Nut/OS

Memory Considerations

Version 2.1

Table of Contents

1 Introduction

1.1 Land of Confusion

In the good old days there was magnetic core memory, which later had been replaced by two types of computer memory, RAM (Random Access Memory) and ROM (Read-Only Memory). Actually both, RAM and ROM provide random access. At that time the confusion started.

The author of this document is nearly the same age as digital electronic technology and had been able to closely follow the development of computer memory over the decades. He has no problem to use the term ROM for flash memory, because he experienced the evolution from mask programmable ROM to PROM, EPROM and finally EEPROM (Electrically Erasable Programmable Read-Only Memory), of which flash memory is a special incarnation. Today flash memory is successfully used for memory cards and sticks, which are obviously not read-only.

Will the development of NVRAM bring back the good old days, where one type of memory serves all purposes? Definitely not. And, btw., if we look more closely to the history of computers, we will find out, that there had been many kinds of computer memory all the way.

1.2 Memory Types Used With Nut/OS

The Nut/OS Realtime Operating System and its applications distinguish three main memory types:

- Read-only memory, typically used for program code and constant data
- Volatile memory, typically used for data and stack space
- Non-volatile memory, typically used for configuration data

Depending on the microcontroller architecture and the memory types available on the target board, volatile memory may be used for program code and read-only memory may be used to store configuration data.

At the time of this writing two microcontroller families are supported, Atmel's 8-bit AVR line and the 32-bit ARM architecture, which is available from several vendors.

1.3 AVR Microcontroller Support

The AVR line of microcontrollers use a Harvard architecture, which separates data and program code memory. Currently Nut/OS has been implemented on the ATmega103, the ATmega128 and the ATmega2561, where the following limits apply:

- 128/256k bytes Flash memory, generally read-only, but self-programmable on the ATmega128 and the ATmega2561
- 4/8k bytes volatile on-chip RAM, often named internal SRAM
- 4K bytes non-volatile on-chip EEPROM

Nut/OS has been initially created for the AVR family with embedded Ethernet applications in mind, where data packets can reach a size of 1500 bytes. Thus, typical systems are equipped with additional external RAM of about 32K bytes. However, small applications without networking capabilities will run fine without external RAM. On the other hand, Nut/OS supports RAM sizes beyond the addressing capabilities of the 8-bit AVR by using banked RAM.

1.4 ARM Microcontroller Support

The ARM architecture does not distinguish between program and data memory. However, program code is usually stored in Flash while RAM is used as data memory. In some implementations program code initially copies itself from Flash to RAM, because RAM is typically faster.

As far as addressing capabilities are concerned, the 32-bit ARM architecture is not that limited. However, real world chips like the AT91SAM7X256, which do not provide an external memory bus, are bounded to internal memory of some 100k bytes of Flash memory and some 10k bytes of RAM.

1.5 Programming Language Support

With very few exceptions, where assembly programming is used, the source code is written in C. Thus, the Nut/OS application programming interface (API) is available for C. Limited support for C++ applications is available, but seldom used and not yet well maintained.

As far as memory considerations are related to a specific programming language, this document will mainly explain them to C programmers.

2 Read-Only Memory

In early computer days the term read-only memory (ROM) refers to solid state storage, which contents can't be modified after initially stored in the device. However, today it is commonly used for types of memory, which can't be modified (re-programmed) easily. Nevertheless, it still serves two main purposes: It is non-volatile, which means that the contents is available immediately after power up. And it is cheap when compared to other types of storage devices.

Nowadays, almost all systems use Flash memory, which has several advantages:

- Low cost
- Non-volatile
- Fast read access
- In-system reprogrammable

The disadvantages are

- Memory cells must be erased to become writeable again
- Erasing and writing (reprogramming) is slow
- Erasing can be performed on entire chips or large sectors only
- The total number of allowed erase cycles in a chip's lifetime is limited

2.1 Read-Only Memory Used With Harvard Architectures

The Harvard Architecture allows different word sizes for data and program memory. This is an advantage for small microcontrollers, when they are designed as reduced instruction set computers (RISC). The larger program words required by RISC systems can be combined with small data word sizes to reduce complexity and costs.

The AVR microcontrollers use 16-bit program words and 8-bit data words. The more advanced members of the AVR family provide an external memory bus interface, which is however limited to the data path. Program code is exclusively stored in internal Flash memory. Thus, the memory available for program code and constant data is not expandable by adding external memory devices, even if the chip offers an external memory bus.

2.2 Reprogramming Flash Memory

If the Flash memory is integrated into the microcontroller, it can be usually erased and rewritten without removing any chip from the application board. This capability is called in-system programming (ISP). It can be done by attaching a desktop computer to the target system via a specific programming adapter and running a specific programming utility on the PC, which exchanges control and data informations with the target, using a specific protocol. Today the JTAG protocol is widely used for this purpose.

