Nut/OS Memory Considerations

Version 1.0

Contents

1 Introduction

Managing a scarce resource.

This document is far from being completed now.

The Nut/OS Realtime Operating System and its applications make use of three different types of memory:

- Flash ROM used for program code and constant data
- RAM used for variables and stack
- EEPROM used for configuration data

The AVR line of microcontrollers are using a Harvard architecture, which separates data and instruction code memory. All known implementations are running on ATmega128/103 CPUs, where the following limits apply:

- 128K bytes Flash ROM, self-programmable on the ATmega128
- 4K bytes on-chip static RAM, often named internal SRAM
- 4K bytes on-chip EEPROM

Nut/OS has been created with embedded Ethernet applications in mind, thus typical systems are equipped with additional external RAM of about 32K bytes. This document also presents the first proposal for implementing bank-switched RAM, which will be supported in the near future.

2 Flash ROM

Memory for program code and constants

Flash read only memory (Flash ROM) is a non-volatile memory, which preserves its contents when power supply is removed. Both, the ISP and the JTAG feature allow to erase and write the content of the flash ROM without removing the CPU from the application board.

All program code must reside in flash ROM. The ATmega program counter is 16 bits wide and can address 64K words or 128K bytes of program memory.

Note, that flash ROM must be erased before rewriting. Flash ROM is organized in sectors. Erasing and writing is done for an entire sector at a time, while reading is possible on a byte basis.

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.

Boot Loader

The ATmega128 is able to self-program its own flash ROM. As an alternative to the ISP or JTAG adapter, a boot loader may be uploaded once. This bootload can then use a different interface like RS232 or Ethernet to receive the application code and use this 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 to execute the bootloader code after reset or redirect interrupts to the boot section. Please refer to the ATmega128 datasheet for further explanations.

Micro ROM File System

To be done.

Program Memory Constants

All AVR instruction codes are two or four bytes long. When executing code, the CPU addresses a word and thus can access the entire flash ROM with its 16 bit program counter.

Beside program code, constant data can be located within the entire flash ROM. However, reading a single byte of data from a 128K byte address space requires 17 address bit, while the instruction set supports a maximum of 16 bits for data addressing. The ATmega128 and 103 use the RAMPZ bit of the page select register to indicate usage of the lower or upper 64K byte flash ROM section.

Placing constant data in flash ROM has several important advantages:

1. RAM space is saved, which is typically more scarce than flash ROM.

2. Constant data in flash ROM can't become easily corrupted.

3. No need for the C runtime library to initialize RAM variables.

Point 2 is very important in these tiny hardware systems, where Nut/OS is running, because they do not provide any kind of memory protection. A single thread is able to destroy the entire RAM and may make the whole system crash.

Point 3, variable initialization, is further explained in the next chapter.

Both compilers, ICCAVR and AVR-GCC, which are used with Nut/OS, support constants in program memory. But they use different approaches.

ICCAVR uses the const keyword to specify variables located in ROM. Upto a certain extend, these variables can be used like normal variables. Of course they are read-only, but that's the definition of a constant anyway. The problem appears when using these variables as function parameters, because the function is not able to determine, whether a parameter passed as a pointer contains an address into RAM or ROM space. This is why a bunch functions exist in two flavours, one for pointers pointing into RAM and another for pointers into ROM space.

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.

Access Time and Endurance

In general, read access to flash ROM is not slower than reading RAM. However, the compiler needs to generate extra code for ROM access, which adds execution time.

One aspect should not be overlooked. Flash ROM has unlimited read capability, but can only be erased and written a finite number of times. The flash ROM in the AVR microcontrollers has a guaranteed endurance of at least 1,000 write/erase cycles. In the real world, this number may be ten or more times higher.

3 RAM

The data space.

Unless battery backuped, the contents of random access memory (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.

Address Range	Used by
0x0000 - 0x001F	32 CPU registers
0x0020 - 0x005F	64 I/O registers
0x0060 - 0x00FF	160 extended I/O registers
0x0100 - 0x10FF	On-chip internal SRAM
0x1100 - 0x7FFF	External SRAM
0x8000 - 0x82FF	Unused, available for extension hardware
0x8300 - 0x831F	RTL8019AS registers
0x8400 - 0xFFFF	Unused, available for extension hardware

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.

