ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Paging in TinyOS

Provide dynamic memory and paging capabilities in TinyOS using virtual addressing

Robin Züger

Semester Thesis

Summer Semester 2006

Supervising Professor: Prof. Dr. Roger Wattenhofer
Supervising Assistants: Nicolas Burri, Pascal von Rickenbach

Contents

1 Introduction **2**
1.1 Structure of this document 2
1.2 Memory model of TinyOS 2
1.3 Task description 2

2 Related Work and Background **4**
2.1 TinyAlloc 4
2.2 Memory Allocator 5
2.3 Flash chips 6

3 TinyPaging **8**
3.1 Basic considerations 8
3.2 From a user's perspective 9
3.2.1 Getting started 9
3.2.2 Functionality 10
3.2.3 Policy 11
3.3 Under the skin 12
3.3.1 Data structure 12
3.3.2 Swapping pages in and out 14
3.3.3 Testing 15

4 Futher Work **16**
4.1 Synchronous commands 16
4.2 BlockStorage 16
4.3 Support of other chips and therefore page sizes 16

5 Conclusion **18**

1 Introduction

This report describes *TinyPaging*, a TinyOS component providing dynamic memory to the developper. It works with virtual addresses allowing to swap out parts of the managed memory to the flash and therewith increasing the amount of available storage space - probably beyond what fits in RAM.

The reader is presumed to have a basic knowledge of TinyOS[1] and to be familiar with the basic concepts of memory management, virtual addressing and paging. In order to understand the implementation in detail, knowledge of nesC[2] or C is necessary.

1.1 Structure of this document

This first section introduces the reader to the basics of the memory model of TinyOS. This should allow to understand the subsequent task description. Section 2 presents related work and the background leading to the solution described in section 3. Further development possibilites are outlined in section 4 and section 5 concludes this report.

1.2 Memory model of TinyOS

TinyOS is the de facto standard operating system for programming sensor networks and has been ported to a wide range of hardware platforms. Part of its success lies in its simplicity and therefore efficiency as can be seen when looking at the memory model. In order to cope with the severe hardware constraints of sensor nodes, TinyOS only allows for static memory allocation. This makes it very space and time efficient because there is no need for maintaining an additional data structure managing the dynamic heap. The only necessary data structure is the call stack keeping track of local variables and a few pointers.

This allows to use the entire RAM for storing information. Furthermore, allocating dynamic memory, e.g. using `malloc()` can be very time-consuming if there is almost no space left when a free chunk of memory of appropriate size has to be found. But the downside is that all variables and their size have to be known at compile time which makes working with dynamic data structures, such as linked lists or hash maps, almost impossible. This might sometimes be a too strong restriction, e.g. keeping the ID of an unknown number of neighbouring nodes.

Another problem is the amount of available RAM because some applications may need more memory, probably just temporarily, than the node offers. This is where the EEPROM[3], also called flash, might come in handy. The flash is mostly larger than the RAM but reading from it and writing to it are operations needing a lot of time and energy. So they should be used with caution.

1.3 Task description

This semester thesis aims at solving the two above mentioned problems in combination. There are already components which provide some kind of dynamic memory such as *TinyAlloc*[5] which is described in the next section. Solutions

for using the flash as storage space exist as well such as loggers[1] and the *Memory Allocator*, described in the subsequent section.

Summing up, we already have dynamic memory, but restricted to physical RAM, and we also have access to flash storage, but only in a restricted manner. The component to be developped shall combine the features of these two approaches. The component shall allow the developper to create dynamic data structures that can exceed the available physical memory.

This is done by hiding part of the complexity. The user of the component should not have to bother with moving data from and to flash. All this is done automatically in the background by the component. There is of course a price that has to be paid for this flexibility. Part of the space and time efficiency is inevitably lost, but this should be kept as little as possible by finding an optimal balance between memory consumption and performance.

[1]Loggers are meant for storing log information in the flash. The main operation is `append()` which adds some data at the end. This stored information can be sent to another node, e.g. the sink, at some point in time.

2 Related Work and Background

This section presents related work and gives some background information. Related work includes *TinyAlloc* and *Memory Allocator*. Both of them contributed their part to *TinyPaging*. The section is concluded by an introduction on how a flash chip works.

