(Revised) Rough Notes on Programming
AVR Microcontrollers in C.

Mechanical Engineering Report 2007/04
P. A. Jacobs
School of Engineering
The University of Queensland.

February 21, 2008

Preface

These notes follow on from the material that you studied in CSSE1000 Introduction to Computer Systems. There you studied details of logic gates, binary numbers and instruction set architectures using the Atmel AVR microcontroller family as an example. In your present course (METR2800 Team Project I), you need to get on to designing and building an application which will include such a microcontroller. These notes focus on programming an AVR microcontroller in C and provide a number of example programs to illustrate the use of some of the AVR peripheral devices.

Session Contents

1. Introduction to the hardware and software development environment. A small but complete application example is implemented with an ATmega88 microcontroller on the STK500 development board.


3. A more extensive application the uses serial communication and more C control structures. Decision and control statements. Functions, scope of variables.

Contents

1 Embedded computing system based on Atmel’s AVR 4

2 Example: flash a LED 7
  2.1 Example peripheral: PORTD bidirectional port 7
  2.2 Development tools – software and hardware 8
  2.3 Preparing the firmware 9
  2.4 C source code 11

3 Example 2: a task loop and a hardware timer 13
  3.1 C source code 14
  3.2 Your own prototype board with ISP 15

4 Elements of a C program 17
  4.1 Types of data 18
  4.2 Expressions 20
  4.3 Assignment statements 22

5 Example 3: Serial-port communication 23
  5.1 C source code 24

6 Decision and flow-control statements 26
  6.1 Conditional statements 26
  6.2 Loops 27
  6.3 Unconditional control statements 29

7 Functions 30

8 Pointers-to and addresses-of variables 32

9 Functions revisited: pass-by-reference 33

10 Arrays 34
  10.1 Pointers and arrays 35
  10.2 Strings 36

11 The C preprocessor 36

12 Example 4: Using the analog-to-digital converter 38
  12.1 C source code 38

13 Interrupts 42
  13.1 Example 5: Using interrupts with a hardware timer 44
  13.2 Example 6: Using interrupts to count button presses 45

14 Software architectures of embedded systems 46
  14.1 Round-Robin 46
  14.2 Round-Robin with interrupts 47
A   ANSI C Reference Card  49
1 Embedded computing system based on Atmel’s AVR

- AVR microcontroller units (MCUs) all have the same core, i.e. same instruction set and memory organisation.

- “flavours” are based on features and complexity
  - tinyAVR: reduced feature set
  - megaAVR: lots of features and peripheral devices built in
  - AVR or classic AVR: in between range of features

- selection based on
  - tight budget - choose one with just enough functionality
  - convenience of development - choose one with “bells and whistles”

- in CSSE1000/COMP1300, you used the ATmega8515, a digital-only MCU

- for these demonstrations, we will work with the ATmega88
  - has a nice selection of features (see following page), including a serial port and an analog-to-digital converter (with several input channels)
  - 28-pin narrow DIL package is convenient for prototyping and there are enough I/O pins to play without needing very careful planning.
  - pinout is shown at the start of the Atmel datasheet (book) on the ATmega88 [1]. You will be reading the pages of this book over and over...
  - internal arrangement is around an 8-bit data bus
  - “Harvard architecture” with separate paths and storage areas for program instructions and data
  - We won’t worry too much about the details of the general-purpose registers, the internal static RAM or the machine instruction set because we will let the C compiler handle most of the details.
  - However, memory layout, especially the I/O memory layout is important for us as C programmers; the peripheral devices are controlled and accessed via special function registers in the I/O memory space. For the ATmega88 data memory, we have:
    * 32 general purpose registers (0x0000–0x001F)
    * 64 I/O registers (0x0020–0x005F)
    * 160 extended I/O registers (0x0060–0x00FF)
    * 1kbytes internal SRAM (0x0100–0x04FF)
Features

- High Performance, Low Power AVR® 8-Bit Microcontroller
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 20 MIPS Throughput at 20 MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory segments
  - 4/8/16K Bytes of In-System Self-programmable Flash program memory
  - 256/512/512 Bytes EEPROM
  - 512/1K/1K Bytes Internal SRAM
  - Write/Erase cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C
  - Optional Boot Code Section with Independent Lock Bits
  - In-System Programming by On-chip Boot Program
  - True Read-While-Write Operation
  - Programming Lock for Software Security

- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
  - Real Time Counter with Separate Oscillator
  - Six PWM Channels
  - 8-channel 10-bit ADC in TQFP and QFN/MLF package
  - 6-channel 10-bit ADC in PDIP Package
  - Programmable Serial USART
  - Master/Slave SPI Serial Interface
  - Byte-oriented 2-wire Serial Interface (Philips I²C compatible)
  - Programmable Watchdog Timer with Separate On-chip Oscillator
  - On-chip Analog Comparator
  - Interrupt and Wake-up on Pin Change

- Special Microcontroller Features
  - Power-on Reset and Programmable Brown-out Detection
  - Internal Calibrated Oscillator
  - External and Internal Interrupt Sources
  - Five Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, and Standby

- I/O and Packages
  - 23 Programmable I/O Lines
  - 28-pin PDIP, 32-lead TQFP, 28-pad QFN/MLF and 32-pad QFN/MLF

- Operating Voltage:
  - 1.8 - 5.5V for ATmega48V/88V/168V
  - 2.7 - 5.5V for ATmega48/88/168

- Temperature Range:
  - -40°C to 85°C

- Speed Grade:
  - ATmega48V/88V/168V: 0 - 4 MHz @ 1.8 - 5.5V, 0 - 10 MHz @ 2.7 - 5.5V
  - ATmega48/88/168: 0 - 10 MHz @ 2.7 - 5.5V, 0 - 20 MHz @ 4.5 - 5.5V

- Low Power Consumption
  - Active Mode: 250 µA at 1 MHz, 1.8V
  - 15 µA at 32 kHz, 1.8V (including Oscillator)
  - Power-down Mode: 0.1µA at 1.8V
Block diagram for the ATmega88, as scanned from the datasheet [1].

Pin layout for the ATmega88 for the 28-pin Plastic Dual Inline Package (PDIP), as scanned from the datasheet.
2 Example: flash a LED

- The microcontroller version of the “Hello, World” program.
- We’ll look at the hardware, development environment and software required to make this happen.

2.1 Example peripheral: PORTD bidirectional port

Schematic diagram for a digital I/O pin, as scanned from the datasheet.

![Schematic diagram for PORTD](image)

- peripheral device can be used for digital input or output
- three I/O memory addresses are assigned to this port

<table>
<thead>
<tr>
<th>I/O address</th>
<th>bits</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0B (0x2B)</td>
<td></td>
<td>PORTD register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDRD direction</td>
</tr>
<tr>
<td>0x0A (0x2A)</td>
<td></td>
<td>PIND input</td>
</tr>
<tr>
<td>0x00 (0x29)</td>
<td></td>
<td>data registers</td>
</tr>
</tbody>
</table>

- The names for the registers and the bits are contained in the header file supplied with the C compiler. On my Linux computer, these definitions may be found in `/usr/avr/include/avr/iomx8.h`.

