ECE/CSE 412

Lab #2

Report (85 Points)

Demo (15 Points)

Algorithm Implementation and AVR-GCC Analysis Using AVR Assembly

Joe Maloney
ECE 412

Abstract

Various methods for organizing AVR assembly programs were explored using implementations of division and sorting algorithms, including quicksort and bubblesort. Execution times of the tested sorting algorithms were tested to explore the computational complexity of the two algorithms when implemented in AVR assembly and running on a ATmega328PB, with 1600 samples collected. Additionally, the output of the AVR-GCC compiler was explored via analysis of listing files generated by AVR-GCC when running within Microchip Studio.

Body

Division of 8-Bit Integers

The Division Algorithm

Division of 8-bit integers was performed using a repeated subtraction algorithm. Program flow was controlled as shown in figure 1 below.

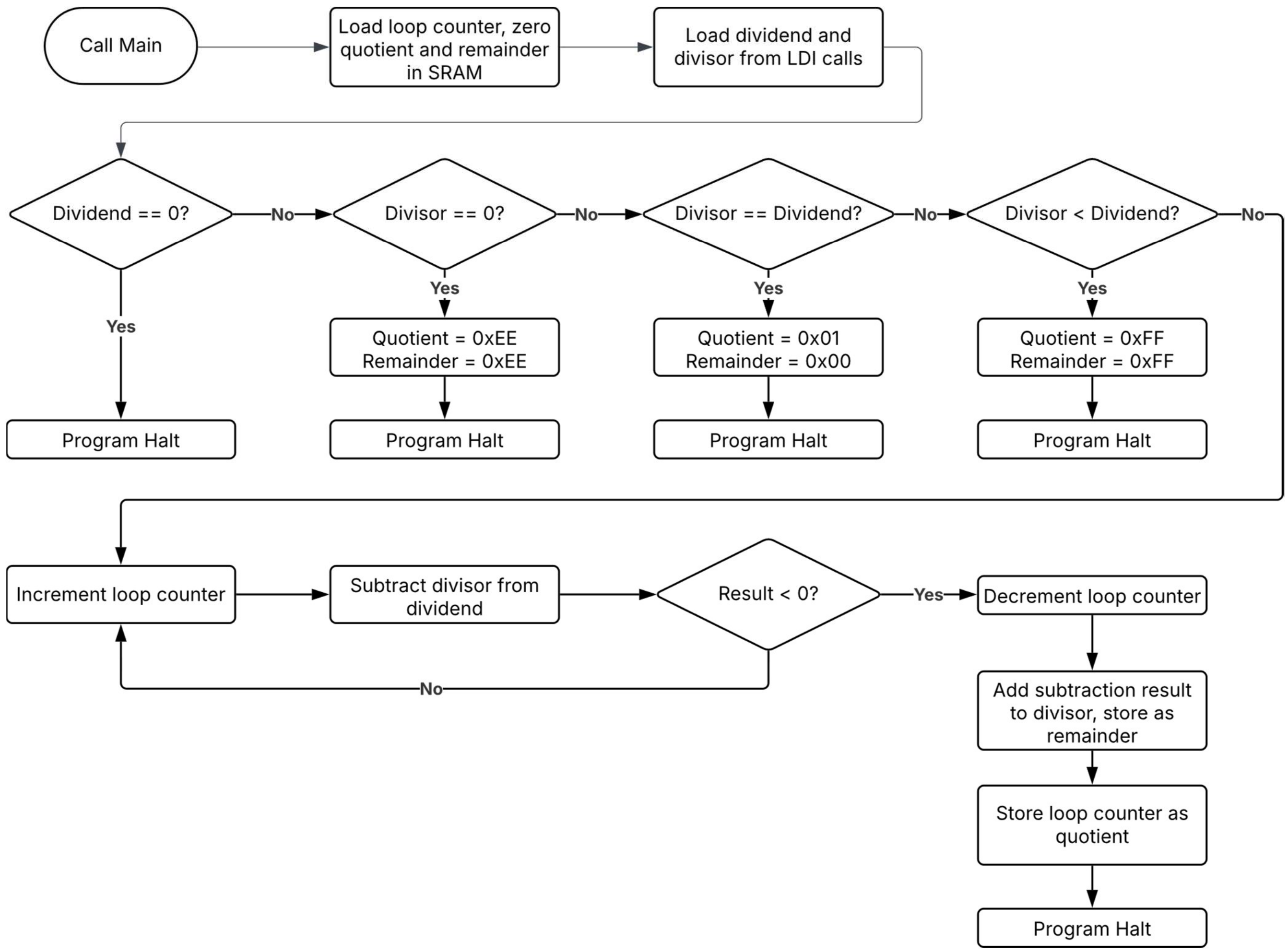

Figure 1: Division Flow Control

The algorithm stores and initializes variables in multiple ways. The quotient and remainder are computed into the registers r0 and r30, then stored at sram locations 0x0100 and 0x0101 respectively. The count is a temporary variable used as a loop counter and stored in r0. It is initialized to the value at SRAM address 0x0100 at the start of the program by the instruction "LDS r0 , count", where count is an assembly-time constant equal to 0x0000. The value stored by the SRAM at this location is undefined at the time it is read to initialize the loop counter. This will result in incorrect division results when not running on the Microchip Studio simulator (where registers and SRAM are zero-initialized), or without first zero-initializing the value at SRAM address 0x0100. The divisor and dividend are stored within r31 and r30 respectively, during program execution. Divisor and dividend are initialized from values encoded into LDI instructions on lines 36 and 37. The division algorithm works by first checking a few invalid combinations of input parameters and returning error codes if the given parameters for divisor and dividend would result in an error. The case of divisor==dividend is also checked, skipping the loop and outputting the result directly. Pseudocode of the implemented algorithm is shown below.

```
let divisor = 0
let dividend = 0
let count = 0
let quotient = 0
let remainder = 0
if dividend == 0:
      return
if divisor == 0:
      quotient = 0xEE
      remainder = 0xEE
      return
if dividend == divisor:
      quotient = 1
      remainder = 0
      return
if dividend < divisor:
      quotient = 0xFF
      remainder = 0xFF
      return
do:
      dividend = dividend – divisor
      count += 1
while (dividend > 0)
count -= 1
quotient = count
remainder = dividend + divisor
return
```

Division Function Calls

Function calls used to implement the division algorithm were implemented in two distinct ways. The first was by having the main function perform all calls to the division subroutines directly. The main function called the init, getnums, test, and divide functions directly. These four functions are the subroutines of the division algorithm. When function calls were used this way, the size of the stack was either 2 bytes during a call to a subroutine, or 0 in the main function. The 2 byte stack during a subroutine was used to store the return address of the subroutine to main. The other method of implementing the division algorithm was by having each subroutine call the next subroutine before returning from the current subroutine. This resulted in only one call within main, and a maximum stack size of 8 bytes. The top 6 bytes of the stack stored the return addresses to the subroutines that were previously called by a parent subroutine. The bottom 2 bytes of the stack stored the return address of the initial call to the first division subroutine from main. The version of the division program where all subroutines are called from main is labeled *Division, Single Parent Caller*, as shown in the software section. The version using nested subroutines is labeled *Demonstration*

Code, and executes prior to the sorting algorithm later in the program.