2.1 TinyAlloc

Basically *TinyAlloc* is a TinyOS component offering dynamic memory capabilities similar to C. It implements an interface named *MemAlloc*[2] whose functionality can be used by any component wiring *TinyAlloc*. *MemAlloc*'s main commands are:

- `allocate(HandlePtr handle, int16_t size)`: returns a pointer `handle` to a newly allocated memory region of `size` bytes
- `free(Handle handle)`: frees the memory region to which `handle` points to and returns it to allocatable memory

TinyAlloc is implemented by using an array serving as the managed heap. Its size can be set by the user of the component and is fixed at runtime. The memory regions are referenced indirectly by a another array containing their current addresses, i.e. the handle returned when calling `allocate` only contains a pointer to this intermediate array. This means that, to access the heap, double dereferencing is necessary. Figure 1 shows an example.

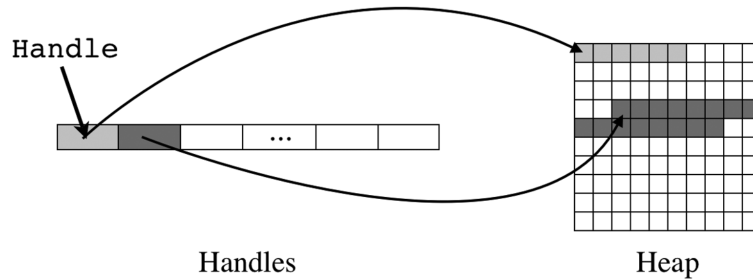

Figure 1: The address of type `Handle` returned to the user only contains a reference to the array of the handles. Dereferencing it twice leads to the current memory location.

Thanks to the double referencing, *TinyAlloc* can move memory regions around freely by altering the addresses in the intermediate array and thus purge the heap which can become fragmented by deallocating memory regions. So this allows to better exploit the available heap memory. It makes it even possible to resize memory regions.

The deterioration of performance due to the double referencing can be overcome by the locking functionality. The latter makes it possible to fix a memory region to the current address, so it can be referenced directly until the region

[2]Actually this interface had been written by the same authors having written *TinyAlloc*.

is unlocked again. This, on the other hand, constricts the relocation of the memory regions.

All commands taking up longer time are implemented in split-phase fashion to not use up too much computing time without interruption[3]. The disadvantage for the user of the component is that he cannot use the memory region right after the `allocate` command but has to wait until the corresponding event is signaled. This makes it slightly more complicated.

Another drawback is that the size of the array containing the handles has a fixed size. This implies that the number of memory regions that can be allocated is restricted. Furthermore, *TinyAlloc* has been developped for processors without memory allignment, this might cause problems on several platforms when dealing with 16 bit integers.

Summing up, it remains to be said that *TinyAlloc* is a good solution when wanting to dynamically allocate large structs, but it is not suited for smaller chunks of memory. The reason for that is the relatively large overhead per allocated memory region: 2 bytes for the entry in the indirection array and 2 bytes for keeping the size[4]. An additional eighth of the heap size is needed to keep track which bytes are used and which ones are still free.

2.2 Memory Allocator

The TinyOS component *Memory Allocator* aims at providing additional capacity for applications requiring more memory than RAM is physically available. To achieve this, it uses some kind of paging algorithm which swaps in and out pages. The developer wiring *Memory Allocator* can define the page size, the memory occupied in RAM and the amount in EEPROM[5] by altering parameters in a config file.

To use this statically allocated space, the *Memory Allocator* provides an interface offering the following major commands:

- `requestMemory()`: returns a handle of type `VMHandle` to a page in RAM, i.e. it allocates a memory region in the size of a page as defined which the user can now freely, to store data in it
- `releaseMemory(VMHandle page)`: releases the page the handle `page` points to, i.e. that memory can be used again when a new page is allocated
- `memoryUse(VMHandle page)`: ensures that the page `page` is loaded into physical memory, i.e. it swaps in the page if it is currently located in EEPROM

The pages are allocated in RAM, but if this space is used up *Memory Allocator* swaps out a page (selected according to a policy) to make allowance for a new page. Since the user does not know which pages are swapped out, he needs to call `memoryUse` everytime before accessing data managed by the *Memory Allocator*. This forces him to wait for the corresponding `memoryUseReady(VMptr`

[3]Some are even split several times.

[4]This is stored in first two bytes of the allocated memory region in the heap itself.

[5]The space in memory and the swap space in EEPROM have to be multiples of the page size.

`Mptr)` event to know at which location in RAM that page currently is (stored in `Mptr`).