Note: 1. WRx, WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. clkVo, SLEEP, and PUD are common to all ports.
• Registers DDRD and PORTD are readable and writeable with initial values for all bits being 0. Register PIND is readable only.

• Writing a 0 to a bit in DDRD sets the corresponding pin to input (and a 1 will set the pin to output. Note that initial value for DDRD bits being 0 implies that all pins are initially configured for input.

• When set as an input pin, a pull-up resistor can be activated by writing a 1 to the corresponding PORTD bit.

• Output buffers can source or sink an absolute maximum current of 40mA per I/O pin and the whole device can cope with a total of 200mA. See section 27 of the datasheet.

2.2 Development tools – software and hardware

• We will be using Atmel’s AVR Studio 4 as our Integrated Development Environment.
  – Consists of editor, simulator, device programming software.
  – We don’t have to remember so much about the individual tools because we can usually select actions and options from the menus.
  – Visit http://www.atmel.com to download a copy for home.

• We will use the WinAVR programming tools.
  – Consists of the GNU C compiler and the AVR-libc standard C library and peripherals library.

• Atmel’s STK500 development hardware which supports the “target” MCU by providing power, clock signal and input/output devices (such as switches, LEDs and an RS232 port).

• To install our ATmega88 MCU in the STK500,
  – use socket SCKT3200A2 (the green one)
  – use the In-System Programming (ISP) header SPROG2 and connect (with the 6-pin cable) to the appropriate header (green)
  – Other jumpers that should be installed:
    ∗ VTARGET, supply voltage to the MCU
    ∗ AREF, analogue reference voltage
    ∗ RESET, allow the master MCU (on the top of the STK500) to control the target’s reset pin.
    ∗ XTAL1, use STK500 on-board clock as the main clock to the target MCU.
* OSCSEL, jumper 1—2 selects the on-board software-generated clock. This is handy on the STK500 because you can adjust its frequency but remember that, when you build your own board, you must have a working clock for the MCU to respond to in-system programming. It could be the MCU’s internal 8MHz clock if you don’t want to build an external clock circuit.

* make sure that BSEL2 is not jumpered.

– Finally, connect a 10-pin jumper cable from PORTD header to the LEDs header.

2.3 Preparing the firmware

(a) Start AVR Studio

(b) Create a new project and configure settings.

project type: AVR GCC
project name: flashleds1
create initial file √
create folder √
location: C:\avr-work
next
dev platform: AVR Simulator
device: ATmega88
Also, at this time, configure the editor (Tools→Options→Editor) to set tab width 4 and check “replace tabs with spaces”.

(c) Type in source code (from the following section)
Save as “flashleds1.c” and use Ctrl-s frequently.

(d) Build the project. (Build→Rebuild All) and check for errors.

– Behind the GUI, avr-gcc has cross-compiled the program and then linked it with the standard library and start-up code to produce an executable program that can be downloaded to the MCU. Alternatively, this code may be run in the simulation environment.

– Note the messages in the lower window (labelled “Build”).

– Note the creation of an “Intel HEX” file that we will later download to the MCU.

(e) Start the simulator (Debug→Start Debugging)

– Note that the yellow arrow points to the first line of code to be executed.

– Stepping options are shown in the toolbar (nongreyed now).

– Open the IO View entry I/O ATmega88, PORTD.

– “Step into” the code a couple of times to see the DDRD bits change.

– Hover the cursor over the “which_bit” variable name to see details of this variable’s value and location in memory. This doesn’t work for DDRD because that is a macro name, however, one can see its address as well as its content in the I/O View window.
(f) Download/program the executable program into the MCU.

- Tools→Program AVR...→Connect→Select “STK500 or AVRISP” and we should get “STK500 Window” showing the state of the target MCU.
- Make sure that the power is supplied to the board and that the “status” LED is a steady green.
- Press “Program” to download the executable program and run it.
- Note messages in the log window telling us how the download process went.
- Note the “Fuse” settings. The factory-set default settings on the ATmega88 seem to be
  * Boot flash section size=1024 words, Boot start address=$0c00.
  * Serial program downloading enabled (greyed on).
  * Brown-out detection disabled.
  * Divide clock by 8 internally.
  * Internal RC oscillator 8MHz; Start up time: 6CK/14CL+65ms

We could try setting
  * External clock; Start up time: 6CK/14CL+65ms
  * Unset divide clock by 8 internally

then experiment with setting the STK500 Oscillator frequency. Values line 3.686 MHz seem odd but are good for RS232 communications.

- Close the STK500 connection window.
  Save project.
  Exit AVR Studio.
  The MCU should continue to run independently so long as power is applied.
2.4 C source code

// flasheds1.c
// Try out the AVR Studio + AVRGCC + STK500.
// PJ, 26−Feb−2007

// get the processor−specific definitions for the IO memory space
#include <avr/io.h>

// macro definition to select bit n, assuming 0 <= n <= 7
#define BIT(n) (1 << (n))

void my_delay(unsigned char c1_start)
{
    unsigned char c1, c2;
    unsigned int count = 0;
    for (c1 = c1_start; c1 > 0; --c1) {
        for (c2 = 255; c2 > 0; --c2) {
            count += 1;
        }
    }
    // we ignore the value of count
    return;
}

int main(void)
{
    unsigned char which_bit;
    DDRD = 0xFF; // all output
    which_bit = 0;
    while (1) { // never-ending loop
        which_bit += 1;
        if (which_bit > 7) which_bit = 0;
        // Turn on a LED by writing 0 to its corresponding MCU pin
        PORTD = ~BIT(which_bit);
        my_delay(10);
    }
    return 0; // we should never get here.
}

Notes on this program:

- Have used C++-style comments.
- Simple statements end with ";".
- Compound statements are delimited by "{" and "}".
- Execution starts at the main function.
- Nonzero values are interpreted at "true" in a Boolean context.
- Operations on individual bits in GCC.

// bit position macro
#define BIT(n) (1 << (n))

// turn on bit n, leaving the others unchanged
PORTD |= BIT(n);
// turn off bit n, leaving the others unchanged
PORTD &= ~BIT(n);

// toggle bit n
PORTD ^= BIT(n);

- Standard C integer objects (desktop computing)
  - int is at least 16 bits
  - sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)

- GCC on AVR
  - char is 8 bits
  - can use shift operators to multiply or divide by powers of 2
  - there is no hardware divide but newer MCUs have hardware multiply
  - avoid the use of floating-point data and operations

- LED connection on STK500
3 Example 2: a task loop and a hardware timer

- The simplest arrangement for an embedded program that performs a loop of tasks with a fixed period.
- Flowchart is somewhat trivial but does visualize the task loop clearly.
3.1 C source code

// flashleds2.c

// The application doesn’t do much; it just lights one LED at a time
// across the bottom of the STK500 board.
// It does, however, demonstrate the use of a hardware timer
// to provide a regular period for the main loop of the application.