Another option is available for external Flash memory or microcontrollers, which are self-programmable. In this case a bootloader may be used, which receives updated program code via a standard interface like USB, RS-232 or Ethernet. There are typically two advantages:

- Program updates do not require special hardware like a JTAG adapter
- Bootloaders are often faster than in-system programming

Of course, the program code of the bootloader itself needs to be stored in flash memory and should be protected against unintentional erasure.

2.3 Keeping Constant Data in Flash Memory

By default, the contents explicitly initialized global and static variables is copied from read-only memory to RAM during C runtime initialization. As many embedded systems offer far more Flash memory than RAM, it may make sense to avoid the RAM copy for variables, which are marked constant. The same applies to constant data, which is not assigned to a variable and which can't be directly embedded into the code because of its size. For example, the C instruction

```
i += 8;
```

will allow to embed the contant 8 directly into the code, when the microcontroller supports machine instructions like

```
add reg,#8
```

or

```
ldr r1,#8
add r2,r1
```

In this case no extra storage is required for the constant. However, the constant string used in

```
printf("Hello world!\n");
```

will be stored in an extra memory area and a pointer to this memory location will be passed to the function printf.

For ARM architectures this is no big deal. Actually the developer can decide, whether he wants to keep this string in Flash to save RAM or if he prefers to use a copy in RAM for faster access.

However, when used with Harvard architectures, this provides a real problem. Remember, that data and program memory are strictly separated and a pointer to any kind of data is expected to point into data memory. Luckily the AVR provides special instructions, which allow to read program memory contents. It will be explained later, how this is implemented by the compiler.

2.4 Flash Memory Access Time and Endurance

On the AVR running at clock speeds of about 16 MHz, read access to flash is not slower than reading RAM. However, when using flash memory to store constant data,

then the compiler needs to generate extra code, which adds execution time.

Faster ARM CPUs may reach the access time limits of available flash memory and additional wait states are needed to reduce access cycle times. On the other hand, RAM is available for faster access without wait states.

Note, that on the ARM7 the external memory bus is limited to 16 bit. Running code in external flash memory will either require to switch to 16-bit Thumb mode with its reduced instruction set or requires two 16-bit read accesses per 32 bit word.

Flash memory has unlimited read capability, but can only be erased and written a finite number of times. The flash memory in the AVR microcontrollers has a guaranteed endurance of at least 10,000 erase/write cycles (1,000 for the ATmega103). External flash memory chips like the AT49BV322 used on the Ethernut 3.0 Board allow more than 100,000 erase/write cycles.

3 Volatile Memory

Unless battery backuped, the contents of RAM is unspecified after powerup.

The 64K byte address space is divided in several parts and slightly differs between the ATmega103 and the ATmega128. The following table shows the data memory layout of the Ethernut Board 1.3-Rev-D with the ATmega128 CPU.

Previous Ethernut Boards with ATmega103 CPU didn't get extended I/O registers, but use this area for internal SRAM, which ends already at 0x0FFF. With both versions the external RAM chip occupies address range 0x0000 - 0x7FFF, but the ATmega103 or ATmega128 CPU activate external addressing starting at address 0x1000 or 0x1100 resp. Thus the lower external SRAM space is wasted.

3.1 Variable Storage

For initialization purposes, the C compiler determines three types of variables:

1.Auto variables, which are defined within functions and are not declared static.

2.Static and global variables with an initial value of zero, located in the .data segment.

3.Static and global variables with an initial value not zero, located in the .bss segment.

Auto variables are placed in the stack area. Any initial value will be assigned after the program entered the function they are declared in.

Static and global variables are initialized to zero by default, if not otherwise specified. These variables are grouped into a single RAM area called the .data segment, which is cleared to zero during initialization. Static and global variables with initial values other than zero are grouped together in a second RAM area called .bss segment, of which a mirror exists in flash ROM. During initialization the flash ROM mirror is copied into the RAM area.

The .data segment is placed at the beginning of the RAM area, followed by the .bss segment.

Up to now, Nut/OS applications are linked for on-chip RAM only. The linker will not be informed of external RAM. It's exclusively used by the Nut/OS heap. This may change in future releases. Right now this limits existing applications to 4K byte variable space. This is no big problem, because large memory areas like arrays and structures can be allocated dynamically from the Nut/OS heap.

4 Non-Volatile Memory

Actually the read-only memory described in chapter 2 is a special type of non-volatile memory. However, Nut/OS distinguishes between non-volatile memory for program code and constant data, where the contents will typically not change during runtime, and non-volatile memory for configuration data, where the contents may change more or less often during runtime and needs to be preserved during periods without power supply. The latter will be handled in this chapter.