Memory Allocator comes with three different policies from which the user can select one in the config file: first-in-first-out FIFO, least-recently-used LRU and least-frequently-used LFU. The data to enable these policies is kept in the background with low overhead and is based on the calls of the `memoryUse` command. In case of the former two policies, namely FIFO and LRU, the developer can be sure that a page remains some time in physical memory. The latter, LFU, forces the user to call `memoryUse` everytime another page has been accessed in between because it could already have been swapped out again.

Although this component allows to make use of the additional storage space in EEPROM, it is only suited for a narrow range of applications due to several reasons: Firstly, *Memory Allocator* requires from the user to call a command and to wait for an event everytime a page is accessed, i.e. there is no locking mechanism forcing the component to keep a certain page in RAM. This is impractical for applications with complex control flow. Secondly, the page in which the desired data is stored has to be known and therefore kept by the user. This allows to store data which is needed complementarily in the same page, but the downside is the additional overhead of selecting the right page.

Apart from all that, it is not finished yet[6]. The two critical parts, loading data from and storing data to secondary storage, are not implemented.

2.3 Flash chips

Working with flash chips is slightly more complex than with RAM because they represent two different types of storage and therefore have different characteristics. Most of today's flash chips are Electrically Erasable Programmable Read-Only Memory, shortly EEPROM[3]. Relating their properties to RAM, this means:

- They are basically read-only, but in contrast to pure ROM's or EPROM's[4] it is possible to electrically erase them. Their advantage is that information is non-volatile.
- Accessing them is significantly slower than accessing RAM, though they are still faster than hard disks in terms of access/seek time.

This means we can use them as secondary storage, but they really have to be treated differently to RAM. Concretely, before writing to the flash chip, the partion to which one intends to write, needs to be erased first. After erasing, writing is possible once. Sadly, there is one more restriction: the size of the partion to be erased is not arbitrary, but dependent on the chip. All chips are divided into a number of pages where one page represents the smallest amount that can be erased. So, this not only limits the size but also gives borders on the regions that can be erased.

A variety of chips with different properties are on the market today and still many of them are fitted to sensor nodes. But only a few offer the possibilites to implement *TinyPaging* for the following reason: They have page sizes of 8kBytes and more, i.e. the smallest unit that can be erased is 8kBytes. Swapping in and

[6] It probably will never be finished, because there has not been any update since 2003.

out pages of this size is absolutely impractical since this is about the size of the entire RAM. Fortunately, the flash chip fitted on the sensor node I am working with, called *TinyNode 584* [7, 8], does not have these limitations. It is fitted with the so-called *AT45DB* [9] having a page size of 264 bytes which makes it ideal for the intended purpose.

Despite that, this EEPROM chip is not as fast as a RAM chip, so we still need to ensure that it is used as rarely as possible. Operations in RAM take normally a few nanoseconds whereas operations involving the EEPROM need several milliseconds.

3 TinyPaging

In this section, you will be presented the actual implementation of *TinyPaging*. There are three main parts: Firstly, you will be given some basic considerations outlining the main concepts. As a second point, *TinyPaging* is described from a user's point of view and finally, we will look under the skin and reveal how it is actually implemented. Please note that each of these three subsections presumes the knowledge of the preceding one.

3.1 Basic considerations

As mentioned in the task description, *TinyPaging* pursues mainly two goals: Providing dynamic memory-like capabilities and increasing the amount of available memory by making use of the flash storage. Thinking back to the two components presented in the previous section, *TinyAlloc* aims at the first goal and *Memory Allocator* provides means for the second. What *TinyPaging* now does is it combines the concepts applied in these two components.

The idea of pages, as it is employed in the *Memory Allocator*, as the unit to be swapped in and out, has been adopted by *TinyPaging*. Remembering what has been explained in section 2.3 on flash chips, each chip has a lower bound that limits the minimal amount of memory being erased at once. So, what we have here is a natural fit and we can exploit this given partitioning of the chip for the swapping, i.e. the size of a swap page matches the page size of the chip. Due to this, we do not need to erase, write and read more than we really want to.

Thinking of dynamic memory from the olden days[7], it basically consisted of two functions: `alloc(size)` and `free(pointer)`. These also are the main commands *TinyPaging* provides, and as *TinyAlloc* supplies them as well: two functions which are simple to use but offer a lot of possibilites to the user.