// PJ, 27-Feb-2007 original for AT90S4433
// 11-Feb-2008 update to ATmega88

// get the processor–specific definitions for the IO memory space
#include <avr/io.h>

// macro definition to select bit n, assuming 0 <= n <= 7
#define BIT(n) (1 << (n))

void setup_timer1(void)
// We will use the 16–bit timer/counter 1 without interrupts
// and we will slow things down by prescaling the system clock.
{
    // Normal port operation, OC1A and OC1B disconnected
    // Leave bits COM1A1 through COM1B0 of TCCR1A as default zero values
    // prescale CK/1024
    TCCR1B |= BIT(CS12) | BIT(CS10);
    return;
}

void reset_timer1(unsigned int counts)
// With a clock frequency of 3.686 MHz on the STK500 and
// a 1024 prescaling factor, we expect a count of 360 to result
// in timer1 overflow after 100 milliseconds.
{
    TIFR1 |= BIT(TOV1); // clear the overflow bit by writing logical 1
    TCNT1 = 0xFFFF - counts; // set the starting count as a 16–bit value
    return;
}

void wait_for_timer1(void)
{
    while ((TIFR1 & BIT(TOV1)) == 0) /∗ do nothing ∗/ ;
    return;
}

int main(void)
{
    unsigned char which_bit;
    DDRD = 0xFF; // all output
    setup_timer1();
    which_bit = 0;
    while (1) { // never–ending loop
        reset_timer1(360); // Now, we have 100ms to do our work.

        // The body of this loop doesn’t do much work.
        // It just makes the next LED turn on before wasting time
        // waiting for timer/counter 1 to overflow.
        which_bit += 1;
        if ( which_bit > 7 ) which_bit = 0;
        // Turn on a LED by writing 0 to its corresponding MCU pin
        PORTD =˜BIT( which_bit );

        // Hopefully, there is some time to wait for the timer to overflow.
        wait_for_timer1();
    }
    return 0; // we should never get here.
}
3.2 Your own prototype board with ISP

- Encourage groups to make a prototype on stripboard.
- Allows software development well before printed-circuit boards have been manufactured.
- Soldered joints are more reliable than spring contacts in breadboards.
- Photograph shows a partly constructed prototype board for a timer. The coloured ribbon cable connects the 6-pin ISP header from the STK500 to a 6-pin inline header on the prototype board. The other pairs of wires are to LEDs that will eventually be mounted on the front panel of the device.
- There is a voltage regulator on the top part of the board for when the board is not powered through the ISP cable.
- An external full-swing crystal oscillator is used for the clock signal.
- Current-limiting resistors are used with the peripheral connectors.
- One pair of lines (RXD and TXD) can be connected to PC’s serial port through the level-shifting chip on the STK500.
Circuit-diagram for the prototype board, including the voltage regulator and the ISP header.
4 Elements of a C program

• We will be dealing with data (variables, register values) and instructions (program code, functions) on what should be done with the data.

• If you want some further reading, see the book [2]. It has a good tutorial introduction but uses a different compiler for the AVRs.

• Basic program structure

// heading comments
...

int main( void ) // this function is special; execution starts here
{
    ... local definitions ...
    ... executable statements ...
    return 0;
}

For a small embedded program, it is unlikely that we should never arrive at the return statement. For a desktop computing application, the zero will be returned to the operating system.

• Comments

// C++ style comment
/* C style comment */

Be careful to avoid nesting the C-style comments.

• Statements

– consist of identifiers, keywords, expressions and other statements
– a single statement ends with a semicolon;
  
b = 10; /* assignment of integer value 10 to variable b */
  if ( i > 1 ) b = a - 2;

  The second statement contains a keyword, expression and another single statement.
– a compound statement is delimited by braces
  
  if ( i > 1 ) { // begin compound statement
      b = a - 2;
      j = j + 1;
  } // end if statement

  Note that there is no semicolon at the end of the compound statement. Putting one there would introduce a null statement.
• An **identifier** is the name of a variable or function.
  - must start with a letter of the alphabet or an underscore `_`
  - this can be followed by any number of letters, digits or underscores
  - usually the first 32 characters are significant
  - case sensitive
  - recommendations
    * identifiers starting with underscores are usually reserved for system names
    * don’t rely on case sensitivity to distinguish names
    * use descriptive names

• **keywords** (or reserved words) are names that have special meaning to the compiler
  - examples: *if* *while* and *for*
  - for a complete list, see the appendix A
  - these cannot be used for variable or function names

• A variable labels a specific memory location (or register) that is set aside to hold a particular type of data.
  - Each variable is identified by its name.
  - Each must be defined before use in a *declaration statement* that may also initialize the value of the variable.
  - General form of the variable declaration statement
    \[\text{type name} ; // \text{comment}\]
  - Examples:
    \[
    \text{int count; // This will accumulate the number of key presses.}
    \text{int count = 0; // On startup, we know that nothing has happened.}
    \]

### 4.1 Types of data

• Internally, all data is stored and handled as binary numbers. All other types are interpretations of the bit patterns.

• **Characters**
  - Examples:
    \[
    \text{char letter; // 8-bit quantity on the AVR.}
    \text{letter = 'a'; // Note the use of single quotes.}
    \]
  - Some special characters
    * \`\b` backspace
    * \`\f` form feed
    * \`\n` new line
    * \`\r` carriage return
In AVR programming, we will use `char` variables to hold 8-bit data (a byte) without necessarily interpreting the data as a character.

- **strings** are arrays of characters

```c
char my_name[] = "Peter";
```

The square brackets signify an array. Note that we use double quotes for string constants.

The storage in memory is `'P','e','t','e','r',\000` where the trailing null character marks the end of the string.

- **int** variables are used to store 16-bit data.
  - `unsigned int` is used to represent a whole number in the range 0 .. 65535
  - (signed) `int` has the range -32768 .. 32767
  - Be careful with large values; the number system wraps around.
  - Usually integers are written in base 10 (or decimal) with digits 0..9.
  - When working with AVR MCUs, it is often convenient to use other bases.
    - base 2 (binary) for single bits
    - base 8 (octal) for groups of 3 bits
    - base 16 (hexadecimal) for groups of 4 bits
  - Examples:
    - `0b1011` binary (0b prefix)
    - `11` decimal (no prefix or leading zero)
    - `013` octal (leading zero)
    - `0xB` hexadecimal (0x prefix)

- **float** variables are used to represent numbers with a fractional part
  - Can only represent a subset of real numbers, for example, 5.5, -2.31 and 9.0.
  - Note the decimal point. This is what tells the compiler to use a floating-point representation.
  - floats are represented as 32-bit patterns on AVR MCUs.
  - Operations are done in software and are slow.
  - Try to avoid using float data on AVR, however, it’s hard to beat the convenience of representing real numbers as float data.
  - Range for float numbers -3.4e+38 .. 3.4e+38
  - Precision is about 6 decimal digits.
• boolean or logical values
  – There is no dedicated type.
  – Use an interpretation of integer data.
    0 is equivalent to false.
    Any nonzero value is interpreted as true.
  – Logical expressions evaluate to 1 (for true) or 0 (for false).

