NanoRK

EECE 494

Sathish Gopalakrishnan

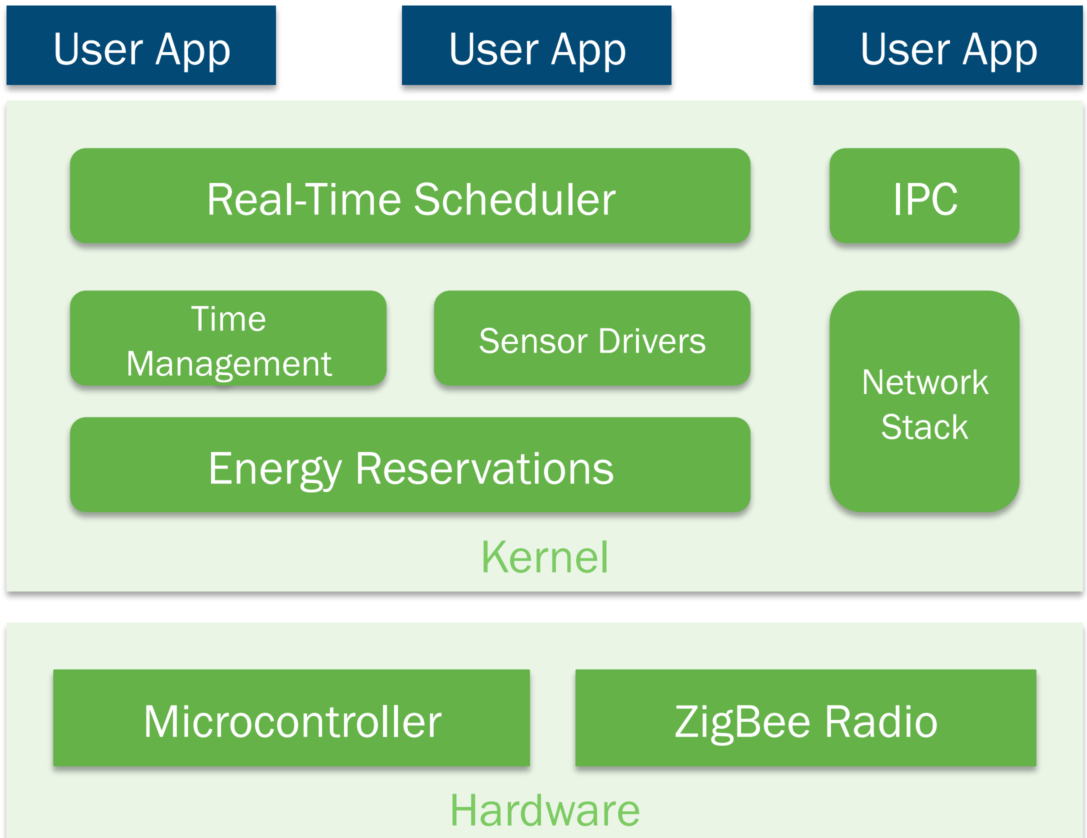
Outline

- Hardware platform for sensor networks
- The NanoRK operating system
- Developing applications with NanoRK
- Some tips and tricks

NanoRK feature list

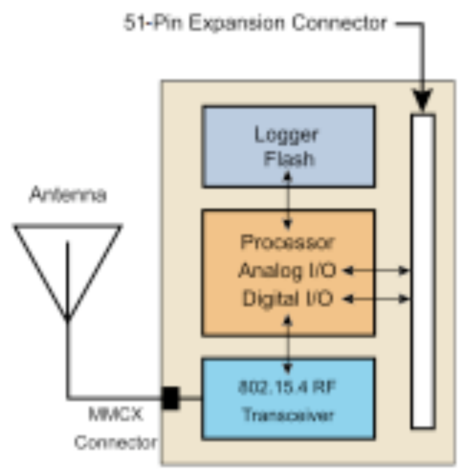
- C GNU tool chain
- Classical preemptive operating system multitasking primitives
- Static priority scheduling support
- Fault handling
- Energy-efficient scheduling
  - Based on *a priori* task set knowledge