Electrically erasable programmable read only memory (EEPROM) is a non-volatile memory, which preserves its contents when power supply is removed. Therefore it's an ideal memory space used by Nut/OS and many applications to hold configuration information.

The ATmega128 contains 4K bytes of data EEPROM memory. It is organized as a separate data space and accessed through two registers, the EEPROM address register and the EEPROM data. Single bytes can be read and written.

4.1 EEPROM Access Time and Endurance

EEPROM read and write access is very slow. Its use should be limited to reading configuration data during system startup, which is seldom modified.

Although limited, the EEPROM has an endurance of at least 100,000 write/erase cycles, which is about the same as modern flash memory endurance.

5 Nut/OS Program Memory

5.1 AVR Program Code

The following steps are used to create the program code and write it into the flash ROM:

1.Creating one or more text files, which contain the application source code.

2.Compiling the source code creates one object file per source code file.

3.Linking all created object files with Nut/OS libraries to create a binary file.

4.Converting the binary file to a hex file.

5.Uploading the hex file contents to the target's flash ROM using an ISP or JTAG adapter.

After reset, the CPU starts execution at address zero by default. This address contains a jump instruction, passing control to the runtime initialization of the C library. When this initialization is done, the library jumps to the main entry of the application code. With Nut/OS, this entry is part of the init module, where the Nut/OS initialization takes place. This will setup memory, timer and thread management of the RTOS and finally start the main application thread, called NutMain.

When using ICCAVR with its IDE, the init.c source file of Nut/OS has to be included into the project. However, it must never be modified. If modification is required, one should make a local copy, which replaces the original Nut/OS code.

An often asked question is, if any possibility exists to extend program code space using external memory. The simple answer is, that this is not possible. However, the self-programming feature of the ATmega128 combined with serial memory devices may offer a solution for specific applications. Another attempt is, to use an interpreter, which reads the code from external data memory. Neither Nut/OS nor the compilers support such an environment.

6 Nut/OS Data Memory

6.1 Global Variables

In general, global variables are stored in RAM.

6.2 Local Variables

Local variables may be either auto or static variables.

6.3 Constant Variables

Variables, which contents will not change.

6.4 Heap Memory

Dynamic memory allocations are made from the heap. The heap is a global resource containing all of the free memory in the system. The C runtime library of both compilers offer their own heap management, but this is currently not used by Nut/OS.

Nut/OS handles the heap as a linked list of unused blocks of memory, the so called free-list. The heap manager uses best fit, address ordered algorithm to keep the free-list as unfragmented as possible. This strategy is intended to ensure that more useful allocations can be made and ends up with relatively few large free blocks rather than lots of small ones.

Nut/OS enables external RAM by default and occupies all memory space beyond the end of internal RAM up to address 0x7FFF as heap memory. If more than 384 bytes of internal RAM are left between the end of the .bss segment and the end of internal RAM, this area minus 256 bytes is added to the heap too. The 256 bytes on top of the internal RAM are left for the idle thread's stack. Obviously, the idle thread uses much less stack space, but interrupt routines will use it when interrupting the idle thread.

Enabling external RAM access in module init.c will also switch off PORTA and PORTC functionality. Nut/OS runs well without external RAM, when no or very limited network functions are used. This requires to modify init.c. It is recommended to make a local copy for modification, which replaces the original Nut/OS code.

6.5 Banked Memory Support

AVR microcontrollers can do a remarkable amount of work, but sometimes the 64k bytes address space just isn't enough. Several suggestions have been posted to the Ethernut mailing list about how to add more RAM. Even replacing all remaining external address space of about 60k wouldn't help much with new upcoming

applications. Finally bank-switched RAM was the selected solution, sometimes also referred to as mapped memory, as it maps a large address space into a smaller address window.

A CPLD was used for the hardware implementation, which includes logic for address decoding and a bank select register. Some support is provided by the Nut/OS API to manage bank selection and hide this hardware specific part from the rest of the system.

7 Nut/OS Stack Usage

7.1 Program Stack

The C runtime library initializes the stack, starting from the top of internal RAM and growing downwards. This stack is used by the Nut/OS idle thread. As each thread requires its own stack space, Nut/OS dynamically allocates the requested size from heap memory when the thread is created. While switching from one thread to another, Nut/OS saves all CPU registers of the currently running thread and restores the previously saved register contents of the thread being started, including the stack pointer.

The memory area used for the stack is allocated by NutThreadCreate(), which is unable to determine how much stack space may be needed by thread. Therefore this size is passed as an argument and must be specified by the caller, actually the programmer. But how can the programmer know?

