Budapest University of Technology and Economics
Department of Measurement and Information Systems

TaltOS

Development of an embedded operating system

Written by Péter Eördegh

Advisor: dr. Csaba Tóth

'The most profound technologies are those that disappear.
They weave themselves into the fabric of everyday
life until they are indistinguishable from it.'
/Mark Weiser computer scientist/

Introduction

Embedded systems are quite important devices at the beginning of the XXI century, and (as Mark Weiser predicted) people usually don't recognize them. Everywhere in our environment, we can see 'intelligent' devices. High-class cars contain more than 30 processors to provide our safety and the best driving performance, mobile phones are the most popular communicating devices, even our washing-machine is digitally programmable, and the e-home concept is not a distant dream any more. Many science disciplines depend on autonomous devices as well. Without embedded systems, we could not reach the Mars with two robots, and satellites would not exist at all. There are many more programs running on non-PCs than on PCs, and in most cases, their work is more vital. Complex embedded systems, dealing with multiple tasks, make a good use of an operating system (or execution environment).

Many embedded operating systems exist on the software market, some of them are free to use as well. Because of severe resource (i.e. limited RAM and Flash memory) constrains, these operating systems are optimized to handle specific tasks in the environment, so each of them is different. On the PC's we got used to the software-updating possibilities, but the commonly used embedded operating systems do not allow changing the program of the running tasks. It is usually impossible to update the software; even if such features exists then special cables are necessary to change the on-chip program in offline mode, storing the whole software again in the non-volatile memory.

If we imagine a sensor network at the development stage, the reconfiguration of fifty boards and chips can take a while with one computer and a serial cable. It would be much easier to change runtime the components with radio communication, so we can program hundreds of boards in no time.

Scalability is another very important factor in autonomous deployments when the system needs to be adaptive to the environment. If we build up the program from well defined building blocks, the code can have reduced size and optimized to its tasks. The componentized architecture also transforms our boards into multi-functioned hardware.

A component-based hard real-time embedded operating system with run-time component change ability could fulfil all these requirements.

'I'm here to let the cat out of the bag: it's not all that hard!
In fact, embedded operating systems are even easier
to write than their desktop cousins -
the required functionality is smaller and better defined.'
/Michael Barr: Programming Embedded Systems in C and C++/

Planning

Embedded Operating Systems

The big difference between the operating systems on the PC and on an embedded device is that the small systems do not have as many and as diverse peripherals and their work is much simpler. So in some cases, if they have the working sequence is easy to implement, and the device has to handle only one or two tasks, it is not necessary to use an embedded OS. When we have a complex job to do with more different tasks, which are executed in parallel but asynchronously, the developer could leverage a 'running environment'.

In this environment, the user code calls the services of the operating system, which take care of task switching, communication, and interrupt handling. These three functionalities are the most important ones in embedded environments.

More than 40 embedded operating systems exist on the market, so it is usually hard to choose from them. The most important point is to find the software, which fits the best in capabilities to the problem situation. In the following overview, I have chosen some of them, which have some similarities to the TaltOS.

TinyOS

TinyOS was developed at the University of California, Berkeley for wireless embedded sensor networks. TinyOS is a component-based architecture with network protocols, distributed services, sensor drivers, and data acquisition libraries. Component integration is done during compile time, and it has an own language built around the idea of providing/requesting interfaces. Task scheduling is non-preemptive, and it does not meet the hard real-time requirements. Because it aims at wireless networks and its open-source license, it is used in many embedded systems in research phase, mainly at universities. (http://www.tinyos.net/)

µC OS II and eCos

These two operating systems are prevalent in the embedded universe; they are called embedded general-purpose multi-tasking OS. Both of them provide preemptive multitasking, and can be used in critical situations. The code

is compiled and linked on a PC as the TinyOS. They are configurable, started to reach the component-ability and ported to many microcontrollers and test boards, but no run-time component change is supported OS at this moment. (http://ecos.sourceware.org/, http://www.ucos-ii.com/)

Symbian

This mobile operating system is designed for mobile phones and their event-driven applications. It is focusing on the wireless communication, graphical user interfaces, and on downloadable java applications. As mobile phones have more and more memory, this operating system is huge, it would be hard to use in resource-constrained situations, but this OS has the run-time program (component) changing ability. (http://www.symbian.com/)

Linux Embedded

As there are more Linux distributions, embedded Linux has more incarnations too. Its advantages are in the network protocols. The highly standardized open source system makes development and porting easy and feasible to many kinds of hardware. It is also much bigger then eCos. It can also start executables and it is free open-source software. (http://www.embedded-linux.org/)

Goals

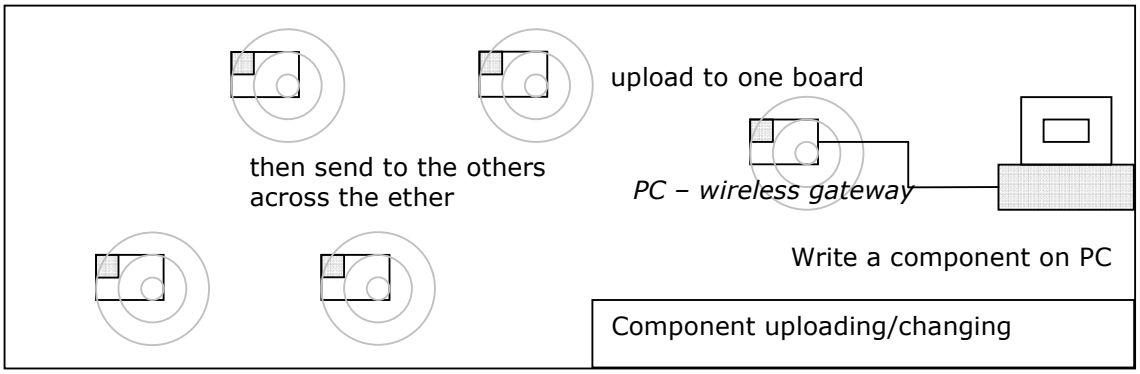
TaltOS should be a component-based, hard real-time, embedded operating system with run-time component changing ability. The targeted systems are microcontrollers, with limited resources to host huge operating systems like Linux Embedded. At the beginning of the project, I had to specify these attributes:

Component-based: component means here a code snippet that can be uploaded to the microcontroller, and started by the operating system or used as libraries. Components can be developed apart from each other; the TaltOS will provide a simple interface between them. Components can be imagined as executables or shared libraries on PCs.

Hard real-time: The best embedded systems can be used in hard real-time situations. This means that next to the performance requirements, the execution of the code is deterministic, and the execution time is determinable for every function. This operating system should be predictable, fast, and stabile. The run-time changeable software needs re-writable non-volatile memory, so TaltOS cannot be used in those critical situations where code modifying is forbidden.

Operating system: it is the base software. It provides many basic functions for the applications and it handles a task scheduler. All the other features can be realized as components. Microcontrollers usually do not have memory protection so neither TaltOS will have. It will be up to the component developer to keep the rules of TaltOS in mind.

Run-time component change ability: This is the main goal of the project. Changing includes removing, updating and uploading too. With this ability, users do not have to gather the sensors when they want to change the software on the chip. Developers upload the base software with a cable, and the components will be broadcasted through wireless links. The communication services will pass the component to TaltOS, and it will do the rest.



The final test would be that we place some wireless communication boards on a table with the uploaded base software. We send TaltOS to these systems and some components to see them working. Then we send a newer version of one of the component and check if the tasks are changed. The final step would be the removal of some unused components.

Of course, stability and determining the worst-case response time will need more time and test cases. These performance tests will be planned just after the first goals are reached.

Principles

Before starting the real planning of TaltOS, I needed to lay down some principles. In many cases, there are more possible solutions, but these principles will helped me not to spend too much time with hesitating. It is hard to make a difference between a planning solution and a principle. These ideas were born before the actual development began. Of course, principles can change.