Another concept applied in *TinyAlloc* is double referencing, i.e. indirect dereferencing involving an additional array which allows to move around blocks within the heap itself. *TinyPaging* does not use this concept, but uses virtual adresses instead. This means when the user allocates a new memory region, he is returned a virtual address. Before using this virtual reference, he needs to call a dereferencing function taking the virtual address and returning the current physical address which can change over time, i.e. when the page on which the region is placed has been swapped out and in again. Using such virtual addressing has advantages over the double referencing approach:

- There is no need for an additional intermediate array which saves a lot of space because this array contains a 16 bit pointer for each memory region, independent from being in use or not.
- There is no artificial upper bound in the number of memory regions that can be allocated as there is when using an intermediate array. The reason for this is simple: The array size is fixed at compile time but there needs to be an entry for each allocated memory region.

[7]By this sarkastic expression, I mean the time before garbage collection came up, when dynamic memory looked the way it is still used in C and C++.

As you might have guessed, there is a shortcoming as well: To figure out the current physical address of a memory region, calling a dereferencing function is necessary. This is not as quick as the double dereferencing, but there is no way to by-pass this because it might be necessary to swap in the page on which the memory region is located first. To overcome this, the user is provided a locking/unlocking functionality allowing to freeze a memory region to a certain physical address. This means he can use the physical address from now on to directly reference the region. This not only makes it easier to access a dynamically allocated chunk but also a lot faster. The number of locks is limited to keep some space for swapping in and out pages and therefore allow access to unlocked memory.

Apart from these functional requirements, there are non-functionals as well: The entire component tries to use as little physical memory as possible but does also take time efficiency into consideration. It has been tried to find a good balance.

3.2 From a user's perspective

This subsection explains the component from a user's point of view, i.e. lists all commands provided by the component and all events that need to be implemented. Apart from a description of the functionality, it also gives some background knowledge to an extent it might help the user to understand how to use it.

3.2.1 Getting started

Please note that *TinyPaging* only works with the *AT45DB* flash chip so far. If your node is fitted with another flash chip, you cannot use it. See section 4 on further work for more information.

The component *TinyPaging* consists of four files / parts:

- `Paging.nc`: contains the interface `Paging` which is provided by *TinyPaging* and has to be used by components wanting to make use of the supplied functionality
- `Paging.h`: contains the declaration of the virtual address type `VAddr` returned by the allocation command and also defines the invalid virtual address `INVADDR`, i.e. instead of `null` or `void`, *TinyPaging* uses its own invalid address
- `TinyPagingM.nc`: contains the module, i.e. the entire implementation of the `Paging` interface
- `TinyPagingC.nc`: contains the component, i.e. the wiring of the different parts of *TinyPaging*

To use *TinyPaging*, wiring the component itself, `TinyPagingC`, is required. The functionality can be accessed by using `interface Paging` in the own module and wiring the interface to the component itself: `MyModuleM.Paging -> TinyPagingC`.

There are a few parameters in the header file `Paging.h` that need to be set beforehand: `PAGES_IN` defines the number of pages that can be swapped

in concurrently, whereas PAGES_TOTAL specifies the overall number of pages. This has to be greater than PAGES_IN but less than 256. When setting these constants, take into consideration that one page offers 232 bytes of memory but uses 266 bytes in RAM[8]. The third parameter, BASE_PAGE, defines the first page on the chip being used for storing the pages in EEPROM, i.e. it uses the pages [BASE_PAGE,BASE_PAGE+PAGES_TOTAL-1]. If you are not using the flash chip for storing other information, 0 is the right choice.

3.2.2 Functionality

As mentioned, *TinyPaging* provides its functionality through the interface `Paging` which arose concurrently to the rest of the component. The offered `command`s are as follows:

- `result_t alloc(uint8_t size)`: Allocates a chunk of memory of `size` bytes. An `allocateComplete` event is signaled as soon as the memory region is ready for use.
- `result_t deref(VAddr addr)`: Dereferences the memory region starting at virtual address `addr`. A `derefComplete` event returns the current physical memory location.
- `result_t free(VAddr addr)`: Frees the memory region that has been allocated at virtual address `addr`. Completion of the operation is signaled by a `freeComplete` event.
- `result_t lock(VAddr addr)`: Locks the memory region allocated at virtual address `addr` to a fixed physical address. This allows to use it without the need of dereferencing virtual addresses. A `lockComplete` event signals the physical address.
- `result_t unlock(VAddr addr)`: This is the counterpart to the `lock` command. It releases a lock and therefore allows the region to be swapped out again. This operation is not split-phase, i.e. there is no according event.