Look Up Table

Look Up Table Implementation on the ATmega328PB

To examine the behavior of a look up table on the ATmega328PB, a 20 value Celsius to Fahrenheit converter was implemented using a look up table located in the flash memory of the ATmega328PB. This was done by writing a contiguous array of pre calculated Fahrenheit values to the flash memory. Each value was the Fahrenheit equivalent of the prior value plus one degree centigrade, starting at 32 degrees Fahrenheit. These were stored in the flash memory of the ATmega328PB, meaning that 2 8-bit values were stored at each address of the flash memory array, one at the low byte and one at the high byte. To calculate the Fahrenheit equivalent of a Celsius temperature, the Z pointer was initialized to the starting address of the array, and the temperature to be converted was added to the Z pointer before reading the array at the new value of the Z pointer with the LPM instruction. This works by selecting the word of the flash memory to be read with the first 15 bits of the Z pointer, and then selecting the high/low byte with the last bit of the Z pointer. Because the values in the array are 1-byte large, the alternating selection of high/low byte by adding one to the Celsius results in correct conversions.

Sorting

Sorting Implementations

Sorting on the ATmega328PB was implemented using two different algorithms, bubblesort and quicksort. Both algorithms performed sorting on an array of 2-byte unsigned integers. The implementation of bubblesort examined in this study was adapted from *Application Note AVR220*. Bubblesort has a theoretical time complexity of $O(n^2)$, and an auxiliary memory usage of O(1). The memory usage of the implementation of bubblesort evaluated in this study was O(1), as no declarations of variables or recursive function calls occur during the main loop of the bubblesort function. Pseudocode for the implementation of bubblesort is shown below.

```
//arr – array of elements to be sorted
//n – length of the array
bubblesort(arr , n)
	let i = n
	let j = 2
	let k = 1
	while (i > n):
		while (j > i):
			if (arr[k] >= arr[j]):
				swap(arr[k] , arr[j])
			j += 1
			k += 1
		i -= 1
		j = 1
```

```
        k = 2
    return
```

The other sorting algorithm evaluated on the ATmega328PB was quicksort. Pseudocode for the implementation of quicksort used is shown below.

```
//arr - array of elements to be sorted
//n - length of the array
quicksort(arr , n)
    let i = 2
    let j = 1
    pivot = arr[1]
    while (i < n):
        if (arr[i] < pivot):
            swap(arr[j+1] , arr[i])
            j += 1
        i += 1
    swap(arr[1] , arr[j])
    quicksort(arr[1:(j-1)] , j-1)
    quicksort(arr[(j+1):n] , n-j)
    return
```

The implementation of quicksort used has a theoretical time complexity of $O(n\log_2 n)$, and auxiliary memory usage of O(n) in the worst case.

Sorting Memory Usage

The dataset to be sorted was stored in the following format on the SRAM of the ATmega328PB:

```
0x0100 : uint16 - dataset size (n)
0x0102 : uint16 - data_1 (first number in array)
0x0103 : uint16 - data_2 (second number in array)
.. .. ..
.. .. ..
.. .. ..
0x0XXX : uint16 - data_n (last number in array)
.. .. ..
.. .. ..
.. .. ..
0x08FF : bottom of stack (grows downwards)
```

It is important to not overrun the stack into the array of data stored in the SRAM of the ATmega328PB. For the implementation of quicksort examined in this study, each call to quicksort resulted in the pushing of a stack frame 6 bytes large onto the stack. If the worst case O(n) memory usage was encountered, the maximum number of values that could be sorted within to 2KB SRAM is:

$$\frac{2048}{d+b} - 2 = 254$$

$$d = number\ of\ bytes\ in\ quicksort\ stack\ frame\ (6)$$

$$b = number\ of\ bytes\ in\ data\ element\ (2)$$

This is however only for the worst case where the array is already sorted, and occurs because the first element in the array is always selected for the pivot. Because the quicksort algorithm is only tested on random datasets, a more realistic average memory usage can be calculated with:

$$6\,log_2\,n + 2n + 1 = m$$
$$n = number\ of\ elements\ in\ dataset$$
$$m = number\ of\ bytes\ of\ memory\ usage$$

With 2KB of SRAM, the average number of elements that could be sorted before overflowing the stack onto the array is ≈ 943 elements. The maximum number of elements tested was 800, meaning that a stack overflow was exceedingly unlikely during testing. During the testing of the two sorting algorithms, the 2KB of memory on the ATmega328PB was organized in the following manner:

```
0x0100   : uint16 - dataset size (n)
0x0102   : uint16 – data_1 (first number in array)
0x0103   : uint16 – data_2 (second number in array)
..  ..  ..
..  ..  ..
..  ..  ..
0x0XXX   : uint16 – data_n (last number in array)
..  ..  ..
..  ..  ..
0x0ZZZ-4 : uint16 – upper array length (big endian) - - - - -
0x0ZZZ-2 : uint16 – ending pivot address (big endian)       - quicksort stack frame
0x0ZZZ   : uint16 – return address- - - - - - - - - - - - - -
..  ..  ..
0x08FE   : uint16 – bottom of stack
0x08FF   : uint16 – test count (big endian)
```

Only the quicksort algorithm required the pushing of stack frames onto the stack for the recursive calls to quicksort. During the testing of bubblesort, the only elements pushed to the stack were calls to helper functions, and never was there a call stack during bubblesort more than 2 functions deep.

Testing Methodology

In order to observe the growth in execution time of the two tested sorting algorithms as the dataset size (n) increased, an automated testing solution was implemented using a host PC running a python script to generate and transmit random datasets to the ATmega328PB, as well as time the execution of the sorting algorithms on the MCU. Complimentary AVR assembly code was executed on the ATmega328PB, enabling it to receive a number of datasets from the host PC and communicate when sorting was stopped/started on the MCU. An Xplained Mini development board was used to host the ATmega328PB for testing. This board features a USB to UART converted that was used for communication with the host PC. First, the python script on the host PC would begin listening for the MCU to send a ready signal over the UART. When this signal was received, the python script would then send the test count (the number of random datasets that would be sorted) to the MCU, and the MCU would acknowledge that this count was stored and received. After this acknowledgement, the host PC would then transmit the size of the first random dataset to be received by the

MCU. The MCU would acknowledge the reception of this number, prompting the host PC to begin sending the randomly generated dataset of the same size. Originally, the MCU would acknowledge each byte of dataset data received, however this communication protocol was abandoned after testing without the acknowledgements showed equal data integrity, but at a much higher overall speed. After receiving the dataset, the MCU would transmit a test start code to the host PC, and the host PC would begin a timer for the current test. The MCU would then sort the test data, and communicate to the host PC that the test was completed, prompting the host PC to stop the timer. The host PC would then save the time, and send the next dataset. The python source code for the host PC is available at *https://github.com/joemowed/CSE412Lab2*. A visual representation of this communication protocol is shown in figure 2 below.