1. This should be a portable operating system (not designed only for my prototype platform) so the assembly and chip/board specific code should be as

minimal as possible, and they should be separated from the core of TaltOS. The programming language is C because of its advantages in embedded systems: it is widely used on the microcontrollers, high and low level language at the same time. C++ and Java are hard to be used in real-time situations. C source code, with a suitable cross compiler, can be easily ported.

2. The first small program, the Bootloader what we have to upload manually, has to be as small as possible. This code will run TaltOS upon reset. Even the operating system should be a component as well, so it will be replaceable. There is no reason in reloading the Bootloader, because it must be on an exact memory address, and a reset during the erase operation could lead to a useless board.

3. The software should satisfy hard real-time expectations in serve conditions, but component changing, software uploading does not have to suit these demands. The reason is that the Flash-writing is slow and potentially unsafe. Because of limit resources, during a defragmentation we have to use a complex algorithm to place the code of new components. All the other functionalities should be deterministic.

4. No global variable in components. Microcontrollers do not use data segment pointer (which is used on PCs) and shared libraries (which are similar to the components) do not use them either. The several C programming guidelines, also discourage their use. This principle is needed for the realization of the project. Later, if there is a possibility to solve the data segment pointer's problem, which is explained later, this principle can be neglected.

5. No data space on the Flash memory for the components. The Flash memory operations are slow and plagued with many difficulties. Small memory is needed for the code of the component, and writing-erasing many times the memory can damage the Flash. Important data should be delivered (by some kind of radio communication) to a safe place that can be reloaded upon reset.

6. After a component update, we would not like to reset the board. This means that we change the component in run-time, although replacing the TaltOS itself will probably need a software-reset.

7. There must be some kind of a registry, which can identify the components. This registry must be reachable after a reset, so this must be

placed in the non-volatile memory. Every component has to have a unique identification number.

8. Every other feature (which is not in the list of goals) such as preemptive multi-tasking and inter process communication are important, but not necessary for an embedded operating system.

These principles were not the same at the beginning of the development because the lack of my knowledge and experience, but they did not change dramatically.

Resources of the development

The microcontroller

My prototype architecture was based on a Philips LPC2106 32 bit microcontroller. The following three paragraphs are adopted from the Users Manual referenced in the bibliography.

ARM7TDMI-S PROCESSOR

The ARM7TDMI-S is a general-purpose 32-bit microprocessor, which offers high performance and very low power consumption. The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler than those of microprogrammed Complex Instruction Set Computers. This simplicity results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective processor core.

Pipeline techniques are employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The ARM7TDMI-S processor also employs a unique architectural strategy known as THUMB, which makes it ideally suited to high-volume applications with memory restrictions, or applications where code density is an issue.

The key idea behind THUMB is that of a super-reduced instruction set. Essentially, the ARM7TDMI-S processor has two instruction sets:

- The standard 32-bit ARM instruction set.
- A 16-bit THUMB instruction set.

The THUMB set's 16-bit instruction length allows it to approach twice the density of standard ARM code while retaining most of the ARM's performance advantage over a traditional 16-bit processor using 16-bit registers. This is

possible because THUMB code operates on the same 32-bit register set as ARM code.

THUMB code is able to provide up to 65% of the code size of ARM, and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system.

The ARM7TDMI-S processor is described in detail in the ARM7TDMI-S Datasheet that can be found on official ARM website.

ON-CHIP FLASH MEMORY SYSTEM

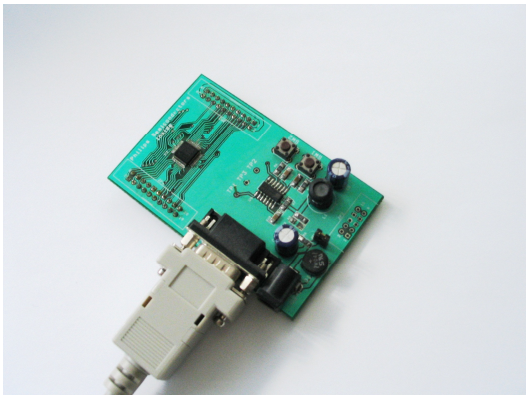
The LPC2106/2105/2104 incorporates a 128K byte Flash memory system. This memory may be used for both code and data storage. Programming of the Flash memory may be accomplished in several ways: over the serial built-in JTAG interface, using In System Programming (ISP) and UART0, or by means of In Application Programming (IAP) capabilities. The application program, using the In Application Programming (IAP) functions, may also erase and/or program the Flash while the application is running, allowing a great degree of flexibility for data storage field firmware upgrades, etc.

ON-CHIP STATIC RAM

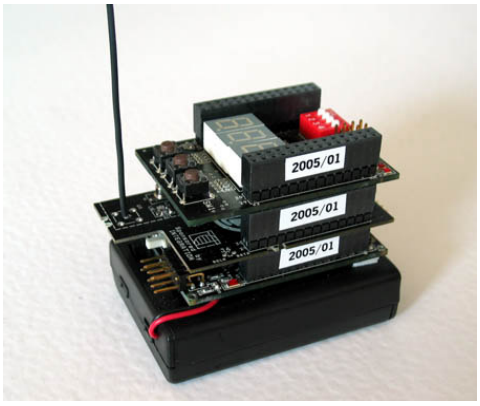
The LPC2106, LPC2105 and LPC2104 provide a 64K byte, 32K byte and 16K byte static RAM memory respectively that may be used for code and/or data storage. The SRAM supports 8-bit, 16-bit, and 32-bit accesses. The SRAM controller incorporates a write-back buffer in order to prevent CPU stalls during back-to-back writes. The write-back buffer always holds the last data sent by software to the SRAM. This data is only written to the SRAM when another write is requested by software. If a chip reset occurs, actual SRAM contents will not reflect the most recent write request. Any software that checks SRAM contents after reset must take this into account. A dummy write to an unused location may be appended to any operation in order to guarantee that all data has really been written into the SRAM.

The board

The board was built at MIT the department at the Budapest University of Technology and Economics. This test board is an early prototype of the ‘mitmót’ (http://bri.mit.bme.hu/?l=mitmot) which will be the main target of TaltOS. It has only a serial port communication and two reset buttons, but it is sufficent to develop and test on it



The testboard made by Balázs Scherer



and the mitmót

The cross-compiler

I used the gcc cross compiler provided by the GNU project. It has an ARM specific target called the GNU ARM. The reason I use this it is that it is free, and easy to develop with it in command line. The other reason is that it can be used under either Linux or Windows as well so it is widely accepted and I can find answers to my questions easily on Internet forums. The gcc also has a reliable linker as well which will bee needed to compile the components. (http://www.gnuarm.com/)

The text editor

I have chosen the Crimson Editor for editing source code and header files, linker scripts and other text files. It is simple, free and easy to use. Its Main advantage is the file-type syntax highlighting. (http://www.crimsoneditor.com/)