The following `event`s are signaled:

- `void allocComplete(VAddr addr, uint8_t* phy, result_t success)`: Signals a completed `alloc` operation. `addr` contains the virtual address at which the allocated chunk can be found by using `deref`. `phy` contains the current physical address at which it can be used momentarily.
- `void derefComplete(uint8_t* phy, result_t success)`: Returns the current physical address `phy` to the corresponding virtual address asked for by using the `deref` command.
- `void freeComplete(result_t success)`: Signals a completed `free`ing request.
- `void lockComplete(uint8_t* phy, result_t success)`: Such an event is signaled in case of a completed `lock`ing operation. `phy` contains the physical address at which the given virtual address is locked.

[8]These values are *AT45DB* specific. For an explanation see section 3.3.

3.2.3 Policy

TinyPaging applies some principles which, if you know them, you might be able to exploit and therefore make the component perform their actions quicker:

- Pages are swapped in and out according to the least recently used, LRU, principle. A list is kept in the background keeping track of each page access and moves a page at the beginning of the list whenever a new chunk is allocated on it, a memory region is dereferenced or freed or a lock is aquired or released[9]. Whenever swapping out is necessary, i.e. there is no empty slot available in RAM, the least recently used page not helding a lock is selected.

- New memory regions are allocated on the most recently used page having an unused chunk of approriate size, i.e. a chunk at least as large as required. This chunk is selected according to the first-fit principle. If no swapped in page offers a chunk of this size, all pages being swapped out are checked. For this, no swapping in is necessary, unless a suitable page is found, since the size of the largest free chunk is stored. If this fails as well, a new page is allocated.

- If a page contains no more memory regions, i.e. all allocated regions on this page are freed, it will be deallocated.

- Locks apply to an entire page, not just to the passed memory region. Whenever a lock is granted, the entire page stays swapped in until the lock is released. To cope with multiple locks on the same page, but different memory regions, they are additive, i.e. a counter is incremented for each lock aquired and decremented for each lock released. A counter value of 0 means that the page is not locked and therefore can be swapped out. To always be able to swap in and out pages, at least two pages have to be unlocked at any point in time.

- Allocated memory regions are 2 byte aligned, i.e. the starting address of each chunk is even. This makes working with 16 bit integers possible. Even though a `uint8_t *` is returned, this can be cast to `uint16_t *` or any other pointer type: `(uint16_t *)pointer_to_cast`

- Each operation needs to swap out at most one and to swap in at most one page. On the reference system, the *TinyNode 584*, swapping out one page and swapping in one page takes about 32 ms. This means that you can expect each command to signal completion within 32 ms in the worst case, but in the average case it is way faster because the execution time heavily decreases if no swapping is required. You can derive from what you have read in this section when a swapping operation might be necessary, i.e. it also largely depends on `PAGES_IN`.

- *TinyPaging* can perform at most one command concurrently, i.e. it is not possible to post a further request while waiting for the event of the preceding command. A `FAIL` will be returned by the command.

[9] Counting unlocking as a page access is done to maintain consistency. It could as well be omited but it will probably not change much since unlocking is mostly done after using the page.

- When calling a command and SUCCESS is returned, the execution is feasible. This means that *TinyPaging* checks if the execution is feasible, e.g. in case of an allocation request if there is a chunk of appropriate size available or a new page can be allocated, and chooses the return value based on this[10]. In case of FAIL, the user can be sure that his request cannot be executed and does not have to wait for the signaled event. The success parameter of the different events is set to FAIL if something went wrong in the communication with the flash chip, but this is very unlikely.

3.3 Under the skin

This section reveals the implementation details of *TinyPaging*. It explains the data structure on which the component is based. According to the famous guideline "If your datastructures are good enough, the algorithm to manipulate them should be trivial" there should be no need for explaining the algorithms. Additionally, you can also look at the code which is extensively commented. This section is concluded by a short outline on how it has been tested.