Stack space is used for two purposes: Register storage during function calls and storage of auto variables. The stack space used for register storage is decided by the compiler and is hard to foresee. It depends on the optimization level, the register usage before, after and within the function call.

Nut/OS API functions called by the application may call other functions as well. In addition, interrupt routines are using the stack space of the interrupted thread and need to store all CPU registers.

Putting this all together, it will become clear, that it is at least difficult, if not impossible to determine the required stack space, due to the asynchronous nature of thread switching. Typically a maximum is estimated and some space are added for safety.

To avoid wasting stack space, the application should...

- ...not execute recursive function calls, unless a maximum nesting level is guaranteed.
- ...not declare large non-static arrays or structures within functions. They should be either declared global or, to retain reentrency, declared as a pointer and allocated from heap memory.

Most application threads will be satisfied with 512 or even 256 bytes of stack. If enough memory is available, you should oversize the stack during development and reduce it later during final testing.

8 Nut/OS Configuration Storage

To be done.

9 I/O Mapping

Some microcontrollers like the Intel 80x86 provide a dedicated I/O bus. This is called port-mapped I/O and requires special CPU instructions to access I/O devices. All CPUs supported by Nut/OS so far provide memory mapped I/O, which means that the same memory bus is used for memory and I/O devices.

In order not to rule out microcontrollers with port-mapped I/O, Nut/OS generally uses special I/O functions, which can be easily implemented on targets using memory-mapped I/O. An additional advantage of using special I/O functions is, that I/O access can be more easily distinguished from memory access, which is most useful for hardware emulation.

10 Memory Mapped File Systems

10.1 UROM File System

Simple read-only file system, located in program memory.

10.2 PNUT File System

Simple file system, which is located in volatile memory. Contents will be lost when restarting the system.

10.3 Tiny File System

Not yet implemented, but simple enough to be usable with 8-bit CPUs.

11 Using a Bootloader

11.1 Bootloading Nut/OS

The Nut/OS distribution includes several sample bootloaders, which make use of the standard protocols BOOTP and TFTP:

- eboot for bootloading the AVR ATmega128/2561 internal Flash memory via RTL8019AS Ethernet Controller
- appload for bootloading the AVR ATmega128 internal flash memory via LAN91C111 Ethernet Controller
- bootmon for bootloading the AT91R40008 internal RAM via DM9000A Ethernet Controller

Some microcontrollers provide factory programmed bootloaders, like SAM-BA for the AT91SAM series, which allows bootloading Nut/OS applications from RS-232 or USB.

11.2 AVR Bootloader

The ATmega128 and its successor the ATmega256 are able to self-program their own flash ROM. As an alternative to the ISP or JTAG adapter, a boot loader may be uploaded once. This bootloader can then use a different standard interfaces like RS232 or Ethernet to receive the application code and use the self-programming feature to write the code into flash ROM.

The flash ROM is divided into two sections, the bootloader section and the application program section. In addition, there are various configurations available with the ATmega128/256 to execute the bootloader code after reset or redirect interrupts to the boot section. Please refer to the related datasheets for further explanations.

12 Compiler Related Considerations

12.1 GCC for AVR

The GNU compiler offers the ability to add attributes to variables. This feature is used by the AVR version of the compiler to implement program memory constants. The attribute progmem forces a variable to reside in ROM. Still the compiler faces the same problem in case of pointers passed as function arguments.

Last not least, many duplicate API functions exist in Nut/OS to support pointers to constants in program memory.

12.2 GCC for ARM

To be done.

12.3 ImageCraft C Compiler for AVR

12.3.1 ImageCraft AVR Runtime Initialization

The standard runtime initialization provided by the ImageCraft Compiler can't be used with Nut/OS. Instead, slightly modified startup files are included in the Nut/OS distribution. Check the Nut/OS Software Manual for further information.

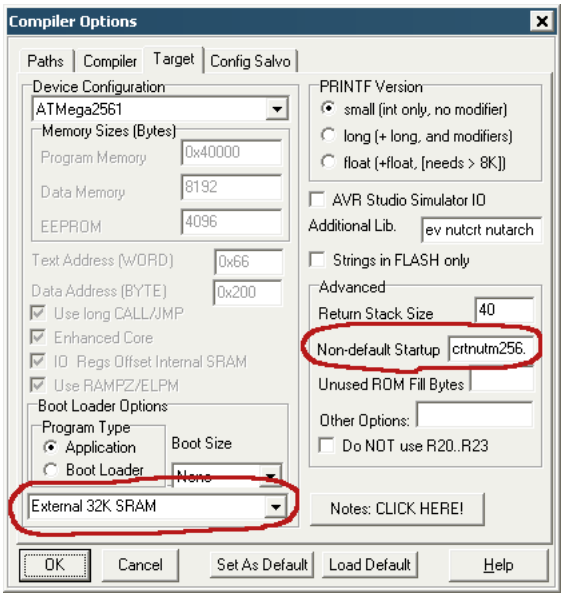

Figure 1: ICCAVR startup settings

When using the ImageCraft IDE, the correct startup file must be specified in the compiler settings. Also make sure to enable external RAM, if available.