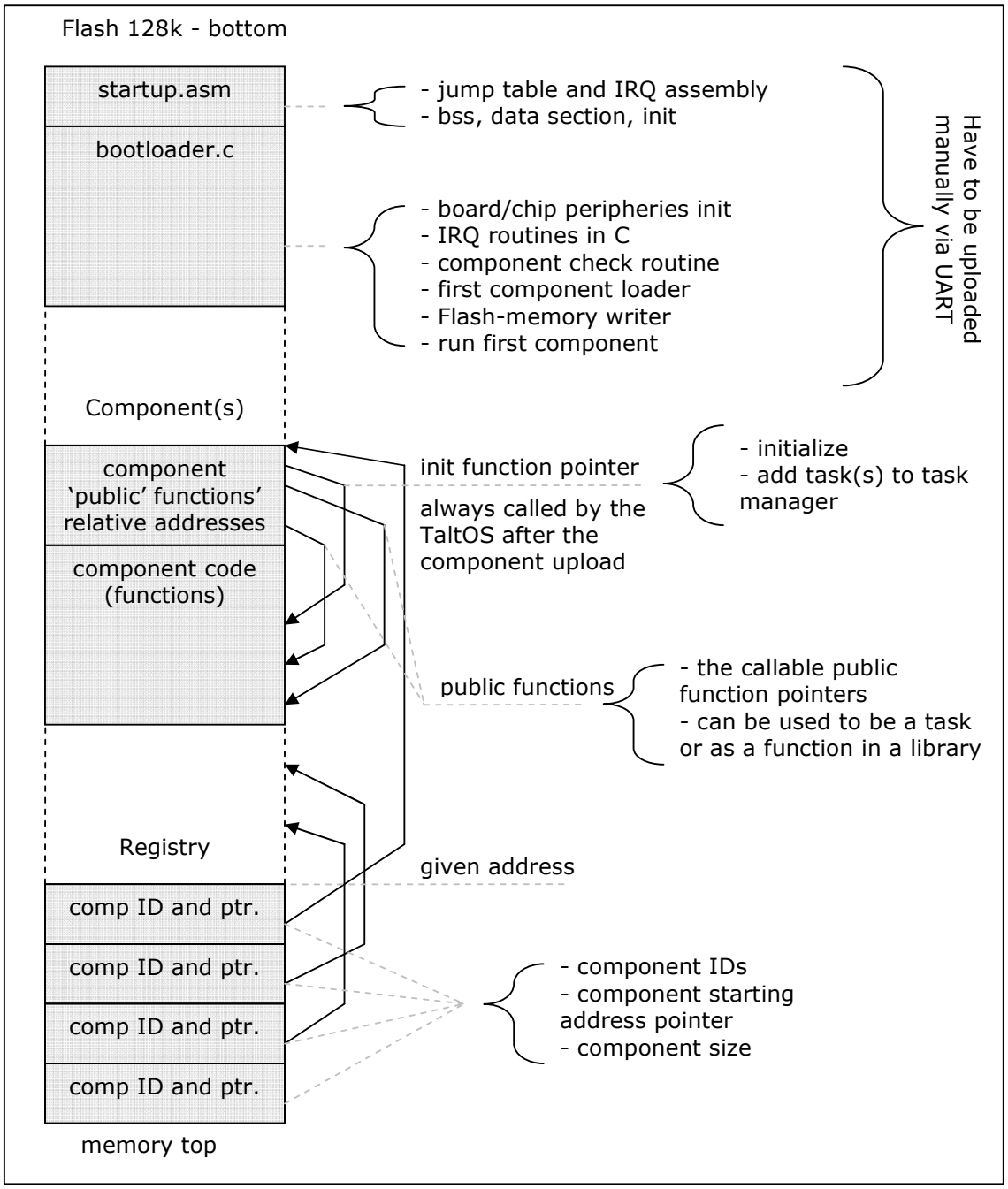
The communicating tools

For uploading the first executable code to the Flash memory, I'm using the lpc210x_isp program provided by the Philips Semiconductors Ltd. For other serial communication, I used the built-in Windows HyperTerminal (which is fully suitable for this role).

Flash memory usage plan

The Bootloader, components and the Registry in the Flash memory

The 128 Kbytes if internal Flash memory is divided to 16 sectors (8 Kbytes/sector); one sector is allocated for the LCP2106 IAP and ISP Flash writing routines. These sectors can be erased fully or written by a block of 512 bytes (in a sector there is 16 blocks). This means that the memory allocation have restrictions.



The Flash memory usage plan with functionality description

The Bootloader will be uploaded to the first sector. Since the second principle, one sector should be enough for it, and this sector will not be erased. If the first sector contains some available area, the first (TaltOS) component can be uploaded here, but this code will not be erased.

All the incoming components will be placed with the first-fit algorithm, which is nearly optimal in most cases, and it is fast. If we would like to change a component, the code will not be erased at that moment, because it is really

probable that other needed components reside on the same sector. (In addition, the lifetime of the Flash memory mainly depends on the number of erases.) In case of many components and frequent updating, a defragmenting routine will be needed to free the code of the unused components.

For the TaltOS Registry the last available sector is allocated. Registry entries contain the IDs of the components (to be able to identify them), the starting address of the code of the components (to be able to locate them), and the size used in the Flash memory (to be able to erase them). They should also give us information if the component is alive, or it was removed.

This sector should not be erased at all, because it keeps the information about the components, so the non real-time defragmenting routine could handle the cleaning of the TaltOS Registry too.

Estimating that 13 (from 2 to 14) sectors are available for the components, 12 (next to the TaltOS) components have room in the Flash memory with not more than 8 Kbytes code size (which is really big in embedded situations).

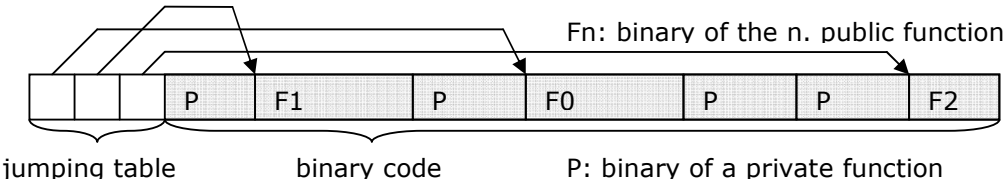
The structure of a component

Shared objects (.so) under Linux and dynamic linking libraries (.dll) under Windows have their own formats, which are based on the symbol table of the object file (.o). This symbol table holds the information about the relative addresses of the functions and the global variables. If we compile the code of a component, we will loose this information so we have to save it somehow.

Because we will have to use relative addresses, the global variable pointer would point to the Flash memory (just before the binary code) which cannot be overwritten in small sizes (so that's the point of the 4. principle).

Function pointers must be reached, so the relative addresses of the functions should be saved in the beginning of the components' binary. With this simple jump ('symbol') table and the Registry a component can call another component's function by knowing the ID and the index of the function in the table. For example, give me the fourth function of the component with the ID 32423… .

This jump table is also useful for the realization of the public - private functions. Inside a component, the developer doesn't need the addresses of the other functions, it is calculated by the compiler, but the other components can reach only those functions, which are accessible by the jump table.

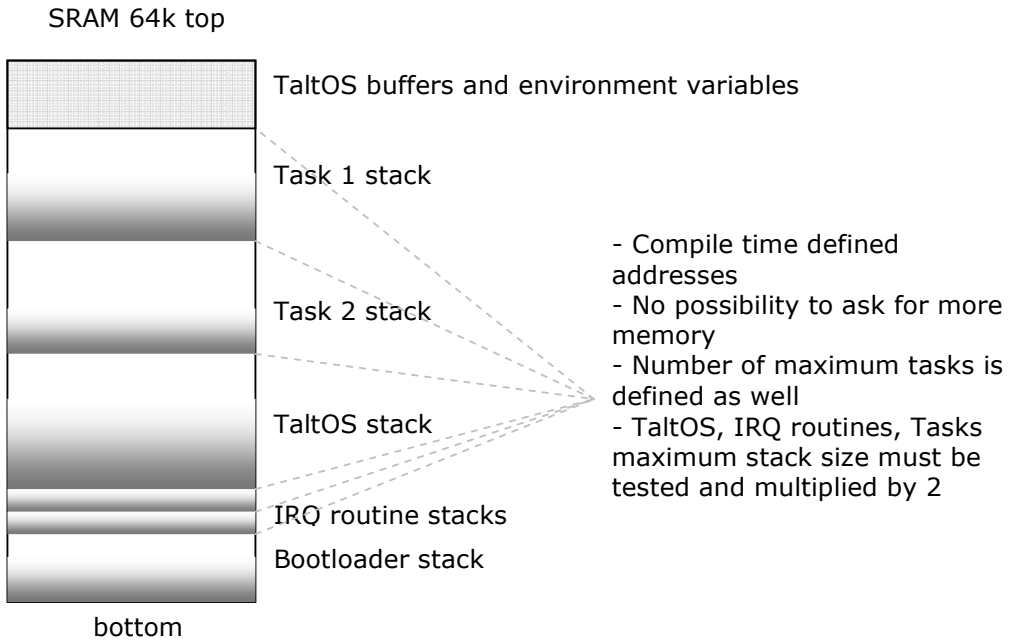


The jump table and the binary of a component

To realize relative addressing, we have to use processors, which are capable for the relative jump. Luckily, the ARM core processors can usually handle this compiler option.