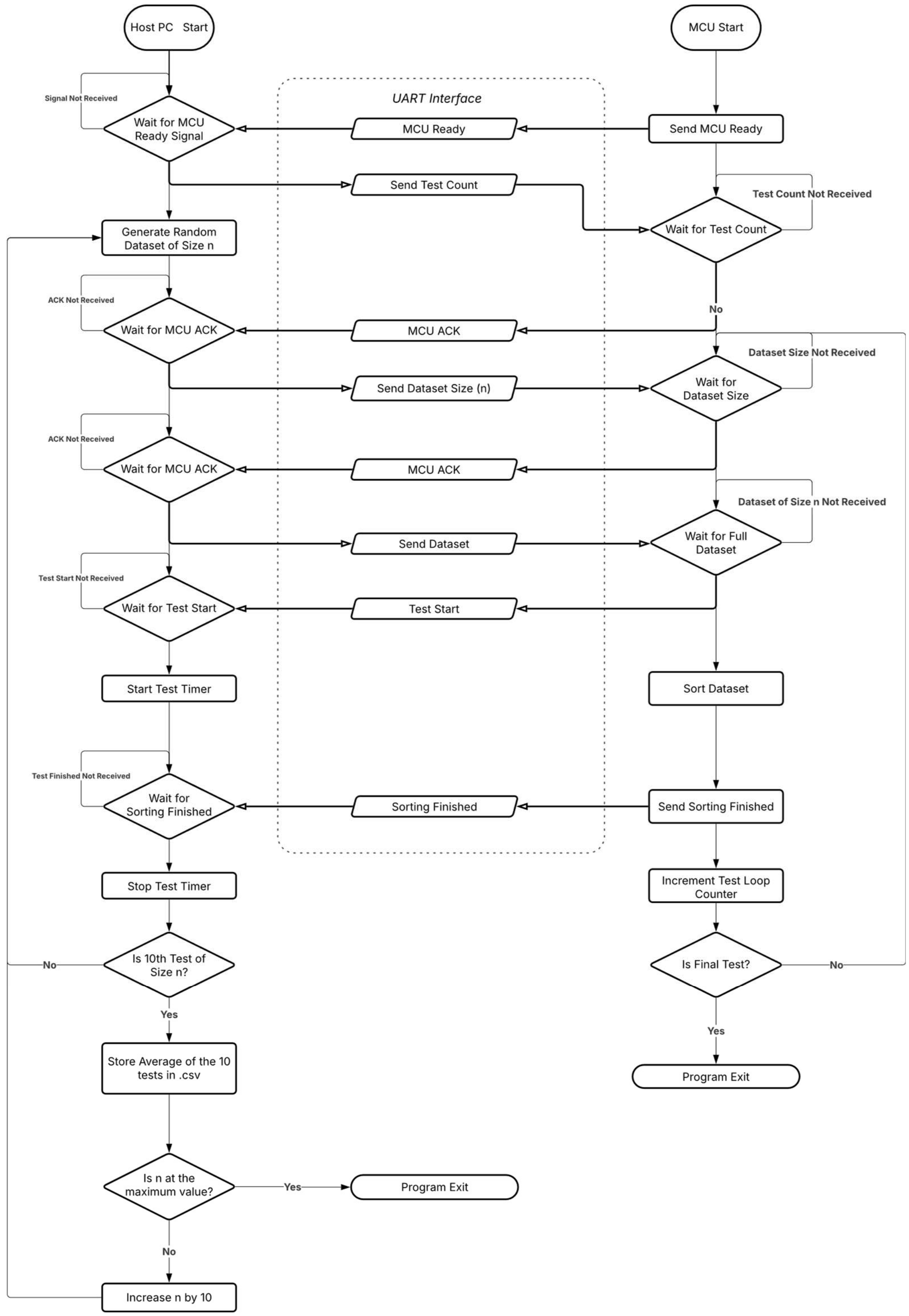

Figure 2: Testing Communication Protocol

Dataset Selection and Data Collection

Datasets varied in size from 10-800 elements, and were tested by increasing the size of the dataset by 10 elements at a time (n=10, n=20, n=30…, n=800). Each dataset of size n was evaluated with 10 random datasets of that size, and the average of those 10 runs was recorded. The data from the 1600 samples, (10 tests per n, 80 different sizes of n tested on 2 different algorithms) is available in appendix A. Graphing the data shows the predicted execution times of $O(n^2)$ and $O(n\log_2 n)$ for bubblesort and quicksort, respectively.

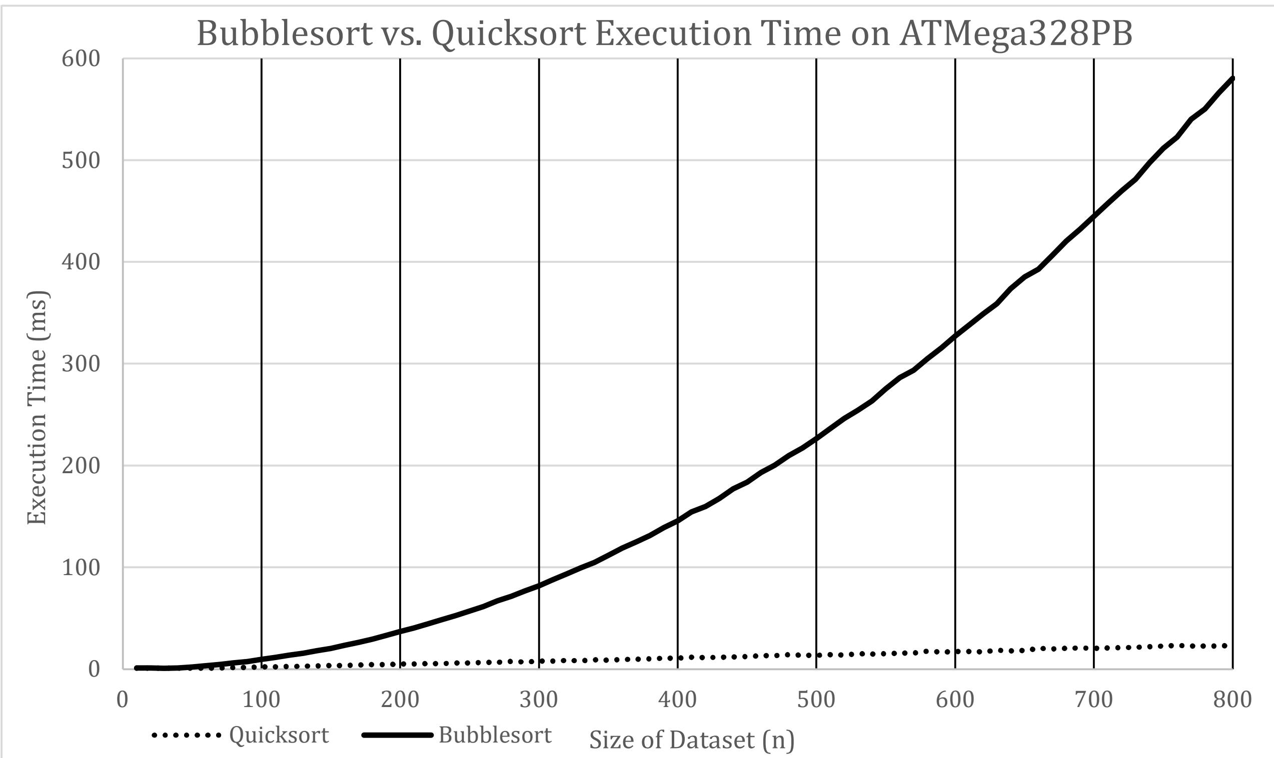

Figure 3: Bubblesort vs. Quicksort Execution Time, Milliseconds

Compilation of C Code with AVR-GCC

Examined Functions

The process of compiling C code into AVR machine code was examined using the AVR-GCC compiler embedded within Microchip Studio. The differences within the generated machine code were examined for compiling the same C code, only changing the data types of the variables involved. 3 global variables alongside a main function were created for examination of each of the 4 types of integers. The 4 types of integers tested were unsigned char, signed char, unsigned int, and signed int. Two of the global variables, global_B and global_C, were initialized to the values 1 and 2 respectively. The third variable, global_A, was not explicitly initialized to any value. The main function used for examination of all 4 data types added the two initialized variables together and stored the result at the location of the uninitialized variable.

Differences in the Listing

Each version of the program was compiled and the resultant listing file examined for differences between the versions. The signed/unsigned pairs of char and int showed no differences, however by examining the listing files there was a difference between the char and int data types. The char data types were 8-bits large and required a single add instruction, while the int data types were 16-bits large and required a add instruction on the low byte, as well as a add with carry instruction on the high byte.