12.3.2 ImageCraft AVR Stack Layout

ICCAVR uses two stacks. A hardware stack is used to store the return address on subroutine calls and a separate software stack is used for auto variables and parameter passing. In case of interrupts, the hardware stack is used to save CPU register contents, which includes the return address. CPU register SP points to the hardware stack and CPU register Y points to the software stack.

In almost all cases the hardware stack will need much less memory space than the software stack. Thus, it will easily fit in internal RAM, which is often faster than external RAM. Unfortunately, the current Nut/OS port for the ImageCraft compiler doesn't make use of this. Instead the total stack space for each thread is allocated from heap memory and the software stack pointer is simply set 40 bytes below the initial hardware stack. Remember, that the AVR stack grows downwards. There is no way to modify the size of the hardware stack and the related setting in the ImageCraft IDE is simply ignored.

Nut/OS uses the hardware stack to store CPU registers during context switch. The related SWITCHFRAME structure is already placed above the current hardware stack pointer when the thread is created. The ENTERFRAME structure is used only for the first time entry into the thread function. Thus, the size of 40 bytes for the hardware stack will actually be increased by the size of the ENTERFRAME structure.

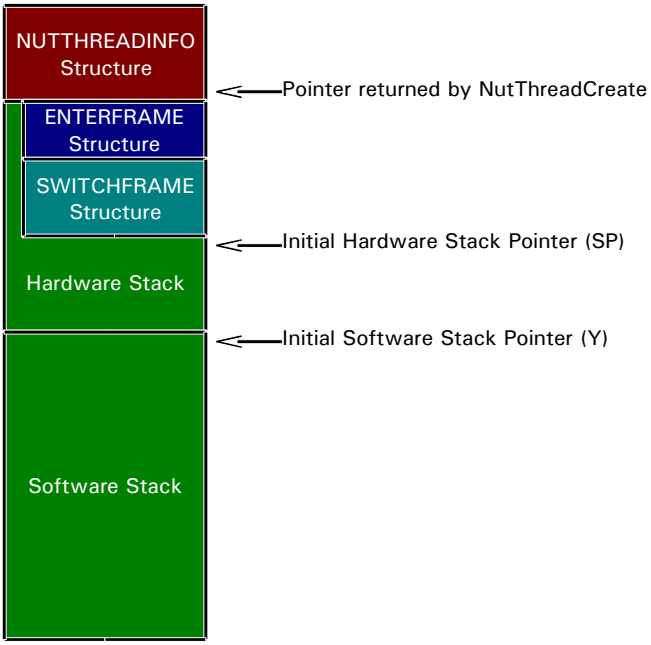

Figure 2: ICCAVR stack, thread is created

Figure 2 shows the initial stack layout after the thread has been created. A switch frame structure and an enter frame structure are already available on the hardware stack.

When the thread starts running for the first time, the SWITCHFRAME structure and the ENTERFRAME structure are pulled from the hardware stack. The final stack layout when entering a new thread is shown in Figure 3.