NanoRK architecture



MICAz mote module

- Atmel ATmega128L microcontroller
- IEEE 802.15.4 compliant RF receiver
- 250 kbps date rate
- Program flash memory: 128K bytes
- Measurement (serial) flash: 512K bytes
- Configuration EEPROM: 4K bytes



MPR2400 Block Diagram

NanoRK reservations

- CPU utilization
  - Time allowed per period
  - Example: 10 ms every 250 ms
- Network utilization
  - Packets in/out per period
- Sensors/actuators usage
  - Sensor readings per second
- [CPU, Network, Peripherals]
  - Represent total energy usage
  - Static offline budget enforcement

NanoRK time management

- ~1ms OS tick resolution
  - Variable tick timer (interrupts occur as required, not every quantum)
- wait_until_xxx ( ) functions
  - Suspend task until event or timeout occurs
  - Enforces reservations
- If reserves are disabled then low priority tasks can starve
  - And battery is wasted

NanoRK fault handlers

- Task time violations
  - OS will enforce time bounds allocated to a task
- Canary stack check
  - Check if user-specified stack has an overflow
  - Not 100%, but incurs low overhead and is better than doing nothing
- Unexpected restarts
  - Capture restart that occurs without power down
- Resource over-use
  - Manage sensors and actuators
- Low voltage detection
- Watchdog timer

Configure NanoRK task set

nrk_cfg.h

```
#define NRK_REPORT_ERRORS
// print error over serial

#define NRK_HALT_ON_ERROR
// stop the kernel if an error happens

// Enable Canary Stack Check
#define NRK_STACK_CHECK

// Max number of tasks in your application
// Be sure to include the idle task
// Making this the correct size will save on BSS memory which
// is both RAM and ROM
#define NRK_MAX_TASKS              5

#define NRK_TASK_IDLE_STK_SIZE    128
// Idle task stack size min=32

#define NRK_APP_STACKSIZE          128
#define NRK_KERNEL_STACKSIZE       128
#define NRK_MAX_RESOURCE_CNT       1
```

Creating a NanoRK task

```
NRK_STK Stack1[NRK_APP_STACKSIZE];
nrk_task_type TaskOne;
void Task1(void);

...

TaskOne.task                = Task1;
TaskOne.Ptos                = (void *) &Stack1[NRK_APP_STACKSIZE];
TaskOne.Pbos                = (void *) &Stack1[0];
TaskOne.prio                = 2;
TaskOne.FirstActivation     = TRUE;
TaskOne.Type                = BASIC_TASK;
TaskOne.SchType             = PREEMPTIVE;
TaskOne.period.secs         = 0;
TaskOne.period.nano_secs = 100*NANOS_PER_MS;
TaskOne.cpu_reserve.secs = 0;
TaskOne.cpu_reserve.nano_secs = 10*NANOS_PER_MS;
TaskOne.offset.secs         = 0;
TaskOne.offset.nano_secs = 0;
nrk_activate_task (&TaskOne);
```

Sample NanoRK task

```
void Task1()
{
     uint16_t cnt,buf;
     int8_t fd,val;

     printf( "My node's address is %d\r\n",NODE_ADDR );
     printf( "Task1 PID=%d\r\n",nrk_get_pid());

     // Open ADC device as read
     fd=nrk_open(FIREFLY_SENSOR_BASIC,READ);
     if(fd==NRK_ERROR)
          nrk_kprintf(PSTR("Failed to open sensor driver\r\n"));
     cnt=0;
     while(1) {
          nrk_led_toggle(BLUE_LED);
          // Example of setting a sensor
          val=nrk_set_status(fd,SENSOR_SELECT,BAT);
          val=nrk_read(fd,&buf,2);
          printf( "Task1 bat=%d",buf);
          val=nrk_set_status(fd,SENSOR_SELECT,LIGHT);
          val=nrk_read(fd,&buf,2);
          printf( " light=%d",buf); cnt++;
     }
     nrk_close(fd);
}
```