3.3.1 Data structure

The main elements of the data structure are two structs: PageStatus and Page. An instance of PageStatus exists for every page, independent from being allocated or not, and from being swapped in or out. These are stored in the allPages array. Its declaration looks as follows:

```
PageStatus allPages[PAGES_TOTAL];
```

PAGES_TOTAL, as you might remember, represents the overall number of pages that can be allocated. The PageStatus struct is defined as follows:

```
typedef struct {
        InPagePtr loc;
        uint8_t free;
} PageStatus;
```

This very simple struct only needs 16 bit for each page. loc stores the current location where this page is swapped in. If it is currently swapped out, its value is set to INVALID[11]. Its type, InPagePtr, is defined as uint8_t and shall emphasize that it points to a location of the inPages array which you will encounter below. free contains the size of the largest free chunk in the page, i.e. we can figure out, just by checking this value, if it is possible to allocate a new memory region of a certain size.

This is a crucial point of the entire data structure, so let me elaborate on it: Each time a chunk is allocated or deallocated, we have to calculate the size of the largest consequtive free memory region of the page. This forward calculation may seem useless because it might never be used, but it has one very big advantage: It allows to balance the execution time of the operations. When looking at *TinyAlloc*, one sees that it always scans the entire array to find a suitable region. Now, let's imagine, *TinyPaging* would do this the same

[10]Note the difference from other components which take the request, return SUCCESS by default and check for feasibility afterwards.

[11]INVALID is a constant with value 255.

way: We had to check a bunch of pages, some of them are even swapped out. Swapping in dozens of pages just to see that there is no space left, can result in a delay of several seconds. By forward calculation on the other hand, which is done when the page is in RAM anyway, we can perform this very quickly and need to do at most one calculation per operation. As you can see, this allows more constant response times, because we have all the values ready to go, when we need them - also for the pages which are currently swapped out.

Now, let's take a look at the data structure keeping track of the pages in RAM. As mentioned at the beginning of this section, this is the second major point. For each available slot, an instance of `Page` is created, which are all stored in the `inPages` array. Its declaration looks as follows:

```
Page inPages[PAGES_IN];
```

`PAGES_IN` is set in the `Paging.h` header file and defines the number of pages that can be swapped in concurrently. The `Page struct` is defined the following way:

```
typedef struct {
        PagePtr page;
        uint8_t locks;
        InPagePtr lessRU;
        InPagePtr moreRU;
        uint8_t data[PAGE_DATA_SIZE];
        uint8_t mask[PAGE_MASK_SIZE+1];
} Page;
```

As you can see, this `struct` is much more extensive. It takes 266 bytes each as it is configured for the *AT45DB* chip. Its first member, `page`, is the counterpart to `loc` of the `PageStatus struct`, as is `PagePtr` to `InPagePtr`. `page` contains the location of the page in the `allPages` array, whereas `loc` links to the current location of the page in the `inPages` array.

`locks` represents the number of locks currently granted for this page. As described in section 3.2.3 on the policy, locks always apply for the entire page. As long as this lock counter is greater than 0, the page remains swapped in at the same location and therefore also keeps its physical address. This allows to directly reference the memory regions it accomodates.

To make allowance for the least recently used policy, it is necessary to keep track of the order in which the pages have been accessed. This only applies for the pages swapped in. `lessRU` and `moreRU` do exactly this: They build a doubly-linked list where `lessRU` contains the page which has been accessed before and `moreRU` stores the page having been accessed more recently. Furthermore, there are two global variables named `leastRU` and `mostRU` storing the head and the tail of the list. In which case they are altered is described in section 3.2.3.

The memory regions returned to the user point to the `data` array, i.e. all the user data is stored in this array. This is exactly the same way as it is done by *TinyAlloc*: A statically allocated array from which the user can dynamically request chunks of memory. Keeping track of the used and free bytes of the array is the `mask`'s purpose. For each byte i in the `data` array, the corresponding bit i of the `mask` is set to 1, if the byte is already part of an allocated memory region. Therefore, the size of the `mask` is defined to be one eighth `data` array size, i.e.

```
PAGE_DATA_SIZE = PAGE_MASK_SIZE << 3;
```

Sadly, keeping the occupied bytes is not enough. Freeing a memory region requires also to know the lengh of it. Fortunatly, due to the 2 byte alignment, we can store this information in the mask as well: To attain such an alignment, it is enough to define all allocated memory region to be of even size[12]. What *TinyPaging* now does, it sets the last bit of the mask belonging to a memory region to 0. Since we know that an odd byte cannot be free if its predecessor is used, this clearly denotes the end of a memory region.