Source Code (Software)

Sorting Algorithm Testing Code:

Receives data from UART, runs selected sorting algorithm on data, and communicates to host PC when sorting is started/finished.

```
;  sorting algorithm testing source code
;
;  Created: 2/14/2025 12:01:11 AM
;  Author : Joe Maloney
;

            .MACRO    U16_CP                 ;  args - rdH,rdL,rrH,rrL compares rdH:rdL to rrH:rdL
            CP        @1    ,  @3
            CPC       @0    ,  @2
            .ENDMACRO
            .MACRO    U16_ADD                ;  args rdH,rdL,rrH,rrL adds rdH:rdL to rrH:rrL and stores in
rdH:rdL
            ADD       @1    ,  @3
            ADC       @0    ,  @2
            .ENDMACRO
            .MACRO    U16_SUB                ;  args rdH,rdL,rrH,rrL subtracts rrH:rrL from rdH:rdL and
stores in rdH:rdL
            SUB       @1    ,  @3
            SBC       @0    ,  @2
            .ENDMACRO
            .MACRO    U16_PUSH               ;  args - rrH,rrL pushes uint onto stack
            PUSH      @1
            PUSH      @0
            .ENDMACRO
            .MACRO    U16_POP                ;  args - rrH,rrL pops uint from the stack
            POP       @0
            POP       @1
            .ENDMACRO
;  Replace with your application code
            .LISTMAC
            .EQU      CHAR_MAX=0xFF
            .CSEG
            .ORG      0x0
            CLR       r0                     ;  clear all registers prior to application start
            CLR       r1
            CLR       r2
            CLR       r3
            CLR       r4
            CLR       r5
```

```
            CLR      r6
            CLR      r7
            CLR      r8
            CLR      r9
            CLR      r10
            CLR      r11
            CLR      r12
            CLR      r13
            CLR      r14
            CLR      r15
            CLR      r16
            CLR      r17
            CLR      r18
            CLR      r19
            CLR      r20
            CLR      r21
            CLR      r22
            CLR      r23
            CLR      r24
            CLR      r25
            CLR      r26
            CLR      r27
            CLR      r28
            CLR      r29
            CLR      r30
            CLR      r31

USART_Init: LDI      r16   ,  0x0          ;  Set baud rate to UBRR0
            STS      UBRR0H,  r16
            LDI      r16   ,  103          ;  49 for 20K baud, 103 for 9600, 12 for 76800
            STS      UBRR0L,  r16
            LDI      r16   ,  (1<<RXEN0)|(1<<TXEN0) ;  enable reciver/transmitter
            STS      UCSR0B,  r16
            LDI      r16   ,  (0<<USBS0)|(3<<UCSZ00) ;  Set frame format: 8data, 1stop bit
            STS      UCSR0C,  r16

start:      RCALL    next                      ;  reserve first 2 bytes on stack for storing the test count
next:       RCALL    getTestCount              ;  get the number of tests to be performed from the uart
            RCALL    testLoop                  ;  run the specified number of tests, getting a new dataset
from the host PC each time
end:        JMP      end

qSortTest:  RCALL    getData                   ;  load new dataset from host PC
            LDI      XL    ,  0x00
            LDI      XH    ,  0x01
            LD       r2    ,  X+
            LD       r3    ,  X+               ;  set X pointer to array start address, and r3:r2 to array
length for quicksort test
            RCALL    sendACK                   ;  start timer on host PC
            RCALL    quickSort
            RCALL    testComplete              ;  stop timer on host PC
            RET

quickSort:  LDI      r16   ,  0x1
            CLR      r17                       ;  use r17:r16 for a constant uint 0x0001
            U16_CP   r17   ,  r16    ,  r3     ,  r2      ;  base case - break if length is 1 or 0
            BRGE     qSortR
            RCALL    part                      ;  after partitioning, the ending address of the pivot is
stored in the Y pointer
            U16_PUSH YH    ,  YL               ;  store pivot location on stack
            U16_SUB  YH    ,  YL     ,  XH     ,  XL      ;  calculate lower array size in bytes
            LSR      YH                        ;  This number is guaranteed to be even
            ROR      YL                        ;  divide by 2 to get number of elements, ror is lsr w/ carry
bit
```

```
                        U16_SUB   r3    ,  r2     ,  YH     ,  YL      ;  calculate number of elements in upper half,
including pivot
                        U16_SUB   r3    ,  r2     ,  r17    ,  r16     ;  Y=Y-1, get rid of the pivot
                        U16_PUSH  r3    ,  r2            ;  store upper array size on stack
                        MOV       r3    ,  YH
                        MOV       r2    ,  YL            ;  move array length into r3:r2 for next call to quicksort
                        RCALL     quickSort              ;  lower half, X is equivalent for this call, r3:r2 holds new
length
                        U16_POP   r3    ,  r2            ;  move upper array length into r3:r2 for next call to
quicksort
                        U16_POP   XH    ,  XL            ;  restore X pointer to previous pivot
                        U16_ADD   XH    ,  XL     ,  r17    ,  r16
                        U16_ADD   XH    ,  XL     ,  r17    ,  r16     ;  move Y to element after previous pivot
(lower byte of first element in upper half array)
                        RCALL     quickSort              ;  Upper half, X is at the element after previous pivot, r3:r2
holds upper half length

            qSortR:     RET

            ;  Array start address is X (array start and pivot are the same element), array length stored at r3:r2
            part:       MOV       r0    ,  XL
                        MOV       r1    ,  XH            ;  write down pivot address in r1:r0
                        MOV       YL    ,  XL
                        MOV       YH    ,  XH            ;  set y pointer to first (non-pivot) value in array
                        LD        r4    ,  X+
                        LD        r5    ,  X+            ;  load the pivot into r5:r4
                        CLR       r17
                        LDI       r16   ,  0x1           ;  use r17:r16 to increment loop counter
                        CLR       r7
                        MOV       r6    ,  r16           ;  use r7:r6 for loop counter, start at 1
            partL1:     U16_CP    r7    ,  r6     ,  r3     ,  r2      ;  stop loop when counter == r3:r2 (array
length)
                        BREQ      partR
                        LD        r12   ,  X+
                        LD        r13   ,  X+            ;  load the current value into r13:r12
                        U16_CP    r13   ,  r12    ,  r5     ,  r4      ;  compare value to pivot
                        BRLO      partL2
                        JMP       partL3                 ;  don't swap if value>pivot
            partL2:     RCALL     qSwap                  ;  swap the pivot and value if value is less than pivot
            partL3:     U16_ADD   r7    ,  r6     ,  r17    ,  r16     ;  increment loop counter
