If this function execution time is greater than 5ms it is impossible to guarantee its execution interval. Another point worth consideration is about the type of context switcher, if it is preemptive or cooperative. On a cooperative system, the process must finish its execution before another one can run on CPU. On a preemptive system, the kernel can stop one process execution anywhere to execute another process. If a system does not “fit” on the available time, there are three options: switch to a faster processor, optimize the execution time of the processes or to redesign the processes frequency needs.

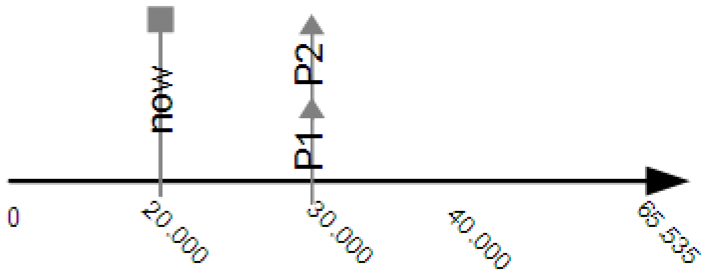
When implementing the time conditions a problem may arrise. Suppose two process P1 and P2. The first is scheduled to happen 10 seconds from now and the second at 50 seconds from now. The timer we are using is 16 bits unsigned (values from 0 to 65.535) counting in miliseconds and now it is marking 45,5 seconds (now_ms =45.535).



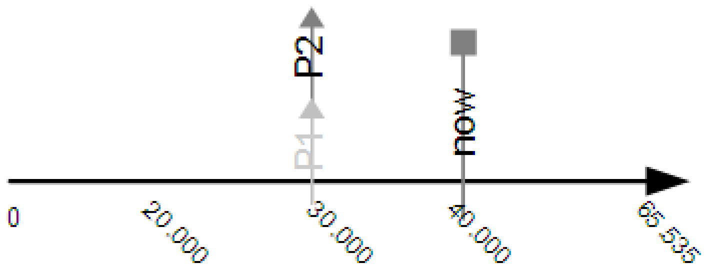
We can see from the picture that the process P2 was correctly scheduled as P2.start = now_ms + 50.000 = 30.000; The now_ms variable will be incremented until 55.535 when the process P1 will started (with the correct delay of 10 seconds). The variable now_ms will continue until 65.535 and then return to zero.

When the overflow happen, exactly 20 seconds has passed from the start (65.535 - 45.535 = 20.000ms). P2 required 50 seconds of delay. Its necessary to wait for more 30 seconds before it can be called, which is exactly what will happen when now_ms get to 30.000.

The problem to use an finite number to measure time may arise when two processes should be called withing a small space of time or even simultaneous.



Suppose that now P1 and P2 are scheduled to happen exactly at now_ms = 30.000. If P1 is called first and takes 10 seconds to be executed we will have the following time-line:



The question now is: From the time-line above (that is the only information that the kernel has), the process P2 should have already been executed and wasn't or it was scheduled to happen 50.535 ms from now?

In order to solve this problem there are two options:

1. Create a flag for each process indicating when it has already being passed by the timer counter. This way we can know if a process behind the counter is late or if it has been scheduled to happen ahead.
2. Create a counter that will be decrement at each kernel clock. It will be executed when it arrives at zero.

The second option introduces more overhead as we need to decrement all processes counters. On the other hand if we allow the counter to assume negative numbers (as it is decremented) we can see for how many time the process is waiting. With this information we can do something to avoid starvation. One option is to create a priority system and promote the process if it stays too much time without being executed.

7.7 Third Example

This time we're adding the time component to the kernel. Each process has a field called start which is decremented as time pass by. When it became zero (or negative) we call the function.

The ExecuteKernel() function will be responsible to find the process that is closer to be executed based on its start counter. It is necessary to go through the whole pool to make this search. When the next process to be executed is found we swap its position with the first one on the pool. After it we just spend time waiting for the process be ready to execute.

