# TaltOS

A component-based hard real-time embedded operating system with run-time component change ability.

Pre-planning

I. The problem situation

Programming embedded systems is a quite important work these days. Every high-tech product contains some printed-circuit board with a processor and memory. But only a few are capable to change his on-chip software, and can do this only with special cables in offline mode, writing the whole software again on the non-volatile memory.

If we imagine a sensor network in a 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 upload a small base software, after that we can runtime change the components with radio communication, so we can program hundreds of boards in no time. The component ability changes our boards to be a multi-functioned hardware. What do we need to do that? A component-based hard real-time embedded operating system with run-time component change ability.

upload to one board

than send to the ether

Component uploading/changing
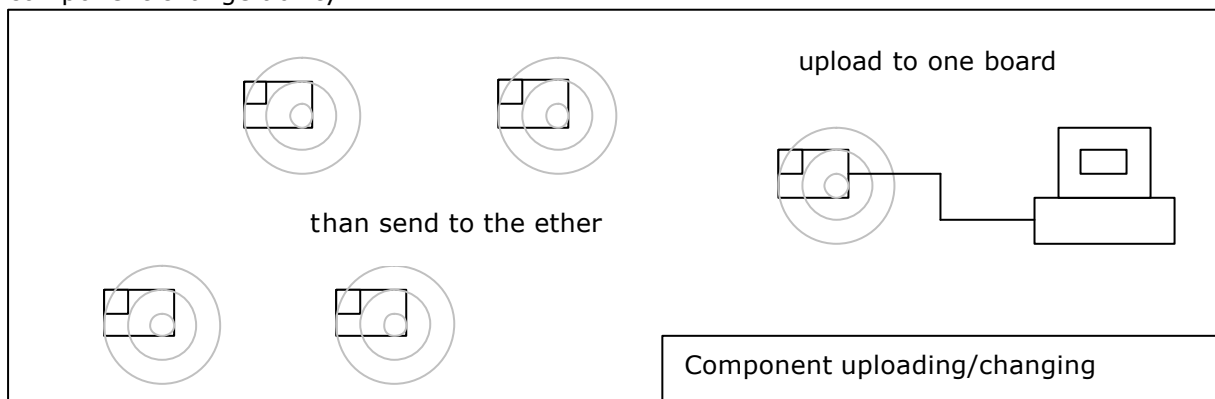
II. Explication

Component-based: The object-oriented aspect is well utilized in PC software development, but not in embedded systems. This software would like to change this. There are many reasons why we use the OOP on our PCs, but the only reason why we didn't use it in embedded systems that it needs more resources like memory, non-volatile memory and processor time. With the modern microcontrollers like the Philips LPC2106 we have all the resources we need, and it is integrated.

Hard real-time: The best embedded systems are hard real-time, so they can be used in critical situations. This operating system should be one of them. It has to be predictable and fast. The stability is also important. During the development these hard real-time properties will be the most important goals next to the component changing ability.

Operating system: The base software. It provides many functions for the applications and of course, it handles a task manager. All the other features can be realized as components. The microcontrollers usually don't have memory protection, so neither the TaltOS will have. It will be up to the component developer to keep to the rules of the TaltOS.

Run-time component change ability: Well, this is the main goal of the project. With this ability, the users don't have to gather every sensor when they want to change the software on the chip. The developers upload the base software with a cable, and the components will be uploaded through a wireless communication which can broadcast the information. The operation system will save the information, and if the component is in the memory, the TaltOS will do the rest.

III. Resources for the development

To develop this software I'm using a Philips chip called LPC2106. This is an ARM-based microcontroller with 128 Kb non-volatile flash-memory and 64 Kb SRAM on a simple evaluation board built at my university (University of Technology and Economy of Budapest). The memories and many communication interfaces are integrated in the chip. There is a detailed description about this microcontroller in the VI. chapter.

The chip contains a small built-in software I can upload a binary to the flash memory with, and another for the in-application programming of the flash. These two are the only burnt-in code in the flash memory. To compile the arm-assembly, C and C++ programs, I'm using the GNU gcc cross compiler (release: 3.4.1).
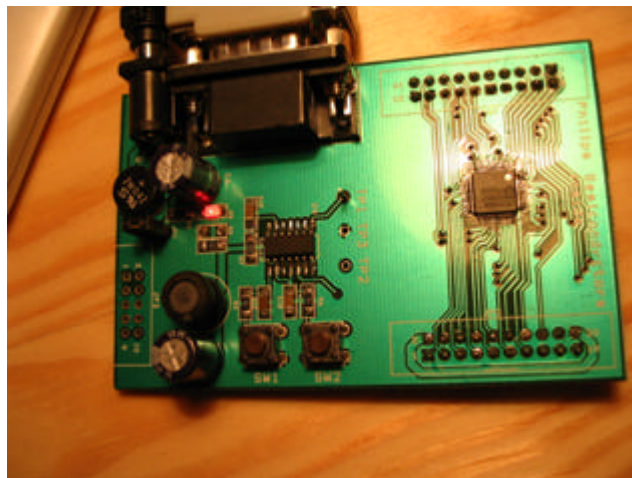
IV. Principles

After getting known the capabilities of this hardware, I needed to lay down some principles to show the way of the development. In many cases there are more possible solutions, but the principles will help not to spend much time with hesitating. Of course, principles can change.

1. This should be a portable operating system, not only designed for this Philips chip, so the assembly and chip/board specific code will be as minimal as possible, the main part of the TaltOS will be in C or rather C++. With a suitable cross compiler, it can be easily ported. The modules will be written also in C++, for the same reason.

2. The first small program we have to upload manually has to be as small as possible. Even the operating system should be a component too.

3. We only need the object-oriented aspect to use its component ability, not for the inheritance ability.

4. The software is to satisfy the hard real-time expects in serious situations, but the component changing, software uploading doesn't have to suit these exigencies. The reason is that the flash-writing is slow, and because of the small space on it, maybe we have to use a complex algorithm to place the components' code.

5. No static variable. We don't want to save data on the chip. (This can be changed.)

6. After a component change, we wouldn't like to reset the board. (This is optional; I hope we can do without reset.)

V. First ideas

```
Flash 128k

  ┌────────────────────┐
  │ startup.asm        │────── - jump table and ISR assembly
  ├────────────────────┤        - bss, data section init
  │ bootloader.c       │        - global ctors, dtors call (gcc)
  │                    │
  │                    │────── - board/chip peripheries init
  │                    │        - ISR in C
  │                    │        - component check routine
  │                    │        - first component loader
  │                    │        - flash- memory writer
  │                    │        - run first component
```

Have to be uploaded manually via UART

Component(s)

component public functions' relative addresses

init function pointer — - initialize
- registration to CT
- add task to task manager
- set ISR

always called after the component upload by the OS

component code and data (const.) (can be multiple classes)

- allocate memory for objects
- some more initializing ☺
- return interface pointer

factory functions

Comp. table (CT)

given address

comp ID and ptr.

comp ID and ptr.

comp ID and ptr.

comp ID and ptr.

In the RAM there are only stacks and heaps for the tasks and for the interrupt service routines (ISR).
We must reserve some place for the flash writing operations.

About the boot-loader

The boot-loader contains only one assembly code for the startup for the chip specific vector table and the bss, data initialization of the second part, the real boot-loader. This also means that this part have to call the global constructors and destructors of the boot-loader. The startup must be in assembly, but the boot-loader's main part is in C.

This will initialize the needed peripheries for the upload (and maybe for the OS), like the UART (radio), the system clock, the vectored interrupt table and the timer. After this, the boot-loader will check if the flash contains any executable components (it can be a previously uploaded TaltOS too.). If there is something, it will run that, if not, he will expect a component from UART (radio), put it on the flash and run it. From this part, the TaltOS (the first uploaded component) will do the rest.

About the TaltOS

This operating system component should be small too. It contains a task manager (preemptive or not, not decided yet) which manages and schedules tasks. It also contains a component loader. The tasks can call some specific functions, which add or remove tasks, maybe the OS will handle the interrupt service routines too, and the tasks can only ask to change the interrupt vector table.

About the components

The components (also the TaltOS) should be written in C++. There are components which can act as a task, others will provide only functions. There is a function that every component must contain. The init function, what the TaltOS will execute after the upload. This can add tasks or initialize some memory and constant data for himself. There can be factory functions which create objects and return the dispointer. This part will be a bit like COM. The init and the factory functions should be C functions (not in-class methods). I have to find a solution to compile components. Because we won't know the memory address of the code before the uploading, so it has to contain relative addresses.

About the component changing

This will be a privileged state, so no interrupts during this operation. The TaltOS has to manage the flash writing; because of the small space there will be a complex algorithm to range the components. During this time the TaltOS will refresh the Component Table too.

About the component table

It is not decided yet if the component would contain its ID and other necessary information and after an unexpected reset the TaltOS would build up the component table in the SRAM or there would be an exact place for the component table in the flash memory for the information. The second solution is to see on the figure. This will be the most important part of the flash memory. Without this the programs won't be able to look up the other component's necessary functions.

About the components' binary

This binary we upload will be a bit different. It is not specified yet what information would it contain. There will be a relative vector table for sure. During the project I have to specify this, too, some sort of file, which is more than just an executable.

VI. Getting know the hardware

1. The cross compiler

The first thing I had to do is to build a cross compiler to this microcontroller. I used the GNU-gcc for this but because of the windows 2000 and the cygwin incompatibility it wasn't successful. So I decided to download the built an compiled GNU-ARM gcc which was complied for the ARM-core processors as this microcontroller.

2. The Hello world! Program

As in every new environment I had to learn how to communicate with my software. This was the first step, I had a sample program for this, and with this help I leant how to write the assembly startup code, periphery initialization (set the vector table, interrupt vector table, set MAM, set the clock, uart0 init…), and with the LPC2000 flash utility I uploaded the program. With the HyperTerminal program, I could read the things what was sent by the microcontroller through the serial port. Since this part I used these two applications to test my software.

3. The task queue and processor time

To build my first kernel, which can start tasks and run the next if the previous ends I read some articles and chapters about it. It was a success. After that I put some features in it, like a flow controlled UART driver and with an external command I could ask the processor about the CPU usage. This kernel source was not preemive.

4. The task queue and processor time in C++

The previous steps were written in C, but I decided to use the OO aspect for the project from this part. So I wrote nearly the same code in C++, but with task priority. I also wrote the main classis for a buffered UART driver and for the Task Manager. Everything is written to be capable to do a preemptive task changing with priority later.

5. Flash operations and uploaded functions

This part was a main turning-point of the project. Here I realized that writing the flash memory is not that easy. It can be written in every 512 Kb. So here I switched back to C to write these things as simple as possible. So this small program can read the UART, store the input in an SRAM buffer. There are commands to clear the buffer, to copy the buffer to a specified flash memory location, and to run the SRAM or flash memory buffer as a program. I also wrote some small functions which can be uploaded and the main program can run them. Here, the program can run time change an uploaded function.

TaltOS

*A component-based hard real-time embedded operating system with
run-time component change ability.*

*First problems and solutions
Advance and further difficulties*

I. The Bootloader

The first step after getting known the hardware was to write the bootloader of the TaltOS. As it was mentioned above, it had to do the initialization of the peripheries, copy and run the first uploaded function or search the component table for the component TaltOS. It was developed from the *flash operations and uploaded functions* code, so it is written in C. This program uses two buffers which aren't in the stack, but it is allocated statically with the define operator like:
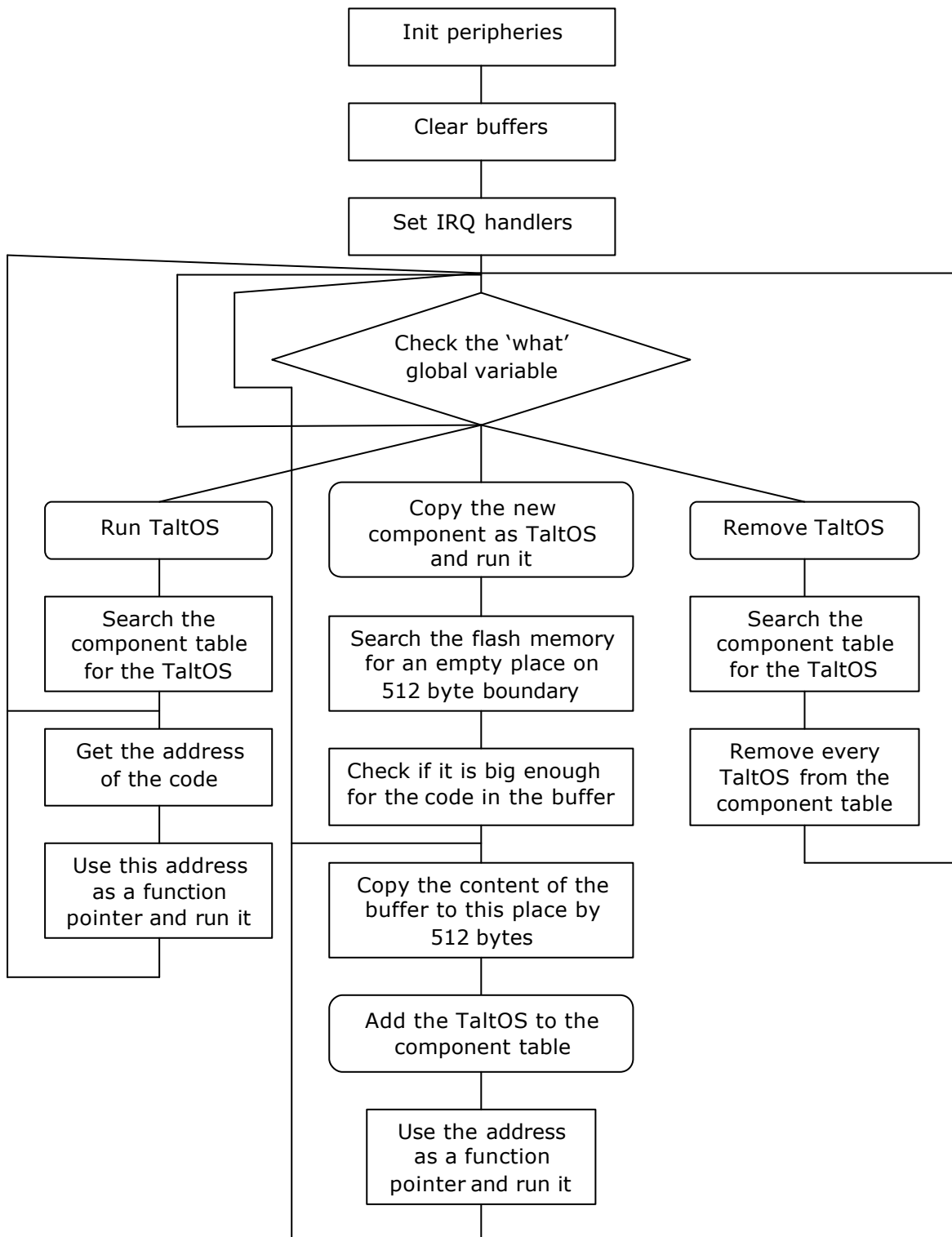
```
#define LPC2106_RAM_TOP_PTR (0x4000FFE0)

//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 (0x1000)              //This is a half of a sector size
#define TALTOS_INPUT_BUFFER_PTR (TALTOS_FLASH_OP_BUFFER_PTR-TALTOS_INPUT_BUFFER_SIZE)
```

The reason will be explained the third chapter which is about the components. The bootloader also uses the component-table to search for the TaltOS or to save the uploaded component as a TaltOS. Here I had to specify the place of the component table and the structure of an entity in it. This part is explained in the second chapter. Anyway, in the bootloader, there are two global variables defined, so that during interrupts they can be modified. One is a buffer state pointer; the other tells the main loop which function it should call. It is controlled by the developer.

So the flow chart of the bootloader is explained in the next figure:

```
                        ┌─────────────────────┐
                        │   Init peripheries  │
                        └─────────────────────┘
                                   │
                        ┌─────────────────────┐
                        │     Clear buffers    │
                        └─────────────────────┘
                                   │
                        ┌─────────────────────┐
                        │   Set IRQ handlers   │
                        └─────────────────────┘
                                   │
                        ╱─────────────────────╲
                       ╱    Check the 'what'    ╲
                       ╲     global variable    ╱
                        ╲─────────────────────╱
```

Init peripheries

Clear buffers

Set IRQ handlers

Check the 'what' global variable

Run TaltOS

Search the component table for the TaltOS

Get the address of the code

Use this address as a function pointer and run it

Copy the new component as TaltOS and run it

Search the flash memory for an empty place on 512 byte boundary

Check if it is big enough for the code in the buffer

Copy the content of the buffer to this place by 512 bytes

Add the TaltOS to the component table

Use the address as a function pointer and run it

Remove TaltOS

Search the component table for the TaltOS

Remove every TaltOS from the component table

And after each character arrived to the input buffer, the buffer pointer is set to the end, and the UART IT handler checks if the last five character is *run* (run the TaltOS) or *cnr* (copy and run the TaltOS) *rmt* (remove TaltOS). The * characters are also included. If one of them matches, the global variable 'what' is set to the specified number and the buffer pointer is set back by 5. So after the IT handler is finished, the main loop will launch the needed routine. After the execution of each command, the input buffer is cleared and the variable 'what' is set to 0. In the future steps, these commands won't be needed, but during the development stage they are necessary.

II. The component table alias TaltOS registry

The TaltOS registry will take place in the flash memory at a specified address (see the TaltOS pre-planning). Because of the fixed size of the full table I tried to figure out a structure which has easily countable size. I need an ID value, a starting address and a size (for later garbage collection). The function pointers of a component will take place at the beginning of its code, so they won't take place at the table. So I chose the structure following:

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

And its place in the memory:

```
//Component table pointer
#define TALTOS_COMPONENT_TABLE_PTR (0x0001C000)//Sector 14 (sector 15 is the remapped bootloader)
#define TALTOS_COMPONENT_TABLE_SIZE (0x2000)//This is a full sector size
```

I decided to have two unsigned long IDs so most of the compiled functions can have unique IDs. The address is of course necessary; the size is needed when we would like to remove the component. With this structure, the 8kb long sector can have 512 registered components. It seems lot, but because of the difficulties of the flash memory writing, it isn't as much.

The flash memory can be written (change bits to 0 from 1) by 512 bytes, and can be erased (filled with 1-s) only by the whole sector. In forums I also found that writing 0-s to 0-s can cause problem (if 0 is overwritten more than about 16 times), so if I rewrite the flash memory, I must write 1-s where I want that 0 and 1 stays as they was, and write 0 where I want to change 1 to 0.

So if a component is removed, the TaltOS will write the binary complement the place of its IDs so they will be 0x00000000 (address and size will be needed for garbage collection). So it won't be freed until the next erase. That's why we need 512 registry entity places. Of course the TaltOS will do this erase (with saving first the non 0 entities) if it is needed, or it is asked to do it. The lifetime of flash memory is mainly depending on the erase operation which can be launched 10.000 times with no damage caused. This is the reason why I decided to use this method to remove components.

So inserting or deleting a component aren't easy tasks. A whole registry size of the RAM has to be filled with 1-s. If it is an insert than the program (bootloader or the TaltOS) searches the first relative empty place in the flash registry, inserts the given values to the relative address to the allocated RAM, and writes back the RAM buffer to the same place of the flash memory. If it is a delete, than the program (bootloader or the TaltOS) searches for the relative addresses of the given IDs, place the complement of these IDs to the relative address to the RAM, and writes back the RAM buffer to the same place of the flash memory.

The source code of these operations are:

```
//Register the TaltOS if possible
address_found=false;
for(i = 0; (i < (TALTOS_FLASH_OP_BUFFER_SIZE/sizeof(TaltOS_reg_entity))) &&
(!address_found); i++)
{
        TaltOS_reg_flash = (TaltOS_reg_entity*)(TALTOS_COMPONENT_TABLE_PTR)+i;
        if ((!address_found)&&(TaltOS_reg_flash->ID1 == EMPTY)&&(TaltOS_reg_flash->ID2 == EMPTY))
        {
                TaltOS_reg_buffer = (TaltOS_reg_entity*)(TALTOS_FLASH_OP_BUFFER_PTR)+i;
                TaltOS_reg_buffer->ID1=TaltOS.ID1;
                TaltOS_reg_buffer->ID2=TaltOS.ID2;
                TaltOS_reg_buffer->address=TaltOS.address;
                TaltOS_reg_buffer->size=TaltOS.size;
                address_found=true;
        }
}
copy_to_flash_512(TALTOS_COMPONENT_TABLE_PTR,(unsigned long*)TALTOS_FLASH_OP_BUFFER_PTR,16);


//Check the component table, if it is a TaltOS, write the complement if it's ID to the buffer
for(i = 0; i < (TALTOS_FLASH_OP_BUFFER_SIZE/sizeof(TaltOS_reg_entity)); i++)
{
        TaltOS_reg_flash = (TaltOS_reg_entity*)(TALTOS_COMPONENT_TABLE_PTR)+i;
        if ((TaltOS_reg_flash->ID1 == TALTOS_ID1)&&(TaltOS_reg_flash->ID2 == TALTOS_ID2))
        {
                TaltOS_reg_buffer = (TaltOS_reg_entity*)(TALTOS_FLASH_OP_BUFFER_PTR)+i;
                TaltOS_reg_buffer->ID1=~TALTOS_ID1;
                TaltOS_reg_buffer->ID2=~TALTOS_ID2;
                // We don't erase the address and size, for later garbage collection
        }
}
copy_to_flash_512(TALTOS_COMPONENT_TABLE_PTR,(unsigned long*)TALTOS_FLASH_OP_BUFFER_PTR,16);
```

III. The TaltOS as component

As the TaltOS is a component as well, I had to start thinking about the buildup of a component. Because of the fact, that the starting address of the code isn't known compile-time, I had to compile with relative addresses. It means that each jump in the code is a relative address to the actual program counter. This can work out if the actual processor core can handle relative jumps (This Philips LPC2106 has ARM 7 core, so there is no problem), and the compiler knows how to build with this option (for a gcc compiler it's the –fPIC option). It is like compiling a shared object, or a dynamic linking library.

Due to the fact that microcontrollers don't have data segment, code segment and stack segment registers, to perform these abilities, I have to figure out how to replace these. The main problem is the working data beside the stack, which are usually the global variables. The TaltOS itself has to have global variables which will be allocated during compile-time with 'define'. Till this time the TaltOS uses an input buffer, an output buffer, a flash operation buffer, some environment variables to handle buffers as FIFOs, and a task list.

```
typedef struct
{
        unsigned short input_buf_start;
        unsigned short input_buf_end;
        unsigned short input_buf_act_size;
        unsigned short output_buf_start;
        unsigned short output_buf_end;
        unsigned short output_buf_act_size;
        TaltOS_task actual_task;
} TaltOS_env;
```

```
// Isn't finished
typedef struct
{
        unsigned long* address;
        unsigned char torun;
        unsigned char priority;
        unsigned char ID;
        unsigned char state;
        unsigned long argument;
} TaltOS_task;

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

 #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))

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

IV. Component functions

The components' main specialty, that they have functions which are to be called by tasks or the TaltOS. If we work with a shared object, it contains symbol information about its function's place in the code. But we must compile as a binary, so the symbols are lost in our output. As it was mentioned in the pre-planning, I have to build a function table during the linking-time so that I can call them by relative addressing.

Solving this problem was another key to achieve my goal, and the solution couldn't be simpler. I only have to modify the linker script I use for linking. If for example I want to make 'public' (i.e. accessible) my function draw() and clear(), I only have to add a new section to my linker script like this:

```
SECTIONS
{
        .functiontable :
        {
                LONG(ABSOLUTE(draw));
                LONG(ABSOLUTE(clear));
        }
        .text :
        {
                *(.text)
        }

}
```

The LONG(ABSOLUTE(symbol)) call in the linker script calculates the absolute address of the symbol and put this long type value to this place. If this component is written in C, the symbol of the function name points to the address where its code begins. Defining this way the function table is really easy. Of course, when a task wants to call a function of a component, it won't know the name of the function. So in the component's specification must contain the number-usage relations like it was specified in the linker script. The first address points to the draw() the second is to the clear().

To help to recover the absolute address of a component's function, the TaltOS will give interface:

```
//Get the component's address by it's ID (Absolute address)
unsigned long* TaltOS_get_component_address(unsigned long comp_id1, unsigned long comp_id2)
{
        unsigned int isfound = 0;
        TaltOS_reg_entity *comp_reg;
        unsigned int i;
        for(i = 0; (i < TALTOS_COMPONENT_TABLE_SIZE/sizeof(TaltOS_reg_entity)) && (!isfound); i++)
        {
                comp_reg = (TaltOS_reg_entity*)TALTOS_COMPONENT_TABLE_PTR+i;
                if ((comp_reg->ID1 == comp_id1) && (comp_reg->ID2 == comp_id2))
                {
                        isfound = 1;
                }
        }
        if (isfound) {return comp_reg->address;}
        else {return 0;}
}

//Get the component's n. function's pointer (Absolute address)
unsigned long TaltOS_get_component_function(unsigned long* address, unsigned long index)
{
        return (unsigned long)(address+(*(address+index))/4);
}
```

The TaltOS_get_component_address returns the absolute starting address of a component (if it exists), the TaltOS_get_component_function needs this address to calculate the absolute address of the function which was given by the index. After this the task only needs to typecast this value to the form he needs to call the function.

The bootloader also does this with the TaltOS:

```
unsigned long* rel_ptr = (unsigned long*)(TaltOS_reg->address);
void (*TaltOS)() =(void(*)())(TaltOS_reg->address+(*rel_ptr)/4);
TaltOS();
```

At this point I've run in a big difficulty. As I tried to use a C++ code as a component I couldn't implement it. The first problem is the symbol naming convention. For a single C function like 'function1' I had to use symbol like '_Z9function1v' the naming convention is much more complicated for an in-class method. I've spent many times to try different compiling options, but with no success. So I decided to go on with the TaltOS in C.

Yes, I know that it was one of the principals mentioned in the pre-planning, but after all I want to write a real-time operating system and the C code is much more predictable in time. And I will have to negotiate the dynamic memory allocation (see chapter five), so the 'new' operator is useless too. As I can imagine that a task wants to call a constructor further problems will appear, like where will be the data part of a class stored. Anyway, the possibility to add a C++ component is still open. And I would like to deal with this topic later.

My tester class was the following (REG8(0xE000C000) = 'A' means to write the A character to the UART0 Transmit holding register):

```cpp
#define REG8(addr) (*(volatile unsigned char *)(addr))

class TempClass
{
        char c;
        public:
        TempClass()
        {
                c='L';
        }
        ~TempClass(){}
        TempClass(char c_in)
        {
                c=c_in;
        }
        void set(char c_in)
        {
                c=c_in;
        }
        char get()
        {
                return c;
        }
        void out()
        {
                REG8(0xE000C000) = c;
        }
};
void function1 ()
{
        REG8(0xE000C000) = '1';
        TempClass tc;
        REG8(0xE000C000) = '2';
        tc.set('X');
        REG8(0xE000C000) = '3';
        tc.out();
        REG8(0xE000C000) = '4';
        tc.set(tc.get()+1);
        tc.out();
}

void function2 ()
{
        REG8(0xE000C000) = 'H';
}
```

As I tested, the result was: 1UIAD which means ARM core aborts (in this order): undefined instruction, software interrupt, prefetch abort and data abort.

V. Task's stack and global variables place

With this compile-time memory allocation, the TaltOS has its own global variables. I haven't implemented the task changing and launching part of the TaltOS, but I've figured out a concept for the future:

The TaltOS will continue to use the stack of the bootloader. The starting address of the stack of other tasks will be placed in the memory, in exact locations. Therefore the number of running tasks will be limited. In an x86 construction, the data is usually before the stack. If a routine wants to start a new task, it have to give the amount of the data needed in bytes to the TaltOS, and the function pointer to call (by ID, index). The TaltOS will pass the data starting pointer to the function. This pointer will be the original stack pointer, and the new stack pointer will be shifted by the amount of the data.

The next figure visualizes this:

1. Task data	1. Task stack	2. Task stack (No data)

Data start
pointer

Stack
pointer
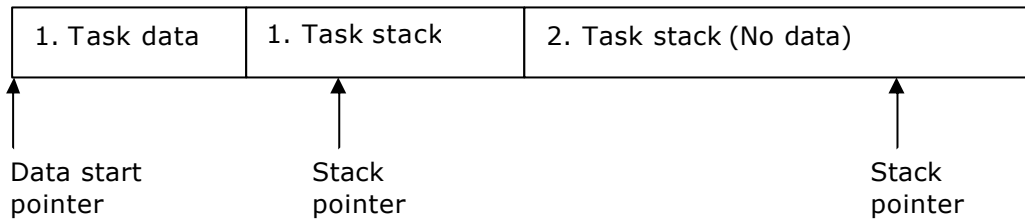
Stack
pointer

With this idea, the stack and the data segment pointer is realizable. But the dynamic memory allocation will be impossible. It is not a big problem, because in real-time embedded systems it is usually not allowed.

VI. The TaltOS v0.1

After these experiences, I compiled the zeroth version of the TaltOS. Yet, it does nothing, but handles two FIFOs, it gives interface for adding and reading from them. After starting, it clears the buffers (FIFOs and a flash operation buffer) and the environment than sets the new IT handler routines to his own. In order to be precise, next to the TaltOS there is an LPC2106 specific code, the TaltOS calls its routines to set the IT handlers. These handlers will call the exact TaltOS functions to read and write the input and output buffers. The strange situation is that to change the IT handlers I need the function pointers. So I had to put out the function pointers of the handlers to the function table, and get these pointers during run-time by calling the above mentioned TaltOS functions.

VII. The sourceforge CVS and design

I really regret that at this point my work isn't accessible to the world. I have launched a project at the Sourceforge CVS but no code has been uploaded to it yet. (There was always problem which had blocked my efforts.) I will also need some design and a homepage to advertise a bit this GPL licensed software. The CVS and design project must run next to my main work.

VIII. Further steps

Component uploading and running

The next step will be to add the ability to the TaltOS to save and run the uploaded components. Here I will have to specify some file format, next to the function table, to verify if it is a component. It will be something like starting with *cmp* and ending with *end*. Maybe this will be realizable with the linker script. If not, I will write a small program to do this, or this can be typed by hand. The TaltOS will start its first function which can add task or initialize some peripheries.

TaltOS Tasks 1

After that I have to write the task handling interface, and use the main loop to change tasks if the last is finished. At first the tasks will run in the stack of the TaltOS and it won't be preemptive.

TaltOS Tasks 2

Here I will try to realize to do the environment change in ARM assembly, and put away his stack pointer to a given address. This will be the point where tasks can have global variables. This part of the project will also be a turning point.

TaltOS further features

The following features will also be part of the project, but there isn't any concept yet how to realize them:

- A component which acts like a bash threw serial port
- An inter task communication feature. I intend to use mailbox protocol.
- The component remover, updater feature.
- The defragmentation of the TaltOS registry and the flash memory feature
- The TaltOS updater with no reboot feature
- Radio Communication feature