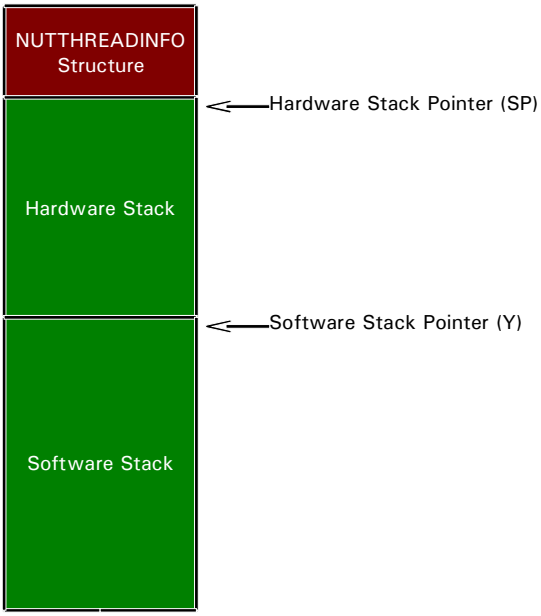

Figure 3: ICCAVR stack, thread started

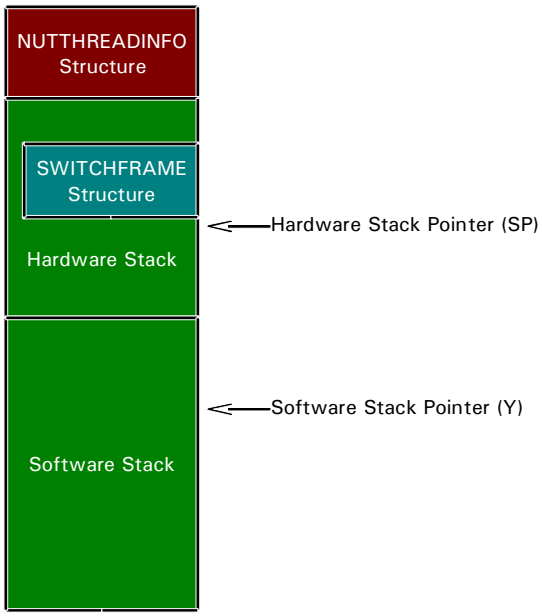

Figure 4: ICCAVR stack, thread is waiting

When the thread is stopped, the context switcher will push a new SWITCHFRAME structure containing the thread's context on the hardware stack (see Figure 4).

The 40 bytes plus the size of the ENTERFRAME structure of 9 bytes (10 byte for the ATmega2561) should be sufficient for most applications.

13 Memory Layout Showcase

This chapter discusses the memory layout of several target boards, which are known to successfully run Nut/OS applications.

13.1 Ethernut 1

The Ethernut 1 Board can be equipped either with an ATmega128 or an ATmega2561 CPU, while early version used an ATmega103 CPU.

A 32k Byte external RAM provides sufficient data space for small to medium network applications.

The 10 Mbit Ethernet interface is implemented by an RTL8019AS. This chip provides a full internal address decoder and occupies only 32 byte address space.

13.1.1 Ethernut 1 with ATmega103

Note: The ATmega103 is no longer in production.

Only 28 kBytes of the 32 kByte external RAM are used, because the lower 4k is overlapped by internal registers and RAM.

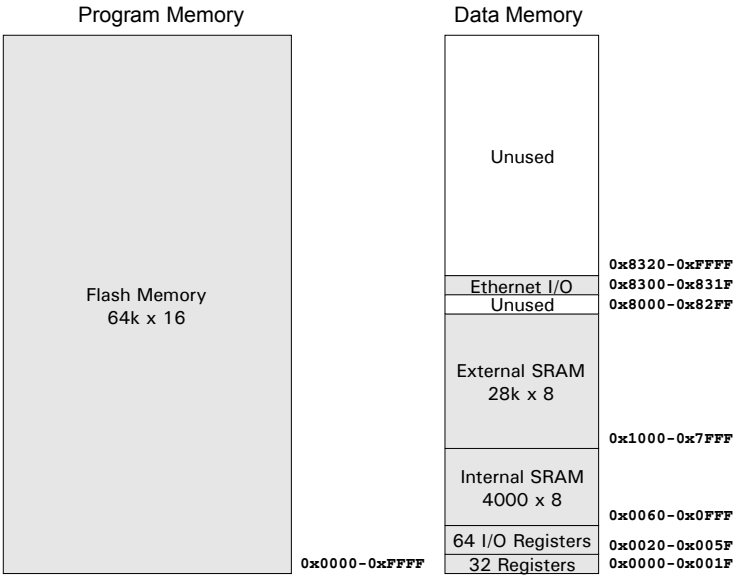

Figure 5: Memory Layout of Ethernut 1 with ATmega103

13.1.2 Ethernut 1 with ATmega128

The ATmega128 is a direct replacement for the ATmega103. It can even run in ATmega103 mode.

When used in native mode, an additional 160 I/O registers and and additional 96 Bytes of internal RAM are provided. Furthermore, the CPU can self-program its flash

memory for bootloading support. An optional boot section can be configured, at sizes of 0.5k, 1k, 2k or 4k words.

The lower 4352 Bytes of the external RAM are not directly usable, because they are overlapped by internal CPU registers and internal RAM. Consult the ATmega128 datasheet for how to access this hidden RAM area.