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.

Upto 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.

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 upto 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.

Program Stack

The C runtime library initializes the stack starting from the top of internal RAM, 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 determing the required stack space is at least difficult, if not even impossible because of the asynchronous nature of thread switching. Typically a maximum is estimated and some bytes are added for safety.

Nut/OS allocates 768 bytes of stack for the main thread, which is far more than most applications will use, if they follow certain rules. To avoid wasting stack space, the application should...

1. ...not execute recursive function calls, unless a maximum nesting level is guaranteed.

2. ...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. Call NutHeapAvailable() to determine the number of bytes available.

Advanced users of Nut/OS may inspect the NUTTHREADINFO structure to track the stack pointer.

External Bank-Switched RAM

AVR microcontrollers can do a remarkable amount of work, but sometimes the 32K bytes RAM just aren'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. The solution would be a bank-switched RAM, sometimes also referred to as mapped memory, as it maps a large address space into a smaller address window.

The following text presents a proposal, open for discussion. It's intended as a request for comments, which should be posted to the Ethernut mailing list.

The following table gives an example, how memory layout looks like with bank switched RAM space.

Address Range	Used by
0x0000 - 0x10FF	Internal data memory
0x1100 - 0x7FFF	32K Bytes unswitched RAM
0x8000 - 0xBFFF	Upto 256 switched RAM banks, 16K bytes each
0xC000 - 0xFEFF	Reserved for memory mapped I/O
0xFF00 - 0xFFFF	Bank select register

For the first hardware implementation it is planned to use discrete logic for address decoding and the bank select register. These discrete components may be later replaced by a programmable logic device.

The software support would be quite limited, because all pointers are still 16-bit. Using C++ style methods to access items in banked memory would significantly reduce the performance. Memory banks are expected to be used as buffer space, which typically requires extensive access by code loops walking through this area. To keep these loops as tight as possible, the application code should have direct access.

However, some support should be provided by Nut/OS to manage bank selection and hide this hardware specific part from the rest of the system.

One part of the RTOS, which becomes involved in bank selection is probably thread switching. This ensures, that a previously accessed memory bank is still available after control returns to a previously stopped thread.

The same scheme applies to interrupts. As all interrupts are registered with and routed through Nut/OS, banks can be switched without changing existing drivers.

In addition a number of core routines will be available to provide some basic tasks:

- Return memory start address and size of the bank switched area.
- Allocate and release single banks and return the number of banks available.
- Select a specified bank and query the currently selected one.

In these core routines, banks are simply specified by their number. Bank 0 is selected by default and may be used as normal data memory for applications, which do not use bank switching. All core routines can be used within interrupts.

API Call	Function Description	Parameters
NutBank()	Returns the first memory address of the bank switched area	None
NutBankSize()	Returns the size of the bank switched area in bytes	None
NutBankAvailable()	Returns the number of available RAM banks. This number doesn't include bank 0	None
NutBankAllocate()	Exclusively allocates a RAM bank and returns its number or 0 if no banks available	None
NutBankFree()	Releases a previously allocated RAM bank	Number of the bank to be released
NutBankShow()	Makes a bank accessible in the memory window	Number of the bank to access, 0 included
NutBankVisible()	Returns the number of the bank currently accessible	Number of the bank to access

These core routines can be used as a basis for more advanced routines, which may be implemented when required.

4 EEPROM

Configuration memory.

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.

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.

Allthough limited, the EEPROM has an endurance of at least 100,000 write/erase cycles, which is much higher than flash ROM endurance.

5 Links

Where to find additional information.

http://savannah.nongnu.org/projects/avr-libc/

AVR C Runtime Library.

http://www.ethernut.de/

Information about the Ethernut board.

http://www.egnite.de/

Home of egnite Software GmbH, the developer of Nut/OS and the Ethernut hardware reference design.

6 Index