SRAM usage plan

This microcontroller has 64 Kbytes volatile SRAM, which will be used as the stacks of the tasks, and the TaltOS. This will be the place for communicating, Flash-operating buffers, and the TaltOS environment pointers as well. This means that TaltOS will use some kind of global variable, which is compliant with the fourth principle.



SRAM usage plan

TaltOS, like any operating systems, communicates with the hardware, so its functions will be launched during interrupt routines. This means that there is no possibility to pass the address of a local variable in argument, the task management will surely need global variables.

The hardware specific code is compiled together with TaltOS. This must be ported to every new system, so adding hardware handling functions to the TaltOS should not be a problem. Components may use the hardware directly too, but all the interrupts should be handled by TaltOS.

As in many real-time embedded systems, there is no dynamic memory allocation support in TaltOS. The tasks can be launched and their stack starting address will be defined. Of course, this determines the maximum number of tasks as well.

TaltOS buffers can provide spaces for inter process communication next to the input output and Flash writing operation buffers. This communication can be a mailbox-like solution, because not only one sensor-board will work at the same time, but also more autonomous entities can work together and TaltOS should give the possibility to the developers to make a real cooperative system.

Provided functionalities and services

TaltOS like any operating system provides services. To be realistic this is the only purpose of its exigency. These services should make the development of the components easy. Interfaces are not only needed between the TaltOS and components but operating system functions must be reached from the hardware side code as well.

General input and output buffers

These buffers are general purpose FIFO-s used by the TaltOS itself. Messages arrive to the board through the main communication channel are to be placed to the input buffer, which will be treated by TaltOS in time. On the other hand, TaltOS can write to the output buffer, and when the hardware or the channel has free capacity, the buffer can be read.

Components will arrive across this buffer as well and the messages to the tasks.

Task management

From the viewpoint of a component, task management means to be able to start tasks, which are nothing more but functions placed in a list with given priority and an argument. The task manager will launch the given functions in time.

Tick

The ticking is a hardware side service. This function is called when a timer interrupt arrives, and this will be the key to the preemptive task changing. The task running statistics can be monitored at this point as well.

Component reaching

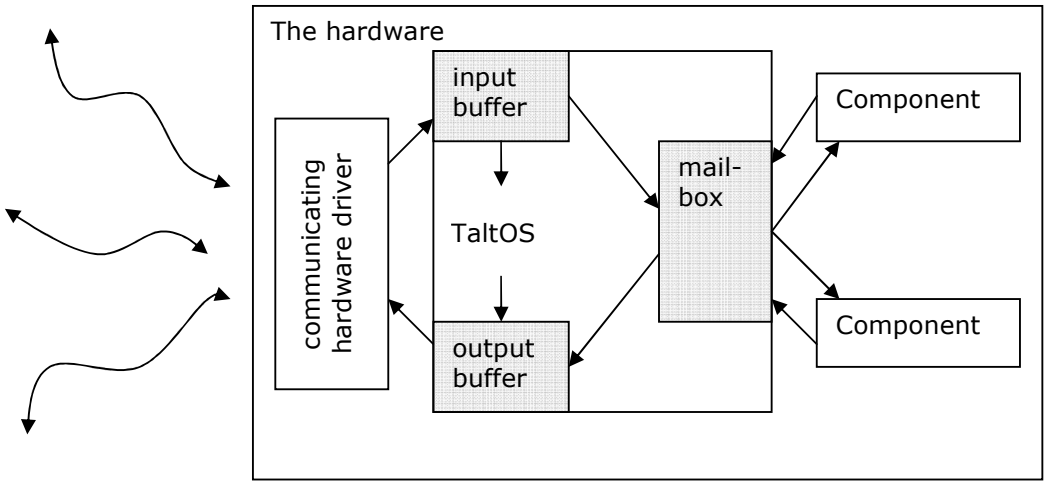
To provide interface between components, TaltOS gives a simple solution to reach the code of another component. The developer only has to know the ID and the number of the function, as it was mentioned in the Flash memory usage plan. This mechanism is used to reach the TaltOS public functions as well. This looks like an endless circle because we need this service to be able to reach all services, which include this one as well. So one or two function pointer will be passed to the started task, and then they will have the ability to reach everything.

Component handling

After the component is uploaded to the memory, it should be started somehow. As a convention, the first function found in the jump table will be called. If this is only a library, it shouldn't do anything. This init function can be imagined as the 'main' function of the component.

Commands

We got used to the command-line prompt of our operating system of the PC. TaltOS should also provide some methods to start or remove components manually. Later the monitoring functions like memory usage, CPU times, mailbox and buffer status or any other services should be realized as a component, which could act like a shell over the TaltOS.



The mailbox and the input-output buffers

Messaging

The method of a mailbox-like communication is useful in embedded situations. There shouldn't be much data transmitting, only some signs and short messages about the last measured values. Components can read (with remove)

or peak (no remove) the last message with the given ID. They will also have the possibility to send messages to other component inside the system or to another nearby board.

Prompting and messaging were not the goal of TaltOS so their realization will be one of the last steps of the development.

The Bootloader

The only functionality of the Bootloader is to start the TaltOS if it is on the chip (of course next to hardware initialization routines) upon manual reset or system halt. As TaltOS itself, this will launch the first function of the uploaded component with the given ID. If the TaltOS is not in the memory, it has to wait as long as a new component has arrived. The Bootloader puts the new component to the memory, and runs it.

The first situation is possible in case of a reset or a collapse. The second happens when we want to program the chip for the first time or when we removed the TaltOS from the Registry.

Initialization

When TaltOS is launched, it has to do some initialization, so it cleans all the buffers, environment variables and the task list. Interrupt handling was set up during the initialization of the Bootloader, so these values have to be overwritten.

After the base initialization, the TaltOS checks every entity in the registry if there is a live component. In case of a reset we have to start every component again just like after their uploading.

After all initialization routines finished, the TaltOS loop can be started.

'I naively thought, "Well, it can't be that difficult to write a kernel.
All it needs to do is save and restore processor registers"
That's when I decided to try it out and write
my own (part time at night and on weekends).'
/Jean J. Labrosse: MicroC/OS II: The Real Time Kernel/

Realization

TaltOS Bootloader

The Bootloader had grown form the first programs I wrote to the LPC2106. It is capable for many things that TaltOS itself is capable for. It has to be able to accept a component, to be able to launch the first function in the jump table, handle the UART, etc. First, I had to expand the basic assembly routine, which is able to start the C code. The ARM core is capable to throw four kinds of abort during its operation. For every abort, the processor jumps to a user specified address in the memory. I have used this aborting to help me debugging the code I was writing. The two most frequent errors where data abort and prefetch abort, which signs that the binary code is not understable, does not match the required 32-bit format. Therefore, when I received the 'AD' characters through the serial port, I knew something was wrong (this works under TaltOS as well).