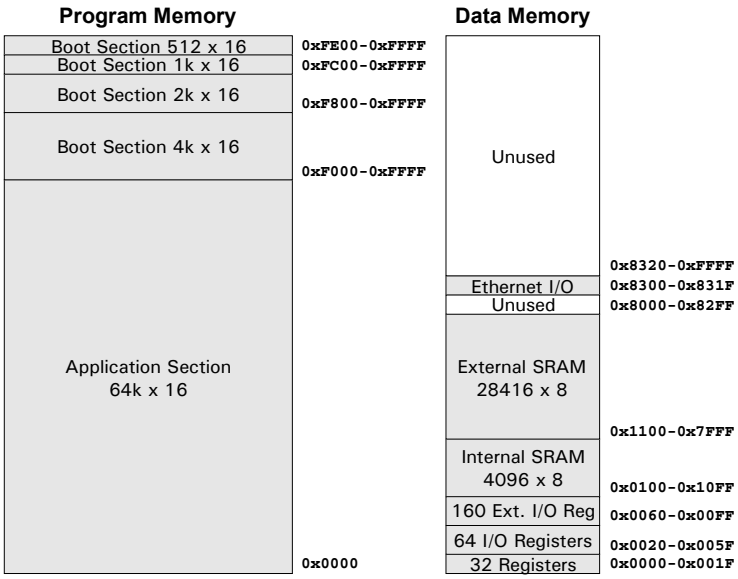

Figure 6: Memory Layout of Ethernut 1 with ATmega128

13.1.3 Ethernut 1 with Atmega2561

The ATmega2561 is the latest member of the AVR family, which is pin compatible to the early ATmega103. Though, it can no longer run in ATmega103 mode, but provides twice the internal RAM and flash memory size of its predecessor, the Atmega128. However, only the upper 23.5 kBytes of the external RAM are directly usable. Check the ATmega2561 datasheet for how to access this hidden RAM.

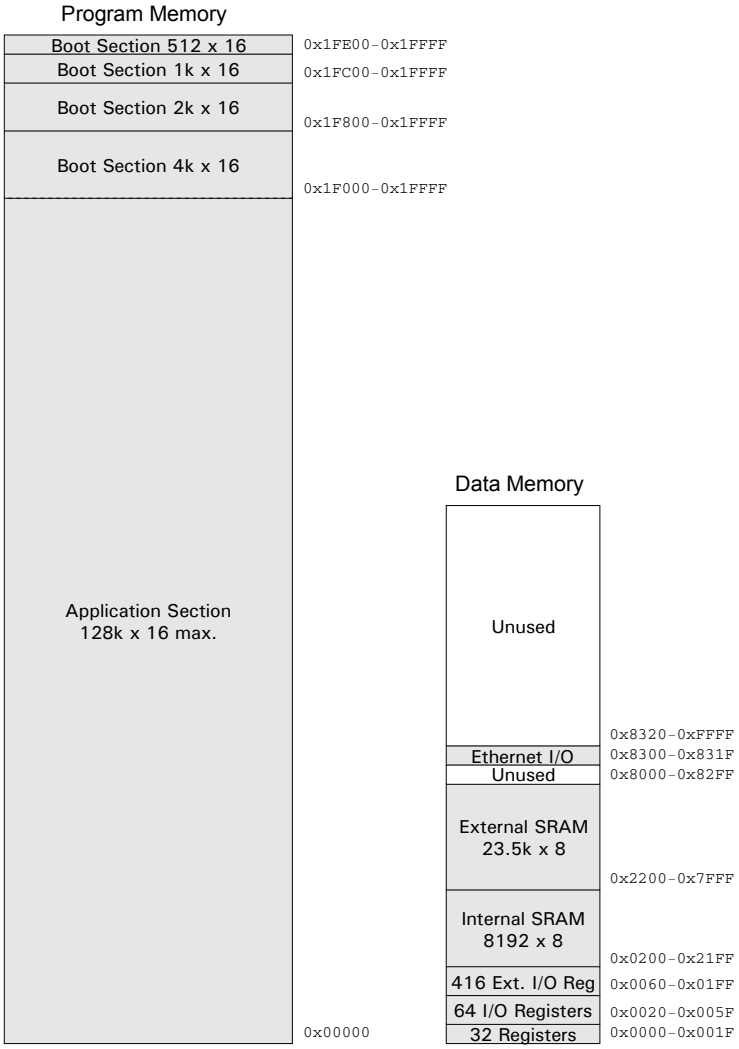

Figure 7: Memory Layout of Ethernut 1 with ATmega2561

13.2 Ethernut 2

Like Ethernut 1, the Ethernut 2 Board can be equipped either with an ATmega128 or an ATmega2561 CPU, but provides 512 kBytes of RAM, 512 kBytes of serial flash and a 10/100 Mbit Ethernet interface. The latter uses a LAN91C111 Ethernet controller.

Due to the limited data address space of the AVR, only 48k of the 512k RAM are visible to the CPU. The lower 32k, which are partly overlapped by internal registers and RAM, are fixed, while any of the 30 banks can be made visible in the upper 16k RAM area. Although banked memory is not supported by the compiler, Nut/OS offers an API that allows to use a 480 kByte streaming buffer.