• program space
  Data may be stored in the flash memory of the AVR. This is separate to the normal RAM data space.

```c
#include <avr/pgmspace.h>
...
prog_char c;
prog_int16_t j;
```

There are also a number of functions in AVR libc to copy data into and out of program space.

### 4.2 Expressions

Expressions are used to specify simple computations. An expression is a combination of operands and operators that express a single value.

• Examples:

```c
int a = 1;
int b = 2;
...
a + 2  // expresses an arithmetic value 3
a >= b  // expresses a logical value 0
a = b  // expresses an assignment of the value 2
```

• arithmetic operators
  + addition (binary operator)
  − subtraction
  * multiplication
  / division (note that newer AVR MCUs have hardware multiplication but not division)
  % modulus (or remainder)
  ++, −− increment and decrement (one operand only)

• precedence
  1. ()
  2. negation
  3. *, /
  4. +, −
• relational operators
  
  \begin{align*}
  &= & \text{is equal to} \\
  > & \text{is greater than} \\
  \geq & \text{is greater than or equal to} \\
  < & \text{is less than} \\
  \leq & \text{is less than or equal to} \\
  \neq & \text{is not equal to}
  \end{align*}

• logical operators result in true (1) or false (0) values
  
  \begin{align*}
  && & \text{and} \\
  ! & \text{not} \\
  || & \text{or}
  \end{align*}

• bitwise operators
  
  \begin{align*}
  \sim & \text{ones complement} \\
  << & \text{shift bit pattern left (with zero fill)} \\
  >> & \text{shift bit pattern right} \\
  & & \text{and} \\
  | & \text{or} \\
  \wedge & \text{exclusive-or}
  \end{align*}

• More examples:

\begin{verbatim}
char i = 0b0011;
char j = 0xB;  // 0b00001011

~i   has the value 0b11111100
i << 2       0b00001100
i & j         0b00000011
\end{verbatim}

- The right operand for the shift operators gives the number of bit-places to shift.
- Each left shift is equivalent to multiplying the left operator by 2.
- Right shift treats signed and unsigned quantities differently.
  unsigned: shifts zero into the most-significant bit signed: the sign bit will be replicated

• Rules of thumb for precedence
  1. multiply and divide come before add and subtract
  2. put parentheses around everything else

• For arithmetic operations, the operand of smaller type is coerced to the larger type.
  For example, \texttt{int} is a larger type than \texttt{char} so, if we have an addition operation involving both \texttt{int} and \texttt{char} operands, the \texttt{char} value will be promoted to an \texttt{int} value before the operation is performed.
4.3 Assignment statements

- A declaration sets aside space in memory for the variable.

    unsigned char z;  // 8-bit quantity

- An assignment statement is used to give the variable a value.

    General form:
    
    \textit{variable} = \textit{expression} ;

    Example:
    
    \texttt{z = 22;}

- The assignment operator means: compute the value of the expression on the right
  and assign that value to the variable.

- May initialize variables in their declaration statement.

    \texttt{int upper_limit = 10000;}

- Sometimes we want a value that does not change.

    \texttt{const int upper_limit = 10000;}

- Interacting with the AVR hardware via the special function registers in I/O memory.

    \texttt{#include <avr/io.h>}
    
    \texttt{int main( void )}
    
    \{  
    
        unsigned char z;
        DDRD = 0xFF;  // sets all port pins as output
        while (1) {
            z = PINB;  // read the binary value from port B
            PORTD |=(z & 0xF); // set the least-significant 4 bits
                              // to look like those at port B
        }
    
    return;
    
    \}

    The read-modify-write statement could be written as
    
    \texttt{PORTD = PORTD | (z & 0xF);}
5 Example 3: Serial-port communication

This example also demonstrates some of the C flow-control structures. The implicit communication between the blocks is via variables.
5.1 C source code

```c
#include <avr/io.h>

// countupdown.c
//
// This application takes input from the buttons at the bottom of
// the STK500 board and accumulates a count of the key presses.
// SW0 == clear total
// SW1 == increment total
// SW2 == decrement total
// Remember to connect PORTB to the switches.
//
// The total is output to the spare RS232 port as a hexadecimal number.
// Remember to connect pins PD0,PD1 to the header RxD,TxD–spare
// with a 2–wire jumper cable.
//
// PJ, 28–Feb–2007 original for AT90S4433
// 12–Feb–2008 revised for ATmega88

#define FOSC 3686400
#define BAUD 9600

void setup_timer1(void)
// We will use the 16–bit timer/counter 1 without interrupts
// and we will slow things down by prescaling the system clock.
{
    TCCR1B |= (1 << CS12) | (1 << CS10); // prescale CK/1024
    return;
}

void reset_timer1(unsigned int counts)
// With a clock frequency of 3.686 MHz on the STK500 and
// a 1024 prescaling factor, we expect a count of 360 to result
// in timer1 overflow after 100 milliseconds.
{
    TIFR1 |= (1 << TOV1); // clear the overflow bit by writing logical 1
    TCNT1 = 0xFFFF - counts; // set the starting count
    return;
}

void wait_for_timer1(void)
{
    while ( (TIFR1 & (1 << TOV1)) == 0 ) /* do nothing */ ;
    return;
}

void setup_uart(void)
{
    UBR0 = MYUBRR; // a value of 23 for 9600 baud with FOSC=3.686 MHz
    UCSR0B |= (1 << TXEN0); // connect the transmit to PD1
    return;
}

void send_character(unsigned char c)
{
    // Note the null statements in the bodies of the following loops.
    while ( (UCSR0A & (1 << UDRE0)) == 0 ) /* wait until empty */ ;
    UDR0 = c; // send the character
    while ( (UCSR0A & (1 << TXC0)) == 0 ) /* wait until complete */ ;
}

unsigned char hex_repr(unsigned char n)
```
// Return a hexadecimal character representing the binary value.
{
    n &= 0x0F; // keep only 4 bits, just in case the user has sent more
    if ( n < 10 ) {
        return n + '0';
    } else {
        return (n-10)+'A';
    }
}

void send_number(unsigned int n)
// Send a 4-digit hexadecimal representation of the number
// to the serial port, starting with the most significant digit.
{
    unsigned char nybble, i;
    for ( i = 0; i < 4; i++ ) {
        nybble = (n & 0xF000) >> 12; // get most significant digit
        send_character(hex_repr(nybble));
        n = n << 4; // move remaining bits left
    }
}