This apparent useless time is needed to synchronize all the events and is a good opportunity to put the system on the low power mode.

```
void ExecutaKernel(void){
 unsigned char j;
 unsigned char next;
 process tempProc;
 for(;;){
     if (start != end){
         //Findind the process with the smallest start
         j = (start+1)%SLOT_SIZE;
         next = start;
         while(j!=end){
         //does the next has a smaller time?
         if (pool[j].start < pool[next].start){
                next = j;
         }
         //get the next one in the circular buffer
         j = (j+1)%SLOT_SIZE;
         }
         //exchanging positions in the pool
         tempProc = pool[next];
         pool[next] = pool[start];
         pool[start] = tempProc.;

         while(pool[start].start>0){
         //great place to use low power mode
         }

         //checking if need to be repeated
         if ( (*(pool[ini].function))() == REPEAT ){
         AddProc(&(vetProc[ini]));
         }
         //next process
         ini = (ini+1)%SLOT_SIZE;
     }
 }
 }
```

Now the interrupt routine. It must decrement the start field of all of the processes.

```
void interrupt_service_routine(void) interrupt 1{
unsigned char i;
 i = ini;
 while(i!=fim){
     if((pool[i].start)>(MIN_INT)){
         pool[i].start--;
     }
     i = (i+1)%SLOT_SIZE;
 }
}
```

The AddProc() function will be the responsible to initialize the process with an adequate value on the struct fields.

```
char AddProc(process newProc){
 //checking for free space
 if ( ((end+1)%SLOT_SIZE) != start){
     pool[end] = newProc;
     //increment start timer with period
     pool[end].start += newProc.period;
     end = (end+1)%SLOT_SIZE;
     return SUCCESS;
 }
 return FAIL;
}
```

Instead of resetting the start counter we add the period to it. This was done because when the function starts, its counter keeps decrementing. If a function needs to be executed at a 5ms interval and it spends 1ms executing, when it finishes we want to reschedule it to execute 4ms ahead (5ms of period + -1ms negative start counter) and not 5ms ahead.

`Void pointers`

When designing the device driver controller we should build an "call distribution center". This center will receive an order from the application, via kernel, and redirect it to right device driver. The problem arises when we think on how many parameters the function should receive: one representing which driver is required, another one indicates which function of that driver should be called and an unknown amount of parameters that need to be passed to the driver. How to build such function?

It can be done with a pointer to void.

```
char * name = "Paulo";
double weight = 87.5;
unsigned int children = 3;

void print(int option; void *parameter){
 switch(option){
     case 0:
         printf("%s",*((char*)parameter));
     break;
     case 1:
         printf("%f",*((double*)parameter));
     break;
     case 2:
         printf("%d",*((unsigned int*)parameter));
     break;
 }
}

void main (void){
 print(0, &name);
 print(1, &weight);
 print(2, &children);
}
```

From the above example we can see how to receive different types using the same function.

8 The Kernel

This is the full kernel presented in the earlier steps. In order to make it run on the development board we are counting on auxiliary libraries: one to work with interrupts (int.c and int.h), one to operate the timer (timer.c and timer.h), one to configure the microcontroler fuses (config.h) and another one with the special registers information (basico.h).

```
//CONFIG.H
//microcontroler fuses configuration
code char at 0x300000 CONFIG1L = 0x01;  // No prescaler used
code char at 0x300001 CONFIG1H = 0x0C;  // HS: High Speed Cristal
code char at 0x300003 CONFIG2H = 0x00;  // Disabled-Controlled by SWDTEN bit
code char at 0x300006 CONFIG4L = 0x00;  // Disabled low voltage programming

//INT.H
void InicializaInterrupt(void);

//TIMER.H
char FimTimer(void);
void AguardaTimer(void);
void ResetaTimer(unsigned int tempo);
void InicializaTimer(void);

//BASICO.H (only part of it)
#define SUCCESS  0
#define FAIL     1
#define REPEAT   2
//bit functions
#define BitFlp(arg,bit) ((arg) ^= (1<<bit))
//special register information
#define PORTD   (*(volatile __near unsigned char*)0xF83)
#define TRISC   (*(volatile __near unsigned char*)0xF94)
```

In order to work with time requirements we need to make some operations in fixed intervals of time, mainly decrement the process start counter. These steps were grouped together in one function: KernelClock(). The user just need to call this function from its own timer interrupt function.

```
void kernelClock(void){
 unsigned char i;
 i = ini;
 while(i!=fim){
     if((pool[i].start)>(MIN_INT)){
         pool[i].start--;
     }
     i = (i+1)%SLOT_SIZE;
 }
}
```