13.2.1 Ethernut 2 with Atmega128

The memory layout is shown in figure 8.

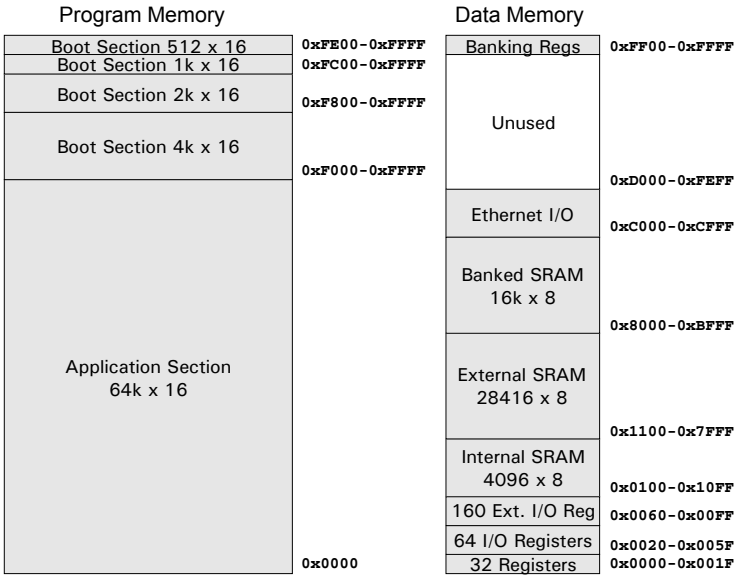

Figure 8: Memory Layout of Ethernut 2 with ATmega128

13.3 Ethernut 2 with Atmega2561

The memory layout is shown in figure 9.

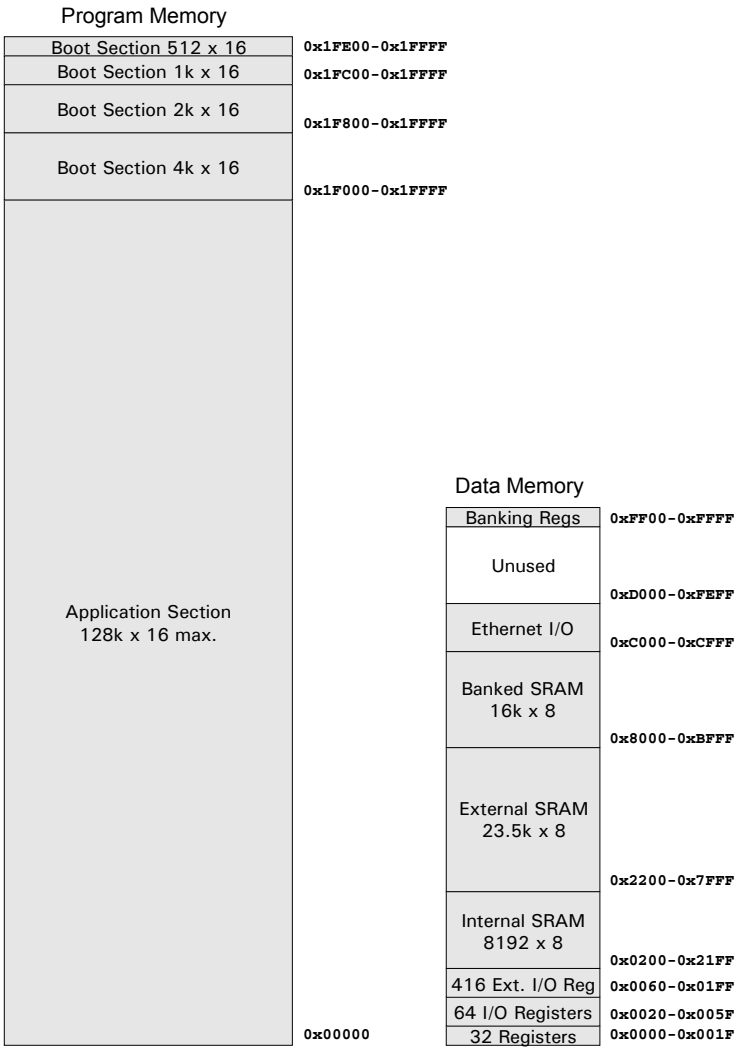

Figure 9: Memory Layout of Ethernut 2 with ATmega2561

13.4 Ethernut 3 with AT91R40008

To be done.

13.5 Nintendo Gameboy Advance

To be done.

13.6 AT91SAM7X256-EK

To be done.

13.7 AT91SAM9260-EK

To be done.

Bibliography

Alphabetical Index