// ------------------ the main program/loop ------------------
int main(void)
{
    unsigned char total, old_total, key_bits;
    // PORTB defaults to input; this is where the switches are connected.
    setup_timer1();
    setup_uart();
    total = 0;
    while (1) { // never-ending loop
        old_total = total;
        key_bits = PINB & 0b00000111; // only look at SW2, SW1 and SW0
        // Note that, when pressed, the switches pull the MCU pins low.
        if ( (key_bits & 0b001) == 0 ) {
            total = 0;
        } else if ( (key_bits & 0b010) == 0 ) {
            total++;
        } else if ( (key_bits & 0b100) == 0 ) {
            total--;
        } if ( old_total != total ) {
            // Only send value when that has been a change.
            send_number(total);
            send_character(\n'); // new line
            send_character(\r'); // carriage return
        } if ( old_total != total ) {
            // Hopefully, there is some time to wait for the timer to overflow.
        wait_for_timer1();
    }
    return 0; // we should never get here.
}
6 Decision and flow-control statements

Some of these notes were adapted from texts [3] and [4].

6.1 Conditional statements

- simplest form

\[
\text{if ( expression ) statement}
\]

The statement may be a single or a compound statement.

- \textbf{else} clause for a case where the condition is false

\[
\text{if ( expression )}
\text{ statement-1}
\text{ else}
\text{ statement-2}
\]

- Even if statement 1 and 2 are single statements, it is a good idea to use braces and indentation to show the structure unambiguously.

- We can nest if statements to get more than two execution paths.

- Unless otherwise indicated by the use of braces, an \textbf{else} clause belongs to the innermost if statement that does not yet have an associated \textbf{else} clause. (page 52, [3])

- For many execution paths, use a \textbf{switch} statement.

\[
\text{switch ( expression ) }
\{
\text{ case const-expr-1 : statements-1}
\text{ case const-expr-2 : statements-2}
\text{ case const-expr-3 : statements-3}
\text{ default : statements}
\}
\]

The \textit{expression} is evaluated and is tested for a match with a number of integer-valued
constants (the `const-expr` values) If a match is found, execution starts at that case. All of the statements from that point until the end of the `switch` construct will be executed or until a `break` statement is encountered. Using a `break` statement is the normal way to stop “fall-through” between `case` clauses. However, making use of fall-through, we may have several cases for one particular set of statements. The optional `default` case is executed if none of the others are satisfied.

6.2 Loops

- pretested loop

```c
while (expression) statement
```

- post-tested loop

```c
do statement while (expression);
```
• for statement

```latex
for ( \text{init\_expr}; \text{cond\_expr}; \text{final\_expr} ) \text{statement}
```

- `init\_expr` is usually an assignment expression.
- The `statement` is executed if `cond\_expr` is nonzero (true). Note that `cond\_expr` does not have to be a boolean expression.
- `final\_expr` is evaluated each time, after the `statement` is executed. It is usually an increment expression.
- Note the semicolons separating the expressions.
- An example of a never-ending loop with a null statement as the body.
  
  ```latex
  for (;;) ;
  ```
6.3 Unconditional control statements

- **break;**
  - Flow of control jumps to the statement immediately following the body of the statement containing the `break` statement.
  - Use for exceptional situations where a loop must be abandoned.
  - Also used to stop execution “falling through” from one case to the next in a `switch` statement.

- **continue;**
  - Execution starts the next iteration of the innermost enclosing `do`, `for` or `while` loop.
  - Useful for skipping the body of a loop in an exceptional situation.

- **goto label;**
  - May be useful in exceptional situations, for example, when breaking out of multiply-nested loops.
  - The `label` needs to be part of a full statement (which could be a null statement).
  - Execution can only jump to a target within a single function body.
  - Example:
    
    ```plaintext
    Start:
    i = 0;  // this is part of the labelled statement
    ...
    if ( i > 200 ) goto Start;
    ```
7 Functions

- Functions are a form of procedural abstraction, so we don’t have to deal with too much detail at any one point in time.

- The idea is to encapsulate an operation or procedure, give it a name, and subsequently use this operation in other parts of the program simply by using its name.

- There is a special function called “main” which is entered when your program starts running (possibly after power-on or some other reset condition for the AVR MCUs).

- All other functions are called directly or indirectly from main except interrupt routines which are called directly by the MCU hardware. (More on that in Section 13.)

- **type of the function**
  - Functions return a value of their declared type whenever they are used.
  - The value returned may be ignored (implicitly discarded). If you wish to always do this, it is probably best to declare the type of the function to be `void` to indicate that no value is returned.

- **definition of the function**
  - declares the name and type of the function and the argument list.
  - provides the body of the function as a compound statement.

- **formal parameters** (or arguments)
  are the names used inside the function to refer to its arguments.

- **actual parameters** (or arguments)
  are the expressions or values used as arguments when the function is actually called.

- If we wish to refer to the function before defining it, we need to declare it to the compiler by specifying its **prototype**.
  In the function prototype, we don’t need to specify the formal parameter names, just their type, however, specifying the names improves readability of your code.

- **function body**
  - The body of a function is always a compound statement.
  - New variables may be declared within this compound statement (or block). These will be local to the function and not be visible to code outside the function.
  - If any new variable has the same name as any global variable, the local variable will hide the global variable.
  - Formal parameters are effectively local variables.
• **passing arguments** (pass-by-value mechanism)
  
  – When a function is called, any arguments that are provided by the caller are simply treated as expressions.
  
  – The value of each expression is used to initialize the corresponding formal parameter.
  
  – The formal parameters behave the same way as other local variables within the function. Any changes to them are not seen outside the function.

• **scope of variables**
  
  – The scope of a variable is the area of the program code in which the variable is valid (or is visible).
  
  – A *global* variable is one that is declared outside any function and is valid anywhere (within the source code module).
  
  – A *local* variable has a scope that is limited to the block in which it is declared and cannot be accessed outside of that block. (*A block is a section of code enclosed in curly braces { ... }*).
8 Points-to and addresses-of variables

unsigned int i, j;    // defines 16-bit variables
unsigned int *p;    // defines a pointer variable

The variable p will contain the address of an unsigned int variable (as opposed to the value of the int).

Conceptual model of the data memory.

The memory locations will have numerical addresses as seen in the AVR datasheet. The names of the variables are shown to the right of the memory locations that they label. The question marks in the memory locations indicate undefined values (presently).

// assign some values
i = 1;
j = 3;
p = &i;

p now points to the variable i because p contains the address (numerical location in data memory) of the variable i.
& is the address-of operator.
To change the value of a variable, given its address, we use the \( * \) (prefix) operator. It is equivalent to “what is pointer to” and is sometimes called the “dereference” operator.

\[
\text{\*p} = 4; \\
j = (*p);
\]

The expression \( \text{\*p} \) evaluates to 4. Also, we may have more than one pointer variable pointing to any particular memory location.