Receiving a data packet

```
void rx_task() {
     uint8_t i, len;
     int8_t rssi, val;
     uint8_t *local_rx_buf;

     // init bmac on channel 25
     bmac_init(25);      bmac_rx_pkt_set_buffer
(rx_buf,RF_MAX_PAYLOAD_SIZE);

     while(1) {
          // Wait until an RX packet is received
          val=bmac_wait_until_rx_pkt();
          // Get the RX packet
          local_rx_buf=bmac_rx_pkt_get(&len,&rssi);

          printf( "Got RX packet len=%d RSSI=%d [",len,rssi);
          for(i=0; i<len; i++ ) printf( "%c", local_rx_buf[i]);
          printf( "]\r\n" );
          // Release the RX buffer so future packets can arrive
          bmac_rx_pkt_release();
     }
}
```

Sending a data packet

```
void tx_task() {
     uint8_t j, i, val, len, cnt;

     printf( "tx_taskPID=%d\r\n",nrk_get_pid());
     // Wait until the tx_task starts up bmac
     // This should be called by all tasks using bmac that
     // do not call bmac_init()...

     while(!bmac_started()) nrk_wait_until_next_period();
     cnt=0;
     while(1) {
          // Build a TX packet
          sprintf( tx_buf, "This is a test %d",cnt );
          cnt++;

          // transmit the packet
          val=bmac_tx_packet(tx_buf, strlen(tx_buf));
          // Task gets control again after TX complete
          nrk_kprintf( PSTR("TX task sent data!\r\n“) );
          nrk_wait_until_next_period();
     }
}
```

Tips and Tricks #1

- **Don't Allocate Large Data Structures Inside Functions**
  - Allocating large data structures in functions puts them on the stack
  - Make them global if need be (bad style for a PC, but this isn't a PC)
  - Stack is usually 128 bytes!
- **Take Care When Passing Large Data Types to Functions**
  - Pass large structures by reference using pointers so less data gets pushed on the precious stack
- **Avoid Recursive Function Calls**
  - Recursive function calls keep pushing onto the stack each time they recurse
- **Use "inline" For Speed And To Save Stack Space**
  - "inline" in C avoids function calls and (you guessed it) doesn't push onto the stack

UBC

Tips and Tricks #2

- **Be very careful with Dynamic Memory**
  - malloc does work, but can cause fragmentation and all sorts of other problems. Use with EXTREME care or better yet not at all.
- **Watch out for strings**
  - Strings declared anywhere consume DATA and hence use RAM.
  - They don't show up using avr-nm
  - Sometimes it is better to pass a numerical value to a function that has a big kprintf() switch inside it
- **Use nrk_kprintf() whenever possible for constant strings**
  - nrk_kprintf() stores strings in FLASH memory using the PSTR() macro
  - Only use regular printf() when the string is dynamic (i.e., you use “%d” to print variables, etc.)

Tips and Tricks #3

How Much Memory Is My Code Using?

- .data is the amount of RAM that your program uses that is defined at startup as a particular value
  - Consumes RAM and ROM
- .bss is the amount of zeroed-out RAM your program uses
  - Consumes RAM only
- **RAM = .data + .bss (+ Kernel Stack)**
- **FLASH = .data + .text**
- Stack appears in .bss section EXCEPT for Kernel, so add Kernel stack to RAM figure

```
Size after:
main.elf  :
section          size          addr
.data             220       8388864
.text           17258             0
.bss             1021       8389084
.stab           41268             0
.stabstr        16934             0
Total           76701
```

RAM = 220 + 1021 + 128 = 1369 bytes
FLASH = 17258 + 220 = 17478 bytes
Total RAM = 4096 bytes
Total ROM = 131072 bytes

Tips and Tricks #4

What variables are using up memory?

- Use `avr-nm` (name) to find a list of symbols and how much memory is consumed

```
avr-nm –S –radix=d –size-sort main.elf
…
(address)      (size)
08388989     00000001 D NRK_UART1_TXD
…
08389446     00000116 B tx_buf
00012074     00000118 T nrk_event_wait
```

"T" refers to the text section. "B" refers to the BSS (what is this?) section. "D" refers to the data section. Strings do not appear in this list because they do not have compiler-mapped labels.

Endnotes

- Read the code for NanoRK and the example applications
- Learn by doing

- Lecture was only a cursory look at NanoRK
- More information: http://www.nanork.org/
- Also read: A. Eswaran, A. Rowe and R. Rajkumar, "**Nano-RK: An Energy-Aware Resource-Centric Operating System for Sensor Networks**," IEEE Real-Time Systems Symposium, December 2005.