The other kernel function stays the same as presented.

```
char kernelAddProc(process newProc){
 //checking for free space
 if ( ((end+1)%SLOT_SIZE) != start){
     pool[end] = newProc;
     //increment start timer with period
     pool[end].start += newProc.period;
     end = (end+1)%SLOT_SIZE;
     return SUCCESS;
 }
 return FAIL;
}

char kernelInit(void){
 start = 0;
 end = 0;
 return SUCCESS;
}
void kernelLoop(void){
 unsigned char j;
 unsigned char next;
 process tempProc;
 for(;;){
     if (start != end){
         //Findind the process with the smallest start
         j = (start+1)%SLOT_SIZE;
         next = start;
         while(j!=end){
         //does the next has a smaller time?
         if (pool[j].start < pool[next].start){
                next = j;
         }
         //get the next one in the circular buffer
         j = (j+1)%SLOT_SIZE;
         }
         //exchanging positions in the pool
         tempProc = pool[next];
         pool[next] = pool[start];
         pool[start] = tempProc.;

         while(pool[start].start>0){
         //great place to use low power mode
         }

         //checking if need to be repeated
         if ( (*(pool[ini].function))() == REPEAT ){
         AddProc(&(vetProc[ini]));
         }
         //next process
         ini = (ini+1)%SLOT_SIZE;
     }
 }
}
```

To declare the interrupt function for SDCC compiler we just need to add ”interrupt 1” at the end of the function name. As mentioned it just reset the timer and calls the KernelClock.

```
//Interrupt
void isr1(void) interrupt 1{
 ResetaTimer(1000); //reset with 1ms
 KernelClock();
}
```

In order to use the kernel we just need to call its initialization function, add the processes with their frequency of execution and call the ExecutaKernel() function.

```
//Blink led 1
char tst1(void) {
 BitFlp(PORTD,0);
 return REPETIR;
}

//Blink led 2
char tst2(void) {
 BitFlp(PORTD,1);
 return REPETIR;
}

//Blink led 3
char tst3(void) {
 BitFlp(PORTD,2);
 return REPETIR;
}

void main(void){
 //declaring the processes
 process p1 = {tst1,0,100};
 process p2 = {tst2,0,1000};
 process p3 = {tst3,0,10000};

 kernelInit();

 kernelAddProc(p1);
 kernelAddProc(p2);
 kernelAddProc(p3);

 kernelLoop();
}
```

9 Building the device driver controller

In order to isolate the drivers from the kernel (consequently from the applications) we will build an device driver controller. It will be responsible to load the driver and pass the orders received from the kernel to the right driver.

9.1 Device Driver Pattern

All kernels presents some kind of pattern to build your driver driver. This standardization is fundamental to the system. Only by having an standard interface, the kernel could communicate with a driver that he knows nothing at compile time.

In order to simplify the pointers usage, we've build several typedefs. ptrFuncDriver is a pointer to a function that returns a char (error/success code) and receives a pointer to void as parameter. It is used to call each driver's function as all of them must have this signature.

```
typedef char(*ptrFuncDrv)(void *parameters);
```

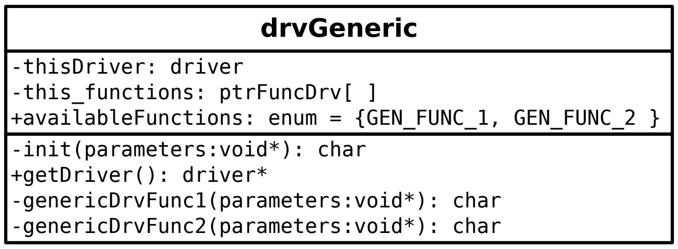
The driver struct is composed of the driver id, an array of ptrFuncDrv pointers (represented as a pointer) and a special function also of the type ptrFuncDrv. This last function is responsible to initialize the driver once it is loaded.

```
typedef struct {
 char drv_id;
 ptrFuncDrv *functions;
 ptrFuncDrv drv_init;
} driver;
```

In order to the device driver controller to access the device drivers, it need to get a pointer to a “driver structure”. Instead of make a statical linking, we set the links with a pointer to a function that, when called, returns the desired device driver.

```
typedef driver* (*ptrGetDrv)(void);
```

A generic driver needs then to implement at least 2 functions: init() and getDriver(). It also needs to have a driver struct and a array of pointers, each position pointing to each function it implements. It also needed to build an enumerator defining the positions of each function pointer in the array.



```
-thisDriver: driver
-this_functions: ptrFuncDrv[ ]
+availableFunctions: enum = {GEN_FUNC_1, GEN_FUNC_2 }
-init(parameters:void*): char
+getDriver(): driver*
-genericDrvFunc1(parameters:void*): char
-genericDrvFunc2(parameters:void*): char
```