One final remark: In order to provide even addreses to the user, the `data` array itself has to start at an even address. This is not inherently given as it is defined as an array of `uint8_t`. Suming up the `struct` size we get 265 bytes which means that each second `struct` starts at an odd address. To overcome this, the `mask` size is incremented to get to an even number an therefore guarantee even starting addresses. An even starting address of the first struct is achieved by inserting a dummy variable right before of type `uint16_t`.

3.3.2 Swapping pages in and out

The preceding section has covered the data structure used for storing information in RAM. This section describes data structure for storing data in EEPROM.

As you have read in the policy (section 3.2.3), swapping in a page is neccessary when

- either not enough space is left on the swapped in pages to carry out an allocation request, but there is a page currently being swapped out which has enough space left,
- or a derefering, freeing or locking request refers to a chunk of memory currently being swapped out.

In most cases swapping in a page is preceeded by swapping out another page first, because all slots are already occupied by a page[13]. It might happen that a slot is empty in the rare case that all allocated memory regions of a page had been freed and therefore the entire page has been deallocated.

There are two questions that need answering: What and where to swap out? Answering the first one is easy: All that needs to be stored in the EEPROM is the `data` and the `mask`. Any other information either becomes irrelevant, such as the number of locks which are 0 for obvious reason, or remain in RAM for faster access, such as the size of the largest free chunk of the page.

Now, where does *TinyPaging* store this information? All pages have a unique number given by their position in the `allPages` array. And, as you have seen in section 2.3 on flash chips, the EEPROM also has pages which are numbered. So, each page as it is defined by *TinyPaging* is stored on a page of the flash chip. To exploit this natural fit as much as possible, the largest page size has been chosen such that `data` and `mask` still fit in one page of the EEPROM.

The *AT45DB* is partitioned in pages of 264 bytes, i.e. we cannot erase smaller units and therefore not use smaller page sizes. Larger page sizes, if they are multiples of these 264 bytes, would be possible but erasing, writing and reading larger units makes it less responsive. So, the best approximation is 232

[12]When the user requests a chunk of odd size, it is just incremented.

[13]Remember: There are PAGES_IN slots but PAGES_TOTAL pages.

bytes for the `data` and an additional eighth, concretely 29 bytes, for the `mask`. Summed up, this allows to use 261 bytes.

Two comments concerning the storage of the pages in the EEPROM: Since *TinyPaging* might not be the only component of your application that makes use of the flash memory, you can define the first page being used by setting the parameter `BASE_PAGE` in the header file `Paging.h`. See also section 3.2.1 on this. Another thing to notice is that although *TinyPaging* never uses more than `PAGES_TOTAL - PAGES_IN` pages concurrently on the flash chip, it does not exploit this fact but uses `PAGES_TOTAL` overall. This simplification saves memory in RAM and makes it faster which, I thought, is more important than saving space in EEPROM which is rarely fully used.

Let's have a closer look at these *virtual addresses* used by *TinyPaging*. Looking at the type definition, we see that these are unsigned 16 bit integers, `uint16_t`, which allow to address up to 64 kBytes of memory. This is not fully exploited. The virtual address is logically divided into two parts: The first 8 bits determine the page number on which the chunk belonging to this address has been allocated, i.e. they give the position of the page in the `allPages` array. Within this page, the second 8 bits determine the start byte in the `data` array.

Based on this, we can calculate the amount of addressable memory: Theoretically, 256 pages were possible. Due to several reasons, one number has to serve as the *invalid number*, also referened as `INVALID`. 255 pages of which each of them offers 232 bytes of memory results in an overall storage capacity of 59,160 bytes.

3.3.3 Testing

Testing turned out to be quite difficult as the TinyOS simulator *TOSSIM*[10] does not support the flash properly. *TinyPaging* uses a certain part of the TinyOS library which has been specifically designed for the *AT45DB* flash chip. This produces a segmentation fault everytime it is called by the simulator. Thus, I had to test in two stages: As a first step in the simulator (without testing the swapping functionality) and then directly on the node.

For testing in the simulator, I have written a simple application called *PagingTest*. It allows to define a certain sequence of operations that are executed. After each step, apart from the result itself, the current status is printed to the console including the number of allocated and locked pages, the current list to comply with the LRU policy, the mask of the most rececently used page, etc. Since there is no test oracle, the results have to be verified by hand.

Testing on the node itself was tedious since the output options are limited to three leds. To by-pass this limitation, I have written several test cases consisting of a number of operations which test for their correctness and then output the result by setting the leds appropriatly.