9 Functions revisited: pass-by-reference

Sometimes we would like a mechanism to affect the variables used as actual parameters to a function. Even though values only are passed through to the formal arguments of a function, we may pass the address of any variable that we want to affect from within the function.

```c
void double_it( int *v )
{
    (*v) = (*v) * 2;
    return;
}

int main( void )
{
    int i = 4;
    ...
    double_it( &i );
    ...
}
10 Arrays

- An array is a collection of variables with identical type and a single name.

- General definition of a single-dimensional array:
  
  \[
  \text{type variable}[\text{size}];
  \]

- Example:
  
  ```c
  int data_list[3];
  ```

  ![Array Diagram]

  Although the size may be an expression, it must be evaluated at compile time. Note also that the array elements start at index 0 and are numbered up to \( n - 1 \) where \( n \) is the number of elements. We can consider this index as an offset from the first element in the array.

- Initialize each element:

  ```c
  int i;
  for ( i = 0; i < 3; i += 1 ) {
      data_list[i] = i + 1;
  }
  ```

  ![Initialised Array Diagram]
- Use the same notation to access the elements:

```c
int sum = 0;
for ( i = 0; i < 3; i += 1 ) {
    sum += data_list[i];
}
```

### 10.1 Pointers and arrays

- The name of the array itself is a pointer to the first element (at index 0).

- Example:

```c
int data_list[3];
int *p;
...
p = data_list;
   // or
p = &data_list[0];
...
*p = 5;
```

We could use “pointer arithmetic” to step through the array.

```c
for ( p = data_list; p <= &data_list[2]; ++p ) *p = 0;
```
10.2 Strings

- Strings are created out of arrays of characters with the last character being the special null character.

- Example:

```c
char name[10];
...
name[0] = 'P'; name[1] = 'e'; name[2] = 't';
```

- Alternatively:

```c
#include <string.h>
...
strcpy(name, "Peter");
```

11 The C preprocessor

- The preprocessor is essentially a specialized text editor which changes your source code before passing it (the transformed source code) onto the compiler.

- All preprocessor directives (also known as commands or macros) begin with a sharp “#” character in the first column and terminate at the end of the line unless they are continues by having a backslash “\” character as the very last character on the line.

- Be careful that the syntax of the preprocessor is different to the C language. You cannot use C constructs as preprocessor directives.

- include files
  - The include directive allows the program to use source code from another file.
  - Examples
    ```c
    #include <avr/io.h>
    #include "my_defs.h"
    ```
    The angle brackets tell the preprocessor to look in the system area for the include file. The double quotes tell the preprocessor start looking in the current area.
    - We can include any file we like but, usually, the file will contain function prototypes and parameter definitions.

- replacement directive
  - General form:
    ```c
    #define name substitute-text
    ```
Examples:
#define BIT(n) (1 << (n))
...
#define NDIM 10
int x[NDIM];
...
for ( i = 0; i < NDIM; ++i ) x[i] = 0;

The convention is to use capital letters for the name of the definition. Also, when parameters are passed to a definition, use parentheses substitute text in order to be safe. The definition may be used in all sorts of places.

- We can also remove a definition:
  #undef NDIM

- Definitions remain in force until they are removed or until the end of the source code file is reached.

- The AVR libc writers have provided definitions for all of the useful I/O registers, for example:

  #include <avr/io.h>
  ...
  DDRD = 0xFF;

- **conditional compilation**

  We may have multiple versions of the source code in one file.

  #define DEBUG
  ...
  #ifdef DEBUG
  ...
  code for the debugging version
  ...
  #else
  ...
  code for the production version
  ...
  #endif
12 Example 4: Using the analog-to-digital converter

- This example demonstrates the construction of a program from several modules. Add source files to the project “tree-view” in the top-left of the IDE.
  - adc_demo.c
  - my_adc.c
  - my_serial.c
  - my_timer.c
- The header files are automatically put under the heading “External Dependencies” when you next build the project.

12.1 C source code

main module

```c
#include "my_timer.h"
#include "my_serial_port.h"
#include "my_adc.h"

int main(void)
{
    unsigned int adc_value, old_value;
    setup_timer0();
    setup_uart();
    setup_adc();
    adc_value = 0;
    while (1) { // never-ending loop
        reset_timer0(180); // Now, we have 50ms to do our work.
        // Attend to all of our chores...
        old_value = adc_value;
        adc_value = get_adc_value(0);
        if ( old_value != adc_value ) {
            // Only send value when that has been a change.
            send_number(adc_value);
            send_character(\n'); // new line
            send_character(\r'); // carriage return
        }
        // Hopefully, there is some time to wait for the timer to overflow.
        wait_for_timer0();
    }
    return 0; // we should never get here.
}
```
ADC module

// my_adc.c
// Functions to access the analogue-to-digital converter.
//
// PJ, 12-Feb-2008

#include <avr/io.h>

void setup_adc(void)
{
    ADCSRA &= ~(1 << ADATE); // select single sample mode
    ADCSRA |= (1 << ADEN);   // enable ADC hardware
    // Set prescaler division factor to 8.
    ADCSRA |= (1 << ADPS0);
    ADCSRA |= (1 << ADPS1);
    ADCSRA &= ~(1 << ADPS2);
    return;
}

unsigned int get_adc_value(unsigned char channel_id)
{
    unsigned char low_byte, high_byte;
    if (channel_id > 5) channel_id = 5;
    ADMUX = channel_id; // start conversion
    while ((ADCSRA & (1 << ADIF)) == 0) /* wait */;
    low_byte = ADCL; // Datasheet says to read ADCL first.
    high_byte = ADCH;
    ADCSRA |= (1 << ADIF); // clear the conversion-complete flag
    return (int)high_byte * 256 + low_byte;
}
# Timer module

// my_timer.c
// Functions to set and wait for hardware timers.
// PJ, 12-Feb-2008, updated for ATmega88

#include <avr/io.h>

void setup_timer0(void)
// We will use the 8-bit timer/counter 0 without interrupts
// and we will slow things down by prescaling the system clock.
{
    TCCR0B |= (1 << CS02) | (1 << CS00); // prescale CK/1024
    return;
}

void reset_timer0(unsigned char counts)
// With a clock frequency of 3.686 MHz on the STK500 and
// a 1024 prescaling factor, we expect a count of 36 to result
// in timer1 overflow after 10 milliseconds.
{
    TIFR0 |= (1 << TOV0); // clear the overflow bit by writing logical 1
    TCNT0 = 0xFF - counts; // set the starting count
    return;
}

void wait_for_timer0(void)
{
    while ((TIFR0 & (1 << TOV0)) == 0) /* do nothing */;
    return;
}

//------------------------------------------------------------------------------

void setup_timer1(void)
// We will use the 16-bit timer/counter 1 without interrupts
// and we will slow things down by prescaling the system clock.
{
    TCCR1B |= (1 << CS12) | (1 << CS10); // prescale CK/1024
    return;
}

void reset_timer1(unsigned int counts)
// With a clock frequency of 3.686 MHz on the STK500 and
// a 1024 prescaling factor, we expect a count of 360 to result
// in timer1 overflow after 100 milliseconds.
{
    TIFR1 |= (1 << TOV1); // clear the overflow bit by writing logical 1
    TCNT1 = 0x0FFF - counts; // set the starting count
    return;
}

void wait_for_timer1(void)
{
    while ((TIFR1 & (1 << TOV1)) == 0) /* do nothing */;
    return;
}
Serial port module

// my_serial_port.c
// Functions to send characters via the hardware USART.
//
// PJ, 12-Feb-2008

#include <avr/io.h>

// Assume that we run the STK500 clock at full speed.
#define FOSC 3686400
#define BAUD 9600
#define MYUBRR FOSC/16/BAUD - 1
// See table 18–1 of datasheet for the baud-rate register formula.

void setup_uart(void)
{
  UBRR0 = MYUBRR; // a value of 23 for 9600 baud with FOSC=3.686 MHz
  UCSR0B |= (1 << TXEN0); // connect the transmit to PD1
  return;
}

void send_character(unsigned char c)
{
  // Note the null statements in the bodies of the following loops.
  while ((UCSR0A & (1 << UDRE0)) == 0) /* wait until empty */;
  UDR0 = c; // send the character
  while ((UCSR0A & (1 << TXC0)) == 0) /* wait until complete */;
}

unsigned char hex_repr(unsigned char n)
// Return a hexadecimal character representing the binary value.
{
  n &= 0x0F; // keep only 4 bits, just in case the user has sent more
  if ( n < 10 ) {
    return n + '0';
  } else {
    return (n-10)+'A';
  }
}

void send_number(unsigned int n)
// Send a 4-digit hexadecimal representation of the number
// to the serial port, starting with the most significant digit.
{
  unsigned char nybble, i;
  for ( i = 0; i < 4; i++ ) {
    nybble = (n & 0xF000) >> 12; // get most significant digit
    send_character(hex_repr(nybble));
    n = n << 4; // move remaining bits left
  }
}
13 Interrupts

- The hardware may signal that it needs servicing via an interrupt request\(^1\).

- If the processor has its interrupts enabled, it suspends whatever it is doing and executes the corresponding interrupt service routine before resuming work on the task code.

- AVR MCUs have a number of interrupt sources, each enabled by one enable bit and each with an entry in the interrupt vector table in the lowest part of program memory. Entries in the table are typically jump instructions.

- When an interrupt occurs, the Global Interrupt Enable bit is cleared and all interrupts are disabled. The program counter is vectored to the actual interrupt vector in order to jump to the interrupt handling routine. Hardware clears the corresponding flag that generated the interrupt.

- See section 6.26 in the AVR-libc reference manual for details of setting up an interrupt handler. For example:

```c
#include <avr/interrupt.h>
...
ISR(TIMER1_OVF_vect)
{
  // Our interrupt handling code here
  // for dealing with the timer overflow signal.
}
```

We are provided with a macro to define the ISR and a table of interrupt vector names for our particular processor.

- Interrupt service routines (ISRs) must save the context of the task code and restore that context when they are finished. The C compiler usually handles that detail for us by writing suitable code at the start and at the end of our code in the ISR.

- Usually, it is neither desirable nor possible to do all of the work in an interrupt routine so the routine needs to signal the task code to do follow-up processing. To do this communication, the ISR shares one or more variables.

- Once one or more variables are shared, we have to be careful because, while in the task code, the interrupt routines may change the data at any instant. This is the classic data-sharing problem.

- It may be difficult to identify when programming in C because single statements in C may be translated to several assembler instructions and the interrupt event may occur within these assembler statements.

\(^1\)Some of these notes from Simon’s text [5].
• One cure for the data-sharing problem is to disable interrupts in “critical sections” of task code. (These are sections of code that must not be interrupted for the system to work properly.)

• Be sure to use the volatile keyword to indicate that the content of a variable may change because of interrupts or other things that the compiler doesn’t know about.

    static volatile int my_flag;

• **Interrupt latency** is the amount of time it take a processor to respond to an interrupt. It is determined by:

    1 The longest period of time during which the interrupt is disabled.
    2 The period of time that it takes to execute routines for higher-priority interrupts.
    3 The length of time that it takes the MCU to stop what it is doing (in the task code), do the necessary book-keeping and start executing code in the ISR. Look this up in the AVR datasheet.
    4 The length of time it takes the ISR to save the context of its just-left task and do the work that constitutes the “response” to the interrupt signal.

Item 2 leads to the desire to make interrupt routines short.
13.1 Example 5: Using interrupts with a hardware timer.

// flashleds3.c
//
// The application doesn't do much; it just lights one LED at a time
// across the bottom of the STK500 board.
// It does, however, demonstrate the use of a hardware timer
// with interrupt to provide a regular period for the display update.
//
// PJ, 28-Aug-2007
// 21-Feb-2008 updated for ATmega88

// get the processor-specific definitions
#include <avr/io.h>
#include <avr/interrupt.h>

// We will use the 16-bit timer/counter 1 with overflow interrupt
// and we will slow things down by prescaling the system clock.
// With a clock frequency of 3.686 MHz on the STK500 and
// a 1024 prescaling factor, we expect a count of 360 to result
// in timer1 overflow after 100 milliseconds.
unsigned int start_count = 360;

// We also need some other data that is preserved between updates
// of the display.
unsigned char which_bit;

void setup_timer1(void)
{
    // Normal port operation, OC1A and OC1B disconnected
    // Leave bits COM1A1 through COM1B0 of TCCR1A as default zero values
    // prescale CK/1024
    TCCR1B |= (1 << CS12) | (1 << CS10);
    TIFR1 |= (1 << TOV1); // clear the overflow flag
    TIMSK1 |= (1 << TOIE1); // enable overflow interrupt
    return;
}

ISR(TIMER1_OVF_vect)
// This routine is executed on the Timer 1 overflow interrupt.
// Note that it is not called directly by any of our other code.
// Also note that the body of this service routine should not
// do much work because the interrupts are currently turned off.
{
    // overflow bit is already cleared by hardware
    TCNT1 = 0xFFFF - start_count; // reset the counter
    // Decide the next LED to turn on then make it happen and leave
    which_bit += 1;
    if (which_bit > 7) which_bit = 0;
    // Turn on a LED by writing 0 to its corresponding MCU pin
    PORTD = ~(1 << which_bit);
}

int main(void)
{
    DDRD = 0xFF; // all output so that we can turn on the LEDs
    setup_timer1();
    TCNT1 = 0xFFFF - start_count; // reset the counter
    which_bit = 0;
    sei();

    // At this point, we can get one with whatever work needs to be done
    // and the Timer1 interrupt will regularly be serviced by the ISR.

    while (1) /* do nothing */ ;
    return 0; // we should never get here.
}
13.2 Example 6: Using interrupts to count button presses

```c
// button-interrupt.c

// Demonstration of using external interrupts on the AVR.
// Port B is output with the 10-pin jumper connected to LEDs on STK500.
// It shows the current count of button presses.
// Note that only LED0..LED5 are connected for ATmega88.
// Port D is input with PD2=INT0 (count up) and PD3=INT1 (count down).
// Use 10-pin jumper to connect to buttons on STK500.

#include <avr/io.h>
#include <avr/interrupt.h>

unsigned char count = 0;

ISR(INT0_vect){
    count++;
    PORTB = ~count;
}

ISR(INT1_vect){
    count--;
    PORTB = ~count;
}

int main() {
    // Set input and output
    DDRD &= ~(1 << PD2); // INT0 as input
    DDRD &= ~(1 << PD3); // INT1 as input
    DDRB = 0xFF; // All output
    PORTB = 0xFF; // turn off LEDs on STK500

    // Enable specific external interrupts
    EIMSK |= (1 << INT0); // INT0 as input
    EIMSK |= (1 << INT1); // INT1 as input
    // Set up interrupt sense control so that falling edges
    // on INT0 and INT1 generate interrupt requests.
    EICRA |= (1 << ISC01); // INT0
    EICRA |= (1 << ISC11); // INT1
    sei();

    // All of the interesting code is now in the ISRs so
    // we just wait around for the hardware to make requests.
    while (1) {
        return 0;
    }
```
Software architectures of embedded systems

• It’s all about system response. For the source of these note, see section 5 of Simon’s text [5].

• How we structure our software depends on how much the MCU needs to do and how strict the time requirements are.

• Options discussed by Simon [5] are:
  – Round-Robin
  – Round-Robin with interrupts
  – Function-queue scheduling
  – Real-time operating system

14.1 Round-Robin

• This is the simplest architecture (and was used in the early examples).

• No interrupts

• No shared-data issues

• Structure of task code:

  ```c
  int main( void )
  {
    // initialize hardware
    while ( 1 ) {
      if ( device A needs service ) {
        // service device A; handle data transfer to or from device
      }
      if ( device B needs service ) {
        // service device B; handle data transfer to or from device
      }
      // ... more devices ...  
    } // end Round-Robin Loop
    return 0;
  } // end main
  ```

• Provided that we have slack in our processing needs, we can make the Round-Robin loop regular by adding a wait-for-timer-tick at the end of the loop.

• If any device needs a response time less than the time needed to do all of the tasks, the system won’t work properly.

• If any task does a large amount of processing, the system response will be poor.

• We may be able to improve response time for one device by testing the device more than once in the loop: ABACADA...
The system is fragile. After changing the task sequence to meet response needs, one change or addition can mess it up again.

14.2 Round-Robin with interrupts

- Interrupt routines deal with the urgent needs of any devices and set flags.
- The main (task) loop polls the flags and does follow-up processing requested by the interrupts.
- Structure of the code:

```c
char flag_dev_A = 0;
char flag_dev_B = 0;
...
void ISR( device_A )
{
  // service device A
  flag_dev_A = 1; // to signal follow-up processing
}
void ISR( device_B )
{
  // service device B
  flag_dev_B = 1; // to signal follow-up processing
}
...
int main( void )
{
  while ( 1 ) {
    if ( flag_dev_A ) {
      flag_dev_A = 0;
      // Do follow-up processing for device A.
    }
    if ( flag_dev_B ) {
      flag_dev_B = 0;
      // Do follow-up processing for device B.
    }
    ...
  } // end Round-Robin Loop
  return 0;
} // end main
```

- All processing that is in the interrupt routines gets higher priority than the task-loop code.
- This ability to set priority for a piece of code is an advantage but it comes with the cost of having to deal with shared-data.
References


C Reference Card (ANSI)

Program Structure/Functions

- function prototype statements
- variable declarations
- function definition statements
- return value;
- */
- include library file
- replacement text
- replacement macro
- define name (c) test
-umed name
- define name for data type
- struct
- void
- local variable declarations
- static
- positive or negative

Data Types/Declarations

- character
- int
- real number (single, double precision)
- short
- long
- double long
- non-negative modulo
- pointer to int
- float
- enumeration constant
- constant (read-only) value
- int
- static
- void
- structure
- size of a data type (type is size)

Initialization

- type name
- type name = value
- char name [ ];
- define name (c) test

Constants

- suffix: long, unsigned, float
- exponential form
- prefix: octal, hexadecimal
- character constant (char, octal, hex)
- newline, cr, tab, backspace
- special characters
- string constant (ends with \0)

Pointers, Arrays & Structures

- declare pointer to type
- declare function returning pointer to type
- declare pointer to function returning type
- array member of pointed-to structure
- member of structure from template
- create structure
- create new name for data type
- size of a data type (type is size)

Operators (grouped by precedence)

- struct name
- struct member
- struct member through pointer
- struct member through pointer (pointer to member)
- increment, decrement
- plus, minus, logical not, bitwise not
- indirection via pointer, address of object
- cast expression to type
- expression
- multiplication, modulo (remainder)
- addition, subtraction
- left, right shift
- relational comparisons
- equality comparisons
- and
- or (inclusive or exclusive)
- logical and
- logical or
- conditional expression
- expression evaluation separator
- unary operators, conditional expression and assignment operators
- group right to left, all others group left to right

Flow of Control

- statement terminator
- block delimiters
- if statement
- if else if statement
- else else if statement
- while statement
- for statement
- do statement
- goto label
- label
- return value from function
- return expr

Flow Constructions

- if statement
- if else if statement
- else if statement
- else statement
- while statement
- for statement
- do statement
- switch statement
- case const;
- default; statement

ANSI Standard Libraries

- include file
- <stdlib.h> <stdio.h> <string.h> <time.h>
- <locale.h> <math.h> <setjmp.h> <signal.h> <stdarg.h>
- <assert.h> <ctype.h> <errno.h> <float.h> <limits.h>
- <assert.h> <ctype.h> <errno.h> <float.h> <limits.h>

Character Class Tests

- alphabetic
- alphanumeric
- control character
- decimal digit
- printing character (not incl space)
- lower case letter
- printing character (incl space)
- printing character (spaces)
- upper case letter
- hexadecimal digit
- convert to lower case
- convert to upper case

String Operations

- length of a
- a is a string, ca, ct are constant strings
- copy char to a
- convert a ct to
- compare a to ct
- only first a char
- pointer to first c in a
- pointer to last c in a
- compare a char of a with ct
- compare a char of ct to a
- compare a char of a with a (may overlap)
- convert to lower case
- convert to upper case
- expression evaluation separator
- unary operators, conditional expression and assignment operators
- group right to left, all others group left to right

© 2017 Joseph H. Silverman