                        JMP       partL1
            partR:      RCALL     qSwapPivot             ;  swap *Y and pivot
                        RET

            qSwapPivot: LD        r14   ,  Y+
                        LD        r15   ,  Y+            ;  store value to be swapped in r15:r14
                        U16_SUB   YH    ,  YL     ,  r17    ,  r16
                        U16_SUB   YH    ,  YL     ,  r17    ,  r16     ;  send Y pointer back to address of value to
be swapped
                        MOV       XL    ,  r0
                        MOV       XH    ,  r1            ;  restore X to pivot location
                        ST        X+    ,  r14
                        ST        X+    ,  r15           ;  store *Y at original pivot location
                        ST        Y+    ,  r4
                        ST        Y+    ,  r5            ;  store pivot at address of Y
                        U16_SUB   YH    ,  YL     ,  r17    ,  r16
                        U16_SUB   YH    ,  YL     ,  r17    ,  r16     ;  send Y pointer back to address of value to
be swapped
                        U16_SUB   XH    ,  XL     ,  r17    ,  r16
                        U16_SUB   XH    ,  XL     ,  r17    ,  r16     ;  send X pointer back to original array start
addr, used for calculating upper/lower half length in qsort
                        RET

            ;  swaps values at (Y+1) and X, does not change X, Y=(Y+1)
            qSwap:      LD        r15   ,  -X
```

```
                LD        r15   ,  -X               ;  retract X back to the address of the value to be swapped
                U16_ADD   YH    ,  YL     ,  r17    ,  r16
                U16_ADD   YH    ,  YL     ,  r17    ,  r16     ;  increment Y pointer
                LD        r14   ,  Y+
                LD        r15   ,  Y+               ;  load value to be swapped from Y pointer into r15:r14
                U16_SUB   YH    ,  YL     ,  r17    ,  r16
                U16_SUB   YH    ,  YL     ,  r17    ,  r16     ;  send Y pointer back to address of value to
be swapped
                LD        r18   ,  X+
                LD        r19   ,  X+               ;  load other value to be swapped into r19:r18
                U16_SUB   XH    ,  XL     ,  r17    ,  r16
                U16_SUB   XH    ,  XL     ,  r17    ,  r16     ;  decrement X pointer to original location
                ST        X+    ,  r14
                ST        X+    ,  r15              ;  store the *(Y+1) value at the original location of X
                ST        Y+    ,  r18
                ST        Y+    ,  r19              ;  store the *X value at (Y+1)
                U16_SUB   YH    ,  YL     ,  r17    ,  r16
                U16_SUB   YH    ,  YL     ,  r17    ,  r16     ;  Y now addresses (Y+1) from the original Y
                RET

        getTestCount:RCALL     sendACK
                LDI       XL    ,  0xFE
                LDI       XH    ,  0x08             ;  set X to last SRAM location
                RCALL     uint16_Rx                 ;  get and store test count in last SRAM location
                RET

        testLoop:   LDI       XL    ,  0xFE
                LDI       XH    ,  0x08             ;  set X pointer to the number of tests to run
                CLR       r17
                LDI       r16   ,  0x1              ;  use r17:r16 to increment loop counter
                LD        r24   ,  X+
                LD        r25   ,  X                ;  load test count into r25:r24, use for loop stop condition
                CLR       r23
                CLR       r22                       ;  use r23:r22 for loop counter
        testL1:     U16_CP    r23   ,  r22    ,  r25    ,  r24
                BREQ      testR
                RCALL     bSortTest                 ;  change this call from bSortTest/qSortTest
                CLR       r17
                LDI       r16   ,  0x1              ;  use r17:r16 to increment loop counter
                U16_ADD   r23   ,  r22    ,  r17    ,  r16     ;  increment loop counter
                JMP       testL1
        testR:      RET

        ;  uses X and Y for indirection to data, Z for accumulator
        bSortTest:  RCALL     getData
                RCALL     sendACK
                RCALL     bubbleSort
                RCALL     testComplete
                RET

        bubbleSort: LDI       XL    ,  0x0              ;  set X to location of n
                LDI       XH    ,  0x1
                LDI       YL    ,  0x4
                LDI       YH    ,  0x1              ;  set Y to second data location
                CLR       ZL
                CLR       ZH                        ;  set Z to 0
                LD        r0    ,  X+
                LD        r1    ,  X+               ;  store the number of numbers (n) in r1:r0,X now points at
low byte of first uint16
                MOV       r18   ,  r0
                MOV       r19   ,  r1               ;  use r19:r18 for outer loop end condition check
                LD        r2    ,  -X
                LD        r2    ,  -X               ;  decrement X to addr of last data uint low byte
                U16_ADD   XH    ,  XL     ,  r1     ,  r0      ;  add n to X address, doing this twice because
uint16 is 2 bytes large
```

```
                        U16_ADD   XH    ,  XL     ,  r1     ,  r0      ;  add n to X address, this makes X point to
the low byte of 1 of the last uint16
                        MOV       r0    ,  XL
                        MOV       r1    ,  XH            ;  load the end of data address into r1:r0, this is stop
condition for the loops
                        LDI       XL    ,  0x2
                        LDI       XH    ,  0x1           ;  X points to first data uint16 low byte
                        CLR       r2
                        CLR       r3
                        CLR       r4
                        CLR       r5
                        CLR       r17
                        LDI       r16   ,  0x1           ;  use r17:r16 to decrement the loop stop condition
                        U16_SUB   r19   ,  r18    ,  r17    ,  r16     ;  outer loop runs (n-1) times
                        CLR       r6
                        CLR       r7                     ;  use r7:r6 for outer loop iterator
            bubbleL1:   U16_CP    r7    ,  r6     ,  r19    ,  r18     ;  outer loop r7:r6 is iterator, starts at 0,
r19:r18 is stop condition, breaks at i = (n-1)
                        BREQ      bubbleR                ;  stop sorting
            bubbleL2:   U16_CP    r1    ,  r0     ,  XH     ,  XL
                        BREQ      bubbleL2end

                        MOV       ZL    ,  XL
                        MOV       ZH    ,  XH            ;  Z reg used for swap, needs to point to original location of
first uint low byte
                        LD        r2    ,  X+
                        LD        r3    ,  X+            ;  Load first uint16 into r3:r2
                        LD        r4    ,  Y+
                        LD        r5    ,  Y+            ;  Load second uint16 into r5:r4
                        U16_CP    r3    ,  r2     ,  r5     ,  r4      ;  compare the numbers
                        BRSH      callSwap               ;  swap if *X >= *Y, brsh is breq for unsigned numbers
                        JMP       bubbleL2
            callSwap:   RCALL     bubbleSwap             ;  swap the numbers if number at X >= number at Y
                        JMP       bubbleL2
            bubbleL2end:LDI       XL    ,  0x2
                        LDI       XH    ,  0x1           ;  reset the X pointer to first uint low byte
                        LDI       YH    ,  0x1
                        LDI       YL    ,  0x4           ;  reset the Y pointer to first uint low byte
                        U16_ADD   r7    ,  r6     ,  r17    ,  r16     ;  increment loop counter
                        U16_SUB   r1    ,  r0     ,  r17    ,  r16
                        U16_SUB   r1    ,  r0     ,  r17    ,  r16     ;  decrement the inner loop stop condition
address by 2 bytes, skip the last element that was sorted in the next iteration
                        JMP       bubbleL1
            bubbleR:    RET

            ;  working regs r21:r20, swaps uint16, assumes *Z is low byte of first uint,r3:r2 is first uint,r5:r4 is
second uint
            bubbleSwap: MOV       r20   ,  r2
                        MOV       r21   ,  r3            ;  store first uint in r17:r16
                        MOV       r3    ,  r5
                        MOV       r2    ,  r4            ;  write second uint into first uint's registers
                        ST        Z+    ,  r2
                        ST        Z+    ,  r3            ;  write second uint into first's sram location
                        ST        Z+    ,  r20
                        ST        Z+    ,  r21           ;  write first uint into second's sram location
                        RET

            getData:    LDI       r26   ,  0x00          ;  set X to start of sram
                        LDI       r27   ,  0x1
                        RCALL     sendACK
                        RCALL     uint16_Rx              ;  get first uint16 at 0x100, this is the number of numbers
(n) in the dataset
                        LDS       r0    ,  0x100
                        LDS       r1    ,  0x101         ;  load n into r0,r1.
                        CLR       ZL
```

```
            CLR       ZH                    ;  use Z for accumulator, and r1:r0 for compare
getDataL1:  U16_CP    r1    ,  r0     ,  ZH     ,  ZL
            BREQ      getDataR
            RCALL     uint16_Rx             ;  get the next dataset number
            ADIW      ZL    ,  1
            JMP       getDataL1
getDataR:   RET

uint16_Rx:  RCALL     USART_Rx              ;  receives a single byte from
            ST        X+    ,  r17
            RCALL     USART_Rx
            ST        X+    ,  r17
            RET

uint16_Tx:  LD        r16   ,  X+
            RCALL     USART_Tx
            LD        r16   ,  X+
            RCALL     USART_Tx
            RET

testComplete:LDI       r16   ,  0xFF
            RCALL     USART_Tx
            RET

sendACK:    LDI       r16   ,  0xF0
            RCALL     USART_Tx
            RET
;  Wait for empty transmit buffer
USART_Tx:   LDS       r17   ,  UCSR0A       ;  working: r17, sends byte in r16 , read uart status reg
            SBRS      r17   ,  UDRE0        ;  infinite loop until I/O is empty, checks if data empty bit
 is set in uart status reg
            RJMP      USART_Tx
;  Put data (r16) into buffer, sends the data
            STS       UDR0  ,  r16
            RET

USART_Rx:   LDS       r17   ,  UCSR0A       ;  reads uart sreg into r17, blocking the read of the uart
 data register until data ready
            SBRS      r17   ,  RXC0
            RJMP      USART_Rx
            LDS       r17   ,  UDR0         ;  read uart data register into r17
            RET
;  table - a non uart dataset used for debugging w/ the simulator
table:      .DB       0x64  ,  0x0    ,  0xc3   ,  0xca   ,  0x38   ,  0xad   ,  0xbc   ,  0x79   ,  0xfc
 ,  0x8e   ,  0x3a   ,  0xbd   ,  0x53   ,  0x83   ,  0x69   ,  0xcb   ,  0x67   ,  0x63   ,  0x55   ,  0xc4   ,  0x09
 ,  0xc0   ,  0xc5   ,  0x5a   ,  0xd3   ,  0x01   ,  0xc0   ,  0x40   ,  0x36   ,  0x3f   ,  0x9d   ,  0xea   ,  0xf8
 ,  0x9e   ,  0x9c   ,  0xea   ,  0x15   ,  0x51   ,  0x07   ,  0xfe   ,  0x58   ,  0xee   ,  0x66   ,  0xca   ,  0xec
 ,  0x9a   ,  0x12   ,  0x3e   ,  0x0d   ,  0xf6   ,  0xa2   ,  0x7b   ,  0xe6   ,  0x0b   ,  0x93   ,  0x2f   ,  0x78
 ,  0x24   ,  0x4c   ,  0x9a   ,  0xf7   ,  0x81   ,  0x04   ,  0x90   ,  0x71   ,  0x3e   ,  0xf5   ,  0xa8   ,  0xbd
 ,  0xbe   ,  0x09   ,  0x1c   ,  0xfb   ,  0xfd   ,  0xd5   ,  0x4a   ,  0x89   ,  0x24   ,  0xfd   ,  0x27   ,  0x00
 ,  0xa1   ,  0x53   ,  0x34   ,  0xd6   ,  0xec   ,  0xd7   ,  0x60   ,  0xfd   ,  0xc1   ,  0x11   ,  0x5d   ,  0x55
 ,  0x77   ,  0x0c   ,  0x0d   ,  0xbc   ,  0x51   ,  0xbb   ,  0x78   ,  0x01   ,  0x39   ,  0x35   ,  0xe4   ,  0x5a
 ,  0x82   ,  0xae   ,  0xd9   ,  0x92   ,  0x74   ,  0xea   ,  0x5f   ,  0x92   ,  0x2d   ,  0x5a   ,  0x96   ,  0xd1
 ,  0xbb   ,  0xc6   ,  0x4b   ,  0x41   ,  0x2e   ,  0xba   ,  0xb6   ,  0xfc   ,  0x21   ,  0x85   ,  0xf8   ,  0xa1
 ,  0x6a   ,  0xee   ,  0x5f   ,  0x6b   ,  0xdb   ,  0x2a   ,  0x75   ,  0x33   ,  0x71   ,  0x6d   ,  0xe2   ,  0x82
 ,  0xf4   ,  0xee   ,  0x97   ,  0x09   ,  0x51   ,  0xd7   ,  0x57   ,  0x0e   ,  0xfe   ,  0x75   ,  0xd6   ,  0xb6
 ,  0xaf   ,  0xda   ,  0x13   ,  0xba   ,  0x4d   ,  0x00   ,  0x27   ,  0xeb   ,  0xe9   ,  0x7d   ,  0x7b   ,  0x31
 ,  0x5b   ,  0x11   ,  0x3d   ,  0xf2   ,  0x8c   ,  0x2e   ,  0xef   ,  0x37   ,  0x8a   ,  0xc7   ,  0xf7   ,  0x25
 ,  0xf4   ,  0xd3   ,  0xee   ,  0x82   ,  0x64   ,  0x8f   ,  0xb0   ,  0x3d   ,  0xd6   ,  0x85   ,  0x22   ,  0x9e
 ,  0x3e   ,  0x67   ,  0x2b   ,  0x36   ,  0x9a   ,  0xd0   ,  0x88   ,  0x9e   ,  0xbf   ,  0x81   ,  0x78   ,  0x43
 ,  0x3b
getDataDebug:LDI       XL    ,  0x0         ;  same as getData, but reads from program flash instead of receiving data
via uart
           LDI       XH    ,  0x1
           LDI       ZL    ,  low(table*2)
```

```
            LDI       ZH    ,  high(table*2) ;  set Z to starting address of table
            RCALL     getuint16Debug        ;  get the number of uint data elements in table (n)
            LDS       r0    ,  0x100
            LDS       r1    ,  0x101        ;  load n into r0,r1.
            CLR       YL
            CLR       YH                    ;  use Y for accumulator, and r1:r0 for compare
debugL1:    U16_CP    r1    ,  r0     ,  YH     ,  YL
            BREQ      getDataDebugR
            RCALL     getuint16Debug        ;  get the next dataset number
            ADIW      YL    ,  1
            JMP       debugL1
getDataDebugR:RET

getuint16Debug:LPM       r16   ,  Z+        ;  same as getuint16, but loads uint16 from program memory for debugging
            ST        X+    ,  r16          ;  store n low byte
            LPM       r16   ,  Z+
            ST        X+    ,  r16          ;  store n high byte
            RET
```

Division, Single Parent Caller

Performs division on a single 1-byte integer, with subroutines called by caller of division routine.

```
;  Division Source Code
;
;  Created: 2/14/2025 12:01:11 AM
;  Author : Joe Maloney / Eugene Rockey
;
;  Declare Variables
;  ******************************
            .DSEG
            .ORG     0x100                  ;  originate data storage at address 0x100
quotient:   .BYTE    1                      ;  uninitialized quotient variable stored in SRAM aka data segment
remainder:  .BYTE    1                      ;  uninitialized remainder variable stored in SRAM
            .SET     count=0                ;  initialized count variable stored in SRAM
;  ******************************
            .CSEG                           ;  Declare and Initialize Constants (modify them for different results)
            .EQU     dividend=20            ;  8-bit dividend constant (positive integer) stored in FLASH memory aka
code segment
            .EQU     divisor=5              ;  8-bit divisor constant (positive integer) stored in FLASH memory
;  ******************************
;  * Vector Table (partial)
;  ******************************
            .ORG     0x0
reset:      JMP      main                   ;  RESET Vector at address 0x0 in FLASH memory (handled by MAIN)
int0v:      JMP      int0h                  ;  External interrupt vector at address 0x2 in Flash memory (handled by
int0)
;  ******************************
;  * MAIN entry point to program*
;  ******************************
            .ORG     0x100                  ;  originate MAIN at address 0x100 in FLASH memory (step through the code)
main:       CALL     init                   ;  initialize variables subroutine, set break point here, Stack contains
address of below call (0x0102),SP=0x08FD (first empty byte on stack),PC=0x0100 (current  instruction)
            CALL     getnums                ;  PC=0x0102, SP=0x08FF, Stack unmodified from above call.  After this
instruction, SP=(SP-2), Stack = 0x0104, PC=0x0111
            CALL     test                   ;  PC=0x0104, SP=0x08FF, Stack unmodified from above call.  After this
instruction, SP=(SP-2), Stack = 0x0106, PC=0x114
            CALL     divide                 ;  PC=0x0106, SP=0x08FF, Stack unmodified from above call.  After this
instruction, SP=(SP-2), Stack= 0x0108, PC=0x0131
endmain:    JMP      endmain
init:       LDS      r0     ,  count        ;  get initial count, Stack = 0x0102,SP=0x08FD,PC=0x010A
            STS      quotient,  r0          ;  use the same r0 value to clear the quotient-
            STS      remainder,  r0         ;  and the remainder storage locations
            RET                             ;  return from subroutine, Stack and SP unmodified from calling this
function, PC=0x0100, after this instruction SP=0x08FF,PC=0x0102 (popped from stack), Stack remains the same
```

```
getnums:    LDI     r30    ,  dividend     ;  SP=0x08FD,PC=0x0111,Stack = 0x0104
            LDI     r31    ,  divisor
            RET                            ;  Stack and SP unmodified from start of function call. After this
instruction, SP=0x08FF,PC=0x0104(popped from stack)
test:       CPI     r30    ,  0            ;  is dividend == 0 ?
            BRNE    test2
test1:      JMP     test1                  ;  halt program, output = 0 quotient and 0 remainder
test2:      CPI     r31    ,  0            ;  is divisor == 0 ?
            BRNE    test4
            LDI     r30    ,  0xEE         ;  set output to all EE's = Error division by 0
            STS     quotient,  r30
            STS     remainder,  r30
test3:      JMP     test3                  ;  halt program, look at output
test4:      CP      r30    ,  r31          ;  is dividend == divisor ?
            BRNE    test6
            LDI     r30    ,  1            ;  then set output accordingly
            STS     quotient,  r30
test5:      JMP     test5                  ;  halt program, look at output
test6:      BRPL    test8                  ;  is dividend < divisor ?
            SER     r30
            STS     quotient,  r30
            STS     remainder,  r30        ;  set output to all FF's = not solving Fractions in this program
test7:      JMP     test7                  ;  halt program look at output
test8:      RET                            ;  otherwise, return to do positive integer division
divide:     LDS     r0     ,  count        ;  Load count (0x0) into r0
divide1:    INC     r0                     ;  Increment loop counter
            SUB     r30    ,  r31          ;  Subtract divisor from dividend
            BRPL    divide1                ;  Repeat loop if divisor>dividend after subtraction (if it is not, N flag
is set and does not branch)
            DEC     r0                     ;  Decrement loop counter, because loop counter is incremented prior to
checking if subtraction resulted in a positive number
            ADD     r30    ,  r31          ;  Add dividend to what remains of the divisor.  What remains of the
divisor is guaranteed to be negative.  This calculates the remainder
            STS     quotient,  r0          ;  store quotient at pre-defined quotient return address
            STS     remainder,  r30        ;  store quotient at pre-defined remainder return address
divide2:    RET                            ;  end function call
int0h:      JMP     int0h                  ;  interrupt 0 handler goes here
```

Fahrenheit to Celsius Look Up Table

```
;
;  lab2p2.asm
;  CelsiustoFahrenheitLook-UpTable
;  Created:   10:17:31 AM
;  Author:   Eugene Rockey / Joe Maloney
            .DSEG
            .ORG     0x100
output:     .BYTE    1                     ;  Assign SRAM address 0x0100 to label output
            .CSEG
            .ORG     0x0
            JMP      main                  ;  partial vector table at address 0x0
            .ORG     0x100                 ;  MAIN entry point at address 0x200 (step through the code)
main:       LDI      ZL     ,  low(2*table) ;  load the low byte of the 2-byte table address into ZL
            LDI      ZH     ,  high(2*table);  load the high byte of the 2-byte table address into ZL
            LDI      r16    ,  Celsius      ;  load the value to be converted into r16
            ADD      ZL     ,  r16          ;  add the value to be converted to the z pointer low byte
            LDI      r16    ,  0            ;  load zero into r16
            ADC      ZH     ,  r16          ;  add the carry from the first addition to the high byte of the Z
pointer
            LPM                             ;  lpm = lpm r0,Z in reality, what does this mean? - Load the address in
program memory at Z into r0
            STS      output ,  r0           ;  store look-up result to SRAM
```

```
        RET                         ;  consider MAIN as a subroutine to return from - but back to where?? -
returns to line after hypothetical call instruction to main
;  Fahrenheit look-up table
table:      .DB     32     ,  34     ,  36     ,  37     ,  39     ,  41     ,  43     ,  45     ,  46     ,  48
  ,  50     ,  52     ,  54     ,  55     ,  57     ,  59     ,  61     ,  63     ,  64     ,  66
            .EQU    celsius=7               ;  modify Celsius from 0 to 19 degrees for different results
            .EXIT
```

Demonstration Code

Used during demonstration, performs 1-byte integer division using nested calls. Also performs the quicksort algorithm on 20 random 2-byte unsigned integers.

```
            .MACRO   ZEROALL                ;  zeros SRAM and registers so that inspecting them is easy, and for
repeatability
            CLR      r0                     ;  Clear register r0
            CLR      r1                     ;  Clear register r1
            CLR      r2                     ;  Clear register r2
            CLR      r3                     ;  Clear register r3
            CLR      r4                     ;  Clear register r4
            CLR      r5                     ;  Clear register r5
            CLR      r6                     ;  Clear register r6
            CLR      r7                     ;  Clear register r7
            CLR      r8                     ;  Clear register r8
            CLR      r9                     ;  Clear register r9
            CLR      r10                    ;  Clear register r10
            CLR      r11                    ;  Clear register r11
            CLR      r12                    ;  Clear register r12
            CLR      r13                    ;  Clear register r13
            CLR      r14                    ;  Clear register r14
            CLR      r15                    ;  Clear register r15
            CLR      r16                    ;  Clear register r16
            CLR      r17                    ;  Clear register r17
            CLR      r18                    ;  Clear register r18
            CLR      r19                    ;  Clear register r19
            CLR      r20                    ;  Clear register r20
            CLR      r21                    ;  Clear register r21
            CLR      r22                    ;  Clear register r22
            CLR      r23                    ;  Clear register r23
            CLR      r24                    ;  Clear register r24
            CLR      r25                    ;  Clear register r25
            CLR      r26                    ;  Clear register r26
            CLR      r27                    ;  Clear register r27
            CLR      r28                    ;  Clear register r28
            CLR      r29                    ;  Clear register r29
            CLR      r30                    ;  Clear register r30
            CLR      r31                    ;  Clear register r31
            RCALL    zeroSRAM               ;  zero the first 0x500 bytes in sram so the memory view looks nice
            .ENDMACRO                       ;  end of macro definition
            .MACRO   U16_CP                 ;  args - rdH,rdL,rrH,rrL compares rdH:rdL to rrH:rdL
            CP       @1     ,  @3           ;  compare low byte
            CPC      @0     ,  @2           ;  compare high byte w/ carry bit from low byte
            .ENDMACRO                       ;  end the macro definition
            .MACRO   U16_ADD                ;  args rdH,rdL,rrH,rrL adds rdH:rdL to rrH:rrL and stores in rdH:rdL
            ADD      @1     ,  @3           ;  add the low bytes
            ADC      @0     ,  @2           ;  add the high bytes w/ carry bit from low bytes
```

```
            .ENDMACRO                       ;  end the macro definition
            .MACRO   U16_SUB                ;  args rdH,rdL,rrH,rrL subtracts rrH:rrL from rdH:rdL and stores in
rdH:rdL
            SUB      @1     ,  @3           ;  subtract the low bytes
            SBC      @0     ,  @2           ;  subtract the high bytes w/ carry bit from low bytes
            .ENDMACRO                       ;  end the macro definition
            .MACRO   U16_PUSH               ;  args - rrH,rrL pushes uint onto stack
            PUSH     @1                     ;  push low byte onto stack
            PUSH     @0                     ;  push high byte onto stack
            .ENDMACRO                       ;  end the macro definition
            .MACRO   U16_POP                ;  args - rrH,rrL pops uint from the stack
            POP      @0                     ;  pop high byte
            POP      @1                     ;  pop low byte
            .ENDMACRO                       ;  end the macro definition
            .LISTMAC                        ;  expand macros in the listing file
;
;
;
;
;  Division Source Code, Single Call
;
;  Declare Variables
;  ******************************
            .DSEG                           ;  location counter in data segment
            .ORG     0x100                  ;  originate data storage at address 0x100
quotient:   .BYTE    1                      ;  uninitialized quotient variable stored in SRAM aka data segment
remainder:  .BYTE    1                      ;  uninitialized remainder variable stored in SRAM
            .SET     count=0                ;  initialized count variable stored in SRAM
;  ******************************
            .CSEG                           ;  Declare and Initialize Constants (modify them for different results)
            .EQU     dividend=20            ;  8-bit dividend constant (positive integer) stored in FLASH memory aka
code segment
            .EQU     divisor=3              ;  8-bit divisor constant (positive integer) stored in FLASH memory
;  ******************************
;  * Vector Table (partial)
;  ******************************
            .ORG     0x0                    ;  set location counter to 0x0
reset:      JMP      main                   ;  RESET Vector at address 0x0 in FLASH memory (handled by MAIN)
int0v:      JMP      int0h                  ;  External interrupt vector at address 0x2 in Flash memory (handled by
int0)
;  ******************************
;  * MAIN entry point to program*
;  ******************************
            .ORG     0x100                  ;  originate MAIN at address 0x100 in FLASH memory (step through the code)
;  For these stack is shown as array of 2-byte values, with value at first index bottom of stack and value at top as
last index (e.g. [bottom,middle1,middle2...,top]
ZEROALL ; required for running division algorithm on physical hardware, only on simulator is the line 89 LDS
guaranteed to load a zero when clearing vars
main:       CALL     init                   ;  call init routine, SP=0x08FF ,Stack = [],PC=0x0100
endmain:    JMP      sort                   ;  halt program, SP=0x08FF,Stack=[],PC=0x0102
init:       LDS      r0     ,  count        ;  SP=0x08FD, Stack=[0x0102], PC=0x0104
            STS      quotient,  r0          ;  use the same r0 value to clear the quotient-
            STS      remainder,  r0         ;  and the remainder storage locations
            RCALL    getnums                ;  call getnums, SP and stack unchanged from start of init call, PC=0x010A
            RET                             ;  SP=0x08FD, Stack=[0x0102],PC=0x010F
getnums:    LDI      r30    ,  dividend     ;  SP=0x08FB, Stack=[0x0102,0x010B], PC=0x010C
            LDI      r31    ,  divisor      ;  load divisor into r31
            RCALL    test                   ;  call test, SP and Stack unchanged from start of call, PC=0x010E
            RET                             ;  SP=0x08FB, Stack=[0x0102,0x010B], PC=0x010F
test:       CPI      r30    ,  0            ;  is dividend == 0 ?
            BRNE     test2                  ;  run code at test2 if this test passes
test1:      JMP      test1                  ;  halt program, output = 0 quotient and 0 remainder
test2:      CPI      r31    ,  0            ;  is divisor == 0 ?
            BRNE     test4                  ;  run code at test4 if this test passes
            LDI      r30    ,  0xEE         ;  set output to all EE's = Error division by 0
```

```
            STS      quotient,  r30         ;  store 0xEE @ quotient address
            STS      remainder,  r30        ;  store 0xEE @ remainder address
test3:      JMP      test3                  ;  halt program, look at output
test4:      CP       r30    ,  r31          ;  is dividend == divisor ?
            BRNE     test6                  ;  run code at test6 if this test passes
            LDI      r30    ,  1            ;  then set output accordingly
            STS      quotient,  r30         ;  store 0x1 at quotient if divisor==dividend
test5:      JMP      test5                  ;  halt program, look at output
test6:      BRPL     test8                  ;  is dividend < divisor ?
            SER      r30                    ;  set r30 to 0xFF
            STS      quotient,  r30         ;  set quotient to 0xFF
            STS      remainder,  r30        ;  set output to all FF's = not solving Fractions in this program
test7:      JMP      test7                  ;  halt program look at output
test8:      RCALL    divide                 ;  call divide subroutine, SP=0x8F9, Stack=[0x0102,0x010B,0x010F],
PC=0x012C
            RET                             ;  return from test call, SP=0x08F9, Stack=[0x0102,0x010B,0x010F],
PC=0x012D
divide:     LDS      r0     ,  count        ;  Load count (0x0) into r0
divide1:    INC      r0                     ;  Increment loop counter
            SUB      r30    ,  r31          ;  Subtract divisor from dividend
            BRPL     divide1                ;  Repeat loop if divisor>dividend after subtraction (if it is not, N flag
is set and does not branch)
            DEC      r0                     ;  Decrement loop counter, becuase loop counter is incremented prior to
checking if subtraction resulted in a positive number
            ADD      r30    ,  r31          ;  Add dividend to what remains of the divisor.  What remains of the
divisor is gaurenteed to be negative.  This calculates the remainder
            STS      quotient,  r0          ;  store quotient at pre-defined quotient return address
            STS      remainder,  r30        ;  store quotient at pre-defined remainder return address
divide2:    RET                             ;  SP=0x08F7, Stack=[0x0102,0x010B,0x010F,0x012D], PC=0x0139
int0h:      JMP      int0h                  ;  interrupt 0 handler goes here

;
;
;
;
;  Sorting Source code, table
            ZEROALL                         ;  zero everything for repeatability and readability of memory/processor
view
sort:       RCALL    getDataDebug           ;  Load constant dataset from flash into sram
;  data for sorting is stored in sram
;  0x101:0x100 - size of the array (n), e.g. if this equals 2, there are 2 2-byte elements in the array
;  0x102 and upwards - the data in the array.  Stored as uint16, with low byte at low address.
;  example:
;  address :   value
;  0x100   :   0x02
;  0x101   :   0x00 - n = 0x0002, 2 elements in array
;  0x102   :   0x01
;  0x103   :   0x00 - arr[0] = 0x0001
;  0x104   :   0xFF
;  0x105   :   0x01 - arr[1] = 0x01FF
            LDI      XH     ,  0x1          ;  set high byte
            LDI      XL     ,  0x0          ;  set X to sram start
            LD       r2     ,  X+           ;  load n low byte
            LD       r3     ,  X+           ;  load n high byte, set X to low byte of first data element
;  quicksort call argument 1: r3:r2 - uint16, this is the number of 2-byte elements in array
;  quicksort call argument 2: X - points to low byte of first element in array
            RCALL    quickSort              ;  sort dataset in place
end:        JMP      end                    ;  end  of program

quickSort:  LDI      r16    ,  0x1          ;  set low byte
            CLR      r17                    ;  use r17:r16 for a constant uint 0x0001
            U16_CP   r17    ,  r16    ,  r3     ,  r2      ;  base case - break if length is 1 or 0
            BRGE     qSortR                 ;  return if array size is 1 or 0 elements
```

```
            RCALL    part                    ;  after partitioning, the ending address of the pivot is stored in the Y
pointer
            U