9.2 Controller engine

The device driver controller will need at least to known all available drivers. It is done by a vector of ptrGetDriver, in which each position holds the pointer to the driver function that return its driver struct. The position which the pointers are stored in the vector is defined by an enumerator, helping identify which pointer is for which driver.

```
//it is needed to include all drivers file
#include "drvInterrupt.h"
#include "drvTimer.h"
#include "drvLcd.h"

//this enumerator helps the developer to access the drivers
enum {
 DRV_INTERRUPT,
 DRV_TIMER,
 DRV_LCD,
 DRV_END /*DRV_END should always be the last*/
};

//the functions to get the drivers should
//be put in the same order as in the enum
static ptrGetDrv drvInitVect[DRV_END] = {
 getInterruptDriver,
 getTimerDriver,
 getLCDDriver
};
```

The device driver controller has one array of drivers and a counter indicating how many drivers are loaded at the moment. There are only 3 functions: one to initialize the internal variables, one to load a driver and one to that parse the commands from the kernel to the correct driver.

Loading a driver is pretty straightforward. If the driver DRV_INTERRUPT needs to be loaded, we go to the available drivers list and ask for the interrupt driver. Then we call its initialization routine and store it on the loaded list. If there is no space for another driver the function returns an error

```
char initDriver(char newDriver) {
 char resp = FIM_FALHA;
 if(driversLoaded < QNTD_DRV) {
     drivers[driversLoaded] = drvInitVect[newDriver]();
     resp = drivers[driversLoaded]->drv_init(&newDriver);
     driversLoaded++;
 }
 return resp;
}
```

The call driver routine go through the loaded drivers list to identify the correct driver. When there is a match the correct function is called and the parameters passed as a pointer to void (in this moment we do not know what are the parameters).

```
char callDriver(char drv_id, char func_id, void *parameters) {
 char i, j;
 for (i = 0; i < driversLoaded; i++) {
     if (drv_id == drivers[i]->drv_id) {
         return drivers[i]->func_ptr[func_id].func_ptr(parameters);
     }
 }
 return DRV_FUNC_NOT_FOUND;
}
```

9.3 *Using the controller engine*

In order to use the controller engine we just need to include its header on the main file and make use of the enumerators defined in each driver file to access the hardware.

```
void main(void) {
 //system initialization
 //the kernel also start the controller init function.
 InicializaKernel();
 initDriver(DRV_LCD);
 callDriver(DRV_LCD, LCD_CARACTER, 'U');
 callDriver(DRV_LCD, LCD_CARACTER, 'n');
 callDriver(DRV_LCD, LCD_CARACTER, 'i');
 callDriver(DRV_LCD, LCD_CARACTER, 'f');
 callDriver(DRV_LCD, LCD_CARACTER, 'e');
 callDriver(DRV_LCD, LCD_CARACTER, 'i');
}
```

The function LCD_CARACTER in the driver DRV_LCD send a character (ASCII coded) to the LCD attached to the microcontroller. If there is any need to modify the LCD or change the port which it is connected, the application will be kept intact, the developer needs only to change the driver.

9.4 Interesting situations

There are some interesting solutions that helps the application to keep its high level while still interacting with the hardware. One of theses situation is to hide the interrupt routine inside a driver while still allowing to the application developer to define its behavior.

```
//defining the type of pointer to use as an interrupt
typedef void (*intFunc)(void);
//store the pointer to interrupt service routine here
static intFunc thisInterrupt;
char setInterruptFunc(void *parameters) {
 thisInterrupt = (intFunc) parameters;
 return FIM_OK;
}
```

The interrupt driver will store a pointer inside itself. This pointer can be change via setInterrupFunc() function. The actual interrupt function will be passed as a parameter.

Also inside the file is the compiler verbosity that indicates which function is responsible to call the interrupt:

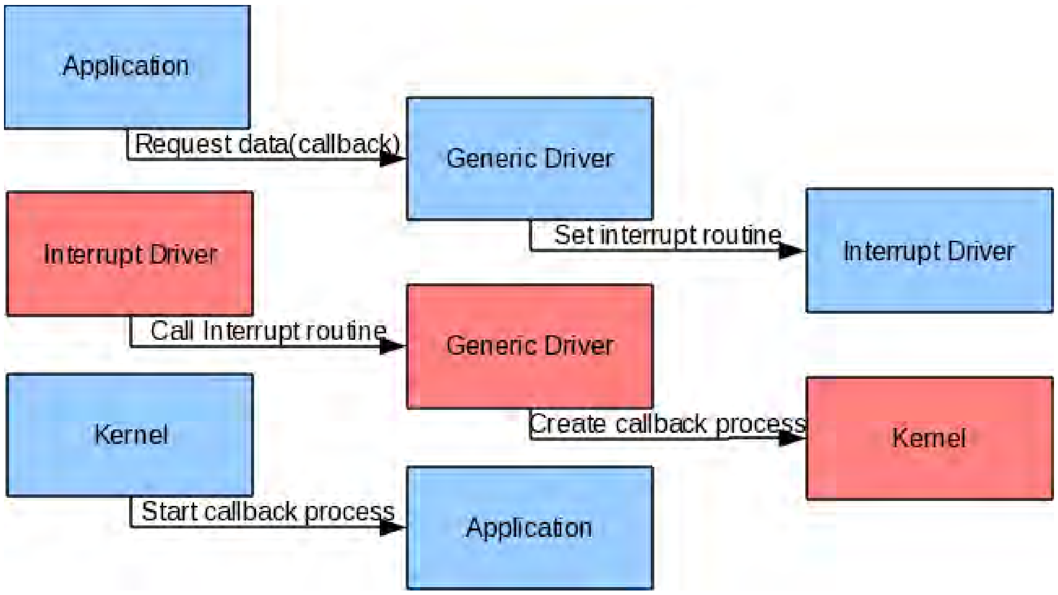
```
//SDCC compiler way
void isr(void) interrupt 1{
 thisInterrupt();
}

//C18 compiler way
void isr (void){
 thisInterrupt();
}
#pragma code highvector=0x08
void highvector(void){
 _asm goto isr _endasm
}
#pragma code
```

By using the pointer to store the ISR, the low end details of the compiler were hidden from the application.

9.5 Driver callback

In some I/O processes, we ask for something and than we need to wait for the answer, generally by pooling the end bit. With the device driver controller, we can call the driver asking for it to start its work and pass a function that will be called back when it has finished its job. This way we save CPU processing time while still getting the result as fast as possible.

In order to accomplish this, the driver must be able to rise an interruption in the system.



First the application request the data from driver and pass the callback function. The driver store the callback for latter use, start the process and setup the interrupt routine. All this are made in normal/application mode.

```
//Process called by the kernel
char adc_func(void) {
  static process proc_adc_callback = {adc_callback, 0, 0};
  callDriver(DRV_ADC,ADC_START,&proc_adc_callback);
  return REPEAT;
}

//function called by the process adc_func (via driver controler)
char startConversion(void* parameters){
  callBack = parameters;
  ADCON0 |= 0b00000010;        //inicia conversao
  callDriver(DRV_INTERRUPT,INT_ADC_SET,(void*)adcISR);
  return SUCCESS;
}
```

When the desired interruption happens, the interrupt that was set is called. The driver do all the required procedures (copy data, raise flags, etc). Before finish, the driver create an new process in the kernel. Note that all whis work is made in Interrupt mode. These function should be fast in order to avoid starvation on the normal/application mode.

```
//interrupt function
void isr(void) interrupt 1 {
 if (BitTst(INTCON, 2)) {//Timer overflow
 timerInterrupt();
 }
 if (BitTst(PIR1, 6)) {//ADC conversion finished
     //calling ISR stored in the adcInterrupt function pointer
     adcInterrupt();
 }
}
//function on the ADC driver called by the ISR
void adcISR(void){
 value = ADRESH;
 value <<= 8;
 value += ADRESL;
 BitClr(PIR1,6);
 kernelAddProc(callBack);
}
```

When the callback became the next on the process pool, the kernel will grant its share on processor time. Now, inside the callback process, we can devote more time on processor hungry tasks, as signal filtering, permanent data storage, etc.

```
//callback function started from the kernel
char adc_callback(void) {
 unsigned int resp;
 //getting the converted value
 callDriver(DRV_ADC,ADC_LAST_VALUE,&resp);
 //changing line and printing on LCD
 callDriver(DRV_LCD,LCD_LINE,1);
 callDriver(DRV_LCD,LCD_INTEGER,resp);
 return SUCCESS;
```