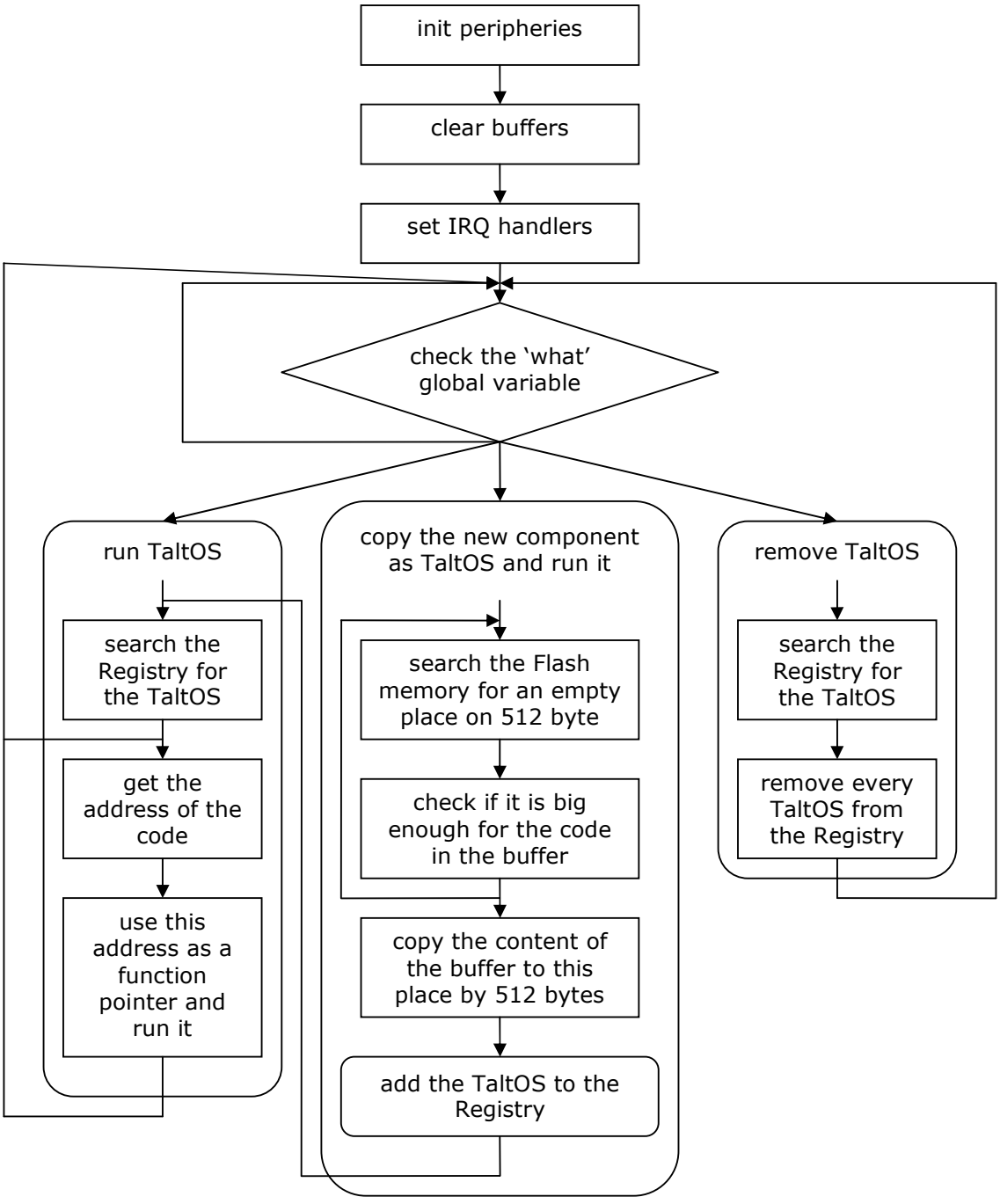
```
do_prefetch_abort:
      mov     r0, #0xE0000000         //Write "A" to USART0 means prefetch abort
      add    r0, r0, #0xC000
      mov     r1, #'A'
      strb   r1, [r0]
do_data_abort:
      mov     r0, #0xE0000000         //Write "D" to USART0 means data abort
      add    r0, r0, #0xC000
      mov     r1, #'D'
      strb   r1, [r0]
```

This was my first encounter with the ARM assembly. In some cases, I compiled a C code snippet only to analyze representation of a routine in assembly, so I have learnt from gcc. For example, the code above shows that the 'mov r0, #0xE000C000' is not a valid instruction, it must be break up to a 'mov' and an 'add' instruction.

At the development phase of the project, the Bootloader is a bit different. TaltOS cannot remove itself yet so I have added some string-recognizing feature to the Bootloader to be able to run, copy and run or remove TaltOS. These commands are the part of the communication, which will be detailed later.

To find a memory area for TaltOS I used the first-fit algorithm. To check if the TaltOS fits to a place I only had to check the first bytes of every 512 bytes, because the Flash memory can be written by this size of data. If the first four

bytes (an unsigned long) hold the 0xFFFFFFFF value (Flash memory is full of 1s when it is empty,), this part can be overwritten. Valid data cannot hold this value because it is an invalid ARM instruction, and no data storing is possible in this memory. So if a block is empty the program checks the next block until there is enough space for the component. If it runs into some kind of written data it continues the search for the next free block (the code cannot be fragmented).



The Bootloader - flow chart

During this search procedure, it is possible that TaltOS starts in one sector and ends in another so the Flash programming procedure has to be called by 512 bytes as well. This routine (called IAP: In Application Programming) is in the memory of the LPC2106. It uses the top 32 bytes of the SRAM (this is important to remember it), and it is implemented in the ARM thumb mode (Here the ARM core works with 16 bit long code. This is useful to know for the compiling: to be able to jump to an ARM thumb code the arm-elf-gcc needs the -mthumb-interwork option).

The LPC2106 is capable to use vectored interrupt handling which is fast and easy to use with function pointers:

```
void lpc2106_vectored_interrupts_init(unsigned long timer0_irq_handler,
                                      unsigned long uart0_irq_handler,
                                      unsigned long default_irq_handler)
{
REG32(VICIntSelect) = 0x0;       // All interrupts are assigned to IRQ category
REG32(VICIntEnable) = 0x50;      // Timer0 (bit4) uart0 (bit6) interrupt is enabled
REG32(VICVectAddr0) = timer0_irq_handler;     // Timer0 IRQ handler
REG32(VICVectCntl0) = 0x24;      // interrupt source is timer0 (source 4)
                                 // and enable (bit 5)
REG32(VICVectAddr1) = uart0_irq_handler;      // UART0 IRQ handler
REG32(VICVectCntl1) = 0x26;      // interrupt source is uart0 (source 6)
                                 // and enable (bit 5)
REG32(VICDefVectAddr) = default_irq_handler;
}
```

TaltOS Registry

TaltOS Registry resides in the Flash memory starting from address 0x00 01 C0 00 which points to the beginning of Sector #14. This whole sector holds the Registry for two reasons. The first is that this sector should not be erased, because loosing the information of the components would lead to a useless system. The second reason is that if I do not erase it, the used but removed data in it will be some kind of garbage, so it must be big enough to be able to update and remove components many times without running out of space.

On the testboard there is not any external memory, so during development, I had only the Flash memory to work with. One can attach a 64 Kbytes EEPROM to the ‘mitmót’, so the Registry can be stored in this easily rewritable external memory.

Registry entries

The Registry (at the moment) holds 4 word long values. The first two unsigned longs (two words) hold the ID of the component. The third one is an address, which points to the beginning of the component, and the last one is the code size in 512 bytes (rounded).

A Registry entry

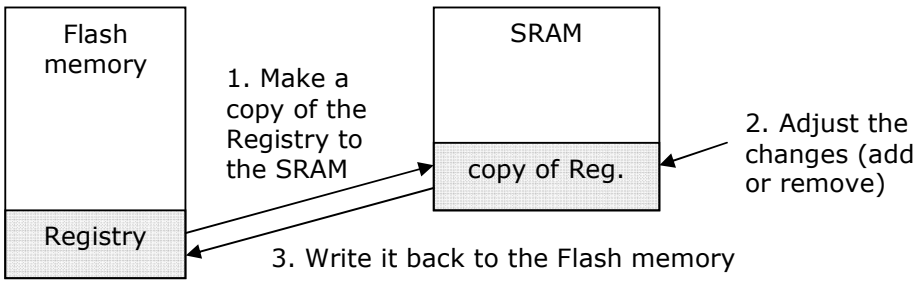
ID1	ID2	address	size

```
typedef struct
{
        unsigned long ID1;
        unsigned long ID2;
        unsigned long* address;
        unsigned long size;
}       TaltOS_reg_entity;
```

I chose to have 2 IDs only because three word long value is a bit harder to handle. For example, the last place in the registry would be 1 word long and binary representations of editor programs are always using four word long columns. Furthermore, one of the IDs can be used to identify the company, which developed the particular component, or this can be used for any other purpose.

Adding and removing from the Registry

After uploading a component to the memory, it must be registered in the Registry. In the Flash memory I can change 1s to 0s, so I search for the first empty (filled with ones) place in the Registry (address pointer cannot hold the value of full 1s), but the Flash memory can be written by the minimum of 512 bytes. Therefore, I make a copy of the Registry to the SRAM, search a space, and then write it back to the same address.



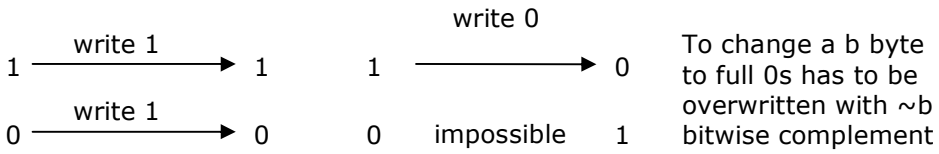
When TaltOS wants to remove an entry, it can only erase the IDs, but cannot free them for a next component. The reason is that the only way to write 1s to the place of the 0s is to erase the whole sector. However, this operation wears the Flash memory (in the specification the maximum number of erasing operations is 10 000) so that is another reason not to erase the Sector #14 as many times as a component is removed/updated.

Therefore, I decided to write 0s to the IDs of the erased components. I wanted to keep the address and size information to be able to do a defragmentation (to remember the place of the removed component). There was no reason in writing 0s over the code as well.

The Flash writing problem

Here I ran to a problem. When I wanted to write something in a memory cell, which was not just filled with 1s, the result was not the expected. If the ID of TaltOS was 0x00000001, it turned to be 0x02000000 or 0x00040000 after trying to the erase it. It was non-deterministic and made one or two bits change back to one.

I read again the documentation of the IAP so that I could fix the problem. At this point, I realized that according to the manual, I have to write 1s to the place of 0s as well. So writing 1 to a bit mean to leave its value unchanged, writing 0 means to change the 1 to 0.



Unfortunately, this little fix did not solve the problem. I had started to read the forums about LPC2106 IAP bugs, and I found a small group of other developers who also faced this error.

The Philips Semiconductor support could not help the problem either. They said that in the Flash memory every 128 bytes have some error correcting bits, which seems to be malfunctioning in case of a rewrite. They suggested to update the basic software on the chip (this one is also called Bootloader, originally it was the 1.3 version, I upgraded it to 1.52). Of course, this change did not help as well. I have also tried to erase again the bad value, but it did not change a bit.

Therefore, I decided to make some kind of correction myself. Flash rewriting seemed to be unreliable, but I was removeing only some IDs from the Registry away, so I only need to know if an ID is garbage. The word long ID contains 4 bytes on this 32-bit processor. I have chosen the last byte to be constant 0x4E which in binary representation: 01001110 (half zero half 1). If I rewrite this value with 10110001 the probability that this rewrite does not change a bit in the value is nearly 0.

A Registry entity

'E' means the 0x4E code, which signs that the ID is not garbage.

This risk in the system exists only because of a hardware problem. In a real situation, TaltOS should work on a more sophisticated hardware.

With this change, the full length of the IDs is not 8 bytes any more, it is only 6 bytes. So at this point, 2 word long error corrected ID seems to be a good choice.

The full size of the sector is 8 Kbytes, with 32 bytes long entries there is space for 256 registrations. This means that if we use 25 different components (which seems to be lot), each of them can be updated ten times before the Registry needs a full cleanup. The space for the code parts will be filled up faster.

Defragmentation is not realized yet, but a possible solution can be to do the necessary cleanup in an SRAM copy, free another sector (e.g. Sector #13), copy the full arranged registry to that place, erase the Sector #14 (This is the critical operation), write back the arranged Registry to the Sector #14. Then erase the Sector #13, which acts like a backup of the Registry during the critical operation.

TaltOS SRAM usage

At the time of writing this document, TaltOS is not capable to launch its tasks with a separated stacks. This part needs a deeper knowledge of the ARM assembly, and more time is needed to gather that kind of knowledge. So now, the tasks and TaltOS are running in the basic stack of the Bootloader. Dedicated stacks are needed only for preemptive multi-tasking.

Stack pointer of the interrupts is placed away according the linker script and the basic startup assembly code I got at the beginning of the development.

TaltOS environment (compile time memory allocation)

As it was mentioned, TaltOS will need some kind of global variables and buffers allocated from the top of the memory. This allocation should be done at compile time by C language preprocessor macros. In hard real-time embedded systems, it is nearly forbidden to use dynamic memory allocation.

Compile time memory allocation is easy to use and comprehensible. Of course, the system will not be so flexible, but TaltOS can be compiled for every new environment with modified memory allocation if it is necessary.

```
#define LPC2106_RAM_TOP_PTR (0x4000FFE0)

//Flash operation buffer
#define TALTOS_FLASH_OP_BUFFER_SIZE (0x2000)  //This is a full sector size
#define TALTOS_FLASH_OP_BUFFER_PTR (LPC2106_RAM_TOP_PTR-
TALTOS_FLASH_OP_BUFFER_SIZE)

//Input data buffer
#define TALTOS_INPUT_BUFFER_SIZE (0x800)      //This is a half of a sector size
#define TALTOS_INPUT_BUFFER_PTR (TALTOS_FLASH_OP_BUFFER_PTR-
TALTOS_INPUT_BUFFER_SIZE)
```

```
//Output data buffer
#define TALTOS_OUTPUT_BUFFER_SIZE (0x800)     //This is a half of a sector size
#define TALTOS_OUTPUT_BUFFER_PTR (TALTOS_INPUT_BUFFER_PTR-
TALTOS_OUTPUT_BUFFER_SIZE)

#define TALTOS_ENV_PTR (TALTOS_OUTPUT_BUFFER_PTR-sizeof(TaltOS_env))

#define TALTOS_TASK_LIST_SIZE (10)
#define TALTOS_TASK_LIST_PTR (TALTOS_ENV_PTR-
(TALTOS_TASK_LIST_SIZE*sizeof(TaltOS_task)))
```

The topmost RAM pointer is not 0x40010000 because the top 32 bytes is used by the built-in IAP. Anyone can easily modify the size of the buffers or the number of tasks, the pointers will be shifted, so no other calculation is needed.

TaltOS functionalities and services

TaltOS component handling

Component uploading in TaltOS is the same as it was in the Bootloader except for some information, which will be passed to the first (initialization) routine. This information will be necessary to reach other components or to start a task.

TaltOS component reaching

To be able to reach functions in other components (including TaltOS as well) the developer will have to use two services. He has to know the IDs of the component and the index of the function as well. The first function will return the starting address of the component (which points to the first element of the jump table).

```
unsigned long* TaltOS_get_component_address(unsigned long comp_id1, unsigned long
comp_id2)
```

The second function will return the absolute pointer of the needed function:

```
unsigned long TaltOS_get_component_function(unsigned long* address, unsigned long
index)
```

These two functions could be implemented in one, but a component will more likely to use only one or two other components. Therefore, it is better to store the address, and call only the second function whenever the developer needs to reach another feature.

To make the development easy, it is better to write a header for every component we develop in the following way:

```
/*TaltOS as component - IDs and public functions*/

#define TALTOS_ID1 (0x0000014E)
#define TALTOS_ID2 (0x0000014E)
```

```
#define TALTOS_INIT                             (0x00000000)
#define LPC2106_UART_HANDLER                    (0x00000001)
#define LPC2106_TIMER_HANDLER                   (0x00000002)
#define LPC2106_DEFAULT_HANDLER                 (0x00000003)
#define TALTOS_CHECK_ORDER                      (0x00000004)
#define TALTOS_RUN_AND_LIST_COMPONENTS          (0x00000005)
#define TALTOS_COPY_COMPONENT                   (0x00000006)
#define TALTOS_A_NEW_FEATURE                    (0x00000007)
```

So the usage will be something like this:

```
unsigned long* TaltOS_address;
void(*func_ptr)(void);
TaltOS_address = TaltOS_get_component_address(TALTOS_ID1, TALTOS_ID2);
func_ptr = (void(*)(void))TaltOS_get_component_function(TaltOS_address,
                                                        TALTOS_A_NEW_FEATURE);
func_ptr();
```

The only problem with this solution is that the component will have to know the absolute address of the two given TaltOS functions. Therefore, these pointers will be passed to the tasks and the initialization routine as well.

TaltOS Task Management

For the component developer the method to add a task to TaltOS task list is only calling a function:

```
unsigned int TaltOS_add_task(unsigned long address, unsigned short priority,
                             unsigned long argument)
```

The address variable is the pointer of the function, which should be called as a task. This value can be the returned address of TaltOS_get_component_function. The priority can be arbitrary; as long as there is no support to do some kind of round-robin scheduling, it is the responsibility of the developers to use the priorities. The argument will be passed as the first argument of the task (function) called. The TaltOS_add_task function searches a place in the task list and stores the information there. An entry in the task list can be described by its C struct:

```
typedef struct
{
      unsigned long address;
      unsigned short priority;
      unsigned short state;
      unsigned long argument;
} TaltOS_task;
```

This list of information will surely grow in time, but the compile-time memory allocation is using the sizeof(TaltOS_task) do determine the size of the task list. For example, the actual stack pointer or the round-robin value can be saved here when preemptive multi-tasking will be realized. Without this feature, the task list is nothing more then a list of functions, which should be called in priority order.

Non-preemptive task switching is realized in the main loop of TaltOS. This routine sets the task parameters back to the basic values after the run, searches for the new task and runs it with the given parameter. If tasks have the same priorities, it works like a task FIFO.

Component communication

Communication between components is not yet implemented, so they cannot receive any kind of data, but there is a possibility to send some information trough the output buffer of TaltOS. This feature does not interfere with other functionalities of TaltOS.

TaltOS and Bootloader communication

Communication between the Bootloader and TaltOS is (at this point) based on four byte long messages. Reason is that on 32-bit microcontrollers it is only one assembly instruction to compare two 4 bytes long values. So at communication side, the interfaces of the input and output buffers are the following:

```
//The input buffer's input function, can be called by the microcontroller
void TaltOS_long_input_write(unsigned long inlong);

//The output buffer's output function, can be called by the microcontroller
unsigned int TaltOS_long_output_read(unsigned long* ret);
```

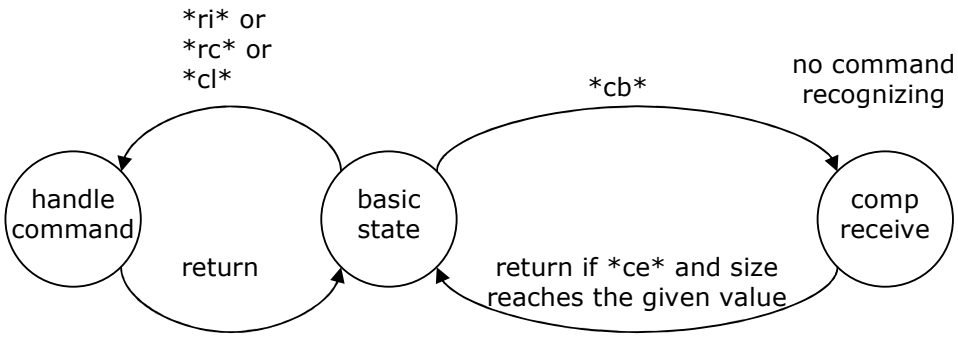
The serial buffer of the LPC2106 is capable to send interrupt just after every 4 characters. On another microcontroller, a small compile-time allocated buffer should be useful to handle this function.

As the Bootloader and TaltOS communicate with the PC in a very simple manner, they are capable to understand some unsigned longs as commands or tokens. These possible commands and tokens are the following:

Character representation	sense under TaltOS	sense under Bootloader
ri	run init functions of all the components	run init function of TaltOS
rc	remove component with the IDs arrived just before this command	remove TaltOS
cl	List the IDs of the components	no reaction
cb	signs the beginning of a component	signs the beginning of a component
ce	signs the end of a component (save and run component)	signs the end of a component (save and run component)

These commands can be typed in using any kind of serial communication software like HyperTerminal.

During component uploading, it is possible that the binary code contains some of these words so that is the reason for begin and end tokens as component delimiters. That is also possible that the '*ce*' sing arrives before the end of the component so we will need the size information of the uploaded component. The procedure to handle these commands is the following:



Command and sign handling

The size information and the last '*ce*' are redundant information, but if some other characters arrive during a component uploading, the procedure can recognize this and ask for a restart.

Component building

As it was described before, the code of the component has to be packed in with tokens and size information. The public function pointers must be placed to the jump table as well. The gcc linker script is capable to handle all these modifications during the linking of the object files. The other solution could be to write a PC program which packs the components, but it is easier and more straightforward to do it during the linking phase.

The Evaluator 7T board

The arm-elf-gcc has many additional files like linker scripts and startup assembly codes that can help the development. Unfortunately, these where developed for the Evaluator 7T board with the Samsung S3C4510/KS32C50100 (ARM7TDMI) chip, so most of them are useless on the LPC2106. For the Bootloader and the components, I have received some aid, but basically I had to write my own linker scripts and startup assembly code.

Linker script

Packaging of the code and creation of the jump table can be done by the linker. We have to make a custom script which is not usual during a PC program development. Here it should look like this:

```
OUTPUT_FORMAT("elf32-littlearm")

SECTIONS
{
	.comp_begin :
	{
		LONG(0x2A62632A);
		LONG(ABSOLUTE(_ecomp) - 8);
	}
	.jumptable :
	{
		LONG(ABSOLUTE(init) - 8);
		LONG(ABSOLUTE(function1) - 8);
		LONG(ABSOLUTE(function2) - 8);
	}
	/* .text is used for code */
	.text :
	{
		/* define symbol name _ftext for start of code */
		_ftext = ABSOLUTE(.);
		*(.text)
		. = ALIGN(4);
	}
	.comp_end :
	{
		LONG(0x0000024E);
		LONG(0x0000034E);
		LONG(0x2A65632A);
		_ecomp = ABSOLUTE(.);
	}
}
```

To add a long value to the binary, the LONG() command should be inserted. In the .comp_begin section we can find the *cb* (which is represented in hexadecimal code), and the absolute value of the _ecomp symbol. This symbol is defined at the .comp_end section. In the jumptable section we can find the symbols of the functions we want to be accessible by the other components. Here the function names hold the value of the pointer, which addresses the function. The 'init' function will be called after uploading of TaltOS, this must be at the first place. The binary code is in the text section, to be sure that this is on 4 bytes boundary we have to put the . = ALIGN(4) at the end of this section. The last (comp_end) section includes the ID1 and ID2 with the error correcting last bytes (here ID1 is 0x00 00 02 E4 and ID1 is 0x00 00 03 E4) and the *ce* sign.

The -8 constants are needed because we put two longs at the front of the component, which will be removed during the component upload. Therefore, the symbols are shifted with 8 bytes, and this -8 will replace this modification.

cb (4 bytes)	size (4 bytes)	jump table (n * 4 bytes)	code (aligned to 4 byte)	*ce* (4 bytes)

Package of a component

Compiling

First we have to compile the source C code to an object file with the gcc compiler so the –c (compile but not link) option is to be inserted. Here the –fPIC flag must be set to use the program counter relative jumps in the code so that the code will be position-independent. I use the –O2 option for optimizing and the –Wall option to get all the possible warnings.

```
arm-elf-gcc -c -fPIC -O2 -Wall *.c
```

At this point new '*.o' files should appear in the same directory. Now we have to use our linker script to make i.e. ELF file which will contain the jump table and the packaging as well with all the object files together. We have to use gcc and set the –nostartfiles flag, which means that we do not want to add any built-in library code (they are not suitable). To pass the linker that we use our own linker script the -Wl,-T -Wl,component_linker.ln should be inserted.

```
arm-elf-gcc -nostartfiles -Wl,-T -Wl,component_linker.ln -o component.elf -Wl,-Map
-Wl,component.map *.o
```

The –Wl,-Map –Wl,component.map will create a memory map file which can be useful to check if the jump table is OK. After this command, we have the ELF representation of the code, which can be easily converted to a flat binary format:

```
arm-elf-objcopy -v -O binary component.elf component.bin
```

HyperTerminal can upload this as a text file, and the TaltOS (or the Bootloader) should do the rest.

'Simplicity is the ultimate sophistication.'
/Leonardo Da Vinci/

Evaluation

Development

In the beginning of the summer in 2004, I have compiled my first code to this microcontroller. For more than a half year I wrote test applications to get used to cross-compiling, serial port communication, Flash writing and function handling. As it was my first experience to write something to a non-PC device, the development was slow, and I could not escape from learning more about compilers, linkers and startup assembly codes, which are usually unknown in a PC oriented project.

At the beginning of the year 2005, I implemented the Bootloader by combining my previous sources I wrote to gain experience. Since then – of course – nearly everything was changed in it. Now, the Bootloader is capable to do more things than its main job. We can order it to run or remove TaltOS, and to upload the incoming component to the memory, then register it in the Registry. It is fully functioning and ready to run, so it has reached its goal.

Most of the Flash-writing problems occurred during the spring of 2005, and without any kind of powerful debugging tool, the sources of the problems where really hard to find. I could only use the serial port to send some "I've reached this part of the code" messages. In some cases, this monitoring had an impact on the execution of the program as well.

The coding of TaltOS had started only in May when I realized how hard is to think over every possibilities in this embedded environment with dynamic components. I did not want to make fast steps, so in most of the time I was only planning a right way to solve a problem by working out more possible solutions and choose between them.

Current state

The uploading and removing part of the basic features had met the first requirements in the last version of TaltOS, although it has been tested on the prototype board with serial cable communication. Therefore, the 'big test' (which was mentioned right after the goal specification) is waiting for the implementation of the radio communication and a gateway between the PC and the 'mitmót' boards.

Serial port communication is at a good level by using all the capabilities of the given hardware. The general input and output FIFOs were essential to success of the TaltOS so their development were among the first tasks.

Non-preemptive priority task management is also ready to use. With this simple solution it is possible to realize a kind of multithreading by re-scheduling short-running tasks to the task list. They can restart themselves again at the end of their run. The developers should be careful with the priorities and long run tasks at this point of the development.

The actual binary size of the operating system is less than 4 Kbytes, so it uses only a half of a sector.

Source code of the TaltOS is fully deterministic, so the worst-case response time is countable. During the tests, the serial port communication and the Flash-writing were the relatively slow operations. The 'upload and run' of TaltOS itself takes less than 1 second. (The maximum attainable speed of the serial port communication depends on the external clock frequency on the board, which is now working on 38400 bps, but with a different crystal oscillator, it can go up to 115200 bps.)

After four month and a year starting from zero experience, the overall result is quite promising, because the PC like program-running experience of an embedded system is possible, as we expected. Although this was the main goal of the project, TaltOS needs a long time to be a real operating system. Yet it is better to call it a running environment.

This software is comparable only to TinyOS that was developed in a university/research environment or to the first version of µC/OS, which was written by only one experienced developer on weekends. At this time, both of them are quite popular, and many person-years work are behind them, TaltOS has less than 1 person-year. Many of the more powerful embedded operating systems have a company and a large number of developers behind them.

Every functionality helps only the component-development, which seems to be a l'art pour l'art concept, but the ongoing tasks as porting it to the 'mitmót', gives the possibility for other students to try out this software in some problem situations.

The innovation

The code changing is not a novelty nowadays, neither the writing of an embedded operating system makes this project unique. Only the newest microcontrollers have enough resources to be able to handle a PC-like program changing. The big thing is the idea (thanks to Péter Völgyesi) to put these two

together, and to find the simplest solutions to make this realizable, which is (as we saw) not only possible, but already done.

Further steps

Short time tasks

The most important immediate challenge in TaltOS is to finish the porting to the real 'mitmót', and to execute the goal test on it. This job needs the radio communication code and the PC–'mitmót' gateway, which will also run on a similar system. The shift from serial cable to radio link communication is also need to be done, but here I will receive a lot of help from another student.

After checking the performance on the new board, some new possibilities revel. For example, the benefits of the EEPROM add-on can be used to evade the previous Flash-writing problems in the Registry. These possibilities have to be analyzed, so that TaltOS can reach higher and higher levels.

The next task is to figure out the concept of the inter-process communication. This should work also as 'inter-board process communication' as well, to be able to develop cooperative systems.

The last short time task is to prepare TaltOS to update itself when a newer version arrives to the board as a component.

Long time task

Once TaltOS is up to usage, the worst-case response time should be calculated for task changing and interrupt handling. Here we need also performance and stress tests to find the limits of the TaltOS.

Another long time task is to make the task management preemptive. This feature will need much more experience of coding in ARM assembly, so it will surely take time to handle this problem.

Without defragmentation, the lifetime of a system with TaltOS is limited mainly by the hardware capabilities. The defragmentation routine will be also important, if we want to use a working system for long time with component updating.

A sophisticated component should be written later which can monitor the health of TaltOS, the SRAM and Flash usage, and the CPU-time allocated for the components.

According to the worldwide development of ad-hoc networking, some kind of algorithms should be implemented in TaltOS as well.

Acknowledgements

First, I would like to thank Péter Völgyesi for giving the main idea of the run-time component changing ability on an embedded system. He is also the one who helped me at my first steps in programming embedded systems, and who corrected this documentation. I hope that he can be pleased with the reached goals, and that I can show him once the software he had dreamt up.

After Peter had left the country, Gábor Samu took over the responsibility of the project as my advisor. I would like to thank him for all the patience he had during my reports, and for all the help he gave me solving the problems. Gábor has left the university as well in August 2005.

I would like to thank also my new advisor dr. Csaba Tóth that he took over the guiding of this project, so that I could write this document and that I can continue my work on TaltOS.

At finally yet importantly I would like to thank my family and friends for all the encouragement I have received during my work.

Bibliography

Bibliography (books):

Michael Barr: Programming Embedded Systems in C and C++,
O'Reilly, 1999, ISBN: 1-56592-354-5

David Seal: ARM Architecture Reference Manual,
Addison-Wesley, 2000, ISBN: 0-201-73719-1

Jean J. Labrosse: MicroC/OS II: The Real Time Kernel,
CMP books, 2002, ISBN: 1-57820-103-9

Bibliography (digital):

http://www.semiconductors.philips.com/acrobat_download/
usermanuals/UM_LPC2106_2105_2104_1.pdf

http://www.gnuarm.com/

http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/gnu-linker/expressions.html

http://forums.semiconductors.philips.com/forums/

http://www.tinyos.net/

http://bri.mit.bme.hu/?l=mitmot

http://ecos.sourceware.org/

http://www.ucos-ii.com/

http://www.crimsoneditor.com/