4 Futher Work

There are a few ideas in my mind which I could not implement due to time constraints. Some of them are outlined here.

4.1 Synchronous commands

This is for certain the most important point on my list. *TinyPaging* is highly optimized to serve requests in an efficient manner. The execution time of all commands varies little, and although swapping might occur[14], short response times can be guaranteed.

On that basis, the prerequisits are given for using synchronous commands, i.e. commands which are not implemented in a split-phase manner by signaling the results through events but rather answer them directly by their return values. This significantly simplifies the control flow in the application making use of *TinyPaging*. I intended to implement it that way, but sadly, the library used for accessing the flash chip does only provide asynchronous requests. Since rewriting this library would have exceeded the time I could spend on this, I had to implement *TinyPaging* asynchronous.

Rewriting this library (located at `.../tinyos-1.x/tos/lib/Flash/AT45DB`) and then *TinyPaging* to serve requests synchronously, would heavily facilitate its usage.

4.2 BlockStorage

In the next major release of TinyOS, namely version 2, a new flash library called *BlockStorage*[11] is going to be introduced. There also exists a version for TinyOS 1.x. It offers several layers of abstration to support portability for multiple flash chips.

I intended to use it as well but there are still a few restrictions. For me, the biggest obstacle was that it does not allow to specify which page one wants to erase. It only allows to erase the first one which might be suitable for many chips because some of them only allow to erase the entire chip due to their page size. For the *AT45DB* and any other chips that could possibly work with *TinyPaging*, this is impractical.

Altering *BlockStorage* and then adapting *TinyPaging* to it would allow to integrate it even more smoothly when working with other components accessing the flash.

4.3 Support of other chips and therefore page sizes

Up to now, the page size is a fixed value, and the virtual addresses are clearly separated into two parts: One that describes the page number and the other defines the offset within this page. The boundery between these two parts is even stricter defined in the source code.

Supporting other flash chips (and therefore nodes) might be enabled through *BlockStorage*. But there is more to that: Other chips might have other page sizes, so this whole business needs to be altered to add flexibility.

[14]As you have seen in the previous section, at most one page is swapped in and out per command.

Changing the virtual address to 32 might also be a valuable option to support a larger memory space.

5 Conclusion

You have been presented *TinyPaging*, a TinyOS component offering dynamic memory-like capabilities by using paging and virtual addressing. It offers all this with very low overhead in space and is also fairly efficient in terms of time, especially when locking memory regions.

I am quite pleased about the outcome and also the work itself. It allowed me to apply a wide range of skills I have been taught over the past four years such as low-level programming and data structure design. Furthermore, it gave me the possibility to gain an insight into programming sensor networks and embedded systems in general, although I do have to admit that I sometimes missed graphical debugging badly[15].

I would be very happy if someone takes the time and adapts the flash library and *TinyPaging* to handle requests synchronously. That would render it very useful for the programmer, especially for fast prototyping.

Last but not least, I really want to thank my assistants, Nicolas Burri and Pascal von Rickenbach, for supporting me through out the entire work. You have been very helpful and friendly all the time.

[15] Please note that leds, although displaying some color, cannot be considered graphical.

References

[1] Official website of the TinyOS project
http://www.tinyos.net

[2] Official website of the nesC programming language
http://nescc.sourceforge.net

[3] Wikipedia on EEPROM
http://en.wikipedia.org/wiki/EEPROM

[4] Wikipedia on EPROM
http://en.wikipedia.org/wiki/EPROM

[5] The TinyAlloc component is part of the TinyOS distribution and to be found in the directory tinyos-1.x/tos/lib/Util

[6] Memory Allocator project website
http://lecs.cs.ucla.edu/ mosheg/Projects/MemoryAllocator.htm

[7] Official website of the TinyNode platform
http://www.tinynode.com

[8] Fact sheet of the TinyNode 584
http://www.tinynode.com/uploads/media/SH-TN584-103.pdf

[9] Using Atmel's Serial DataFlash
http://www.atmel.com/dyn/resources/prod_documents/doc0842.pdf

[10] TinyOS mote simulator TOSSIM
http://www.cs.berkeley.edu/ pal/research/tossim.html

[11] TinyOS Enhancement Proposal 103 on non-volatile storage and BlockStorage
http://www.tinyos.net/scoop/special/working_group_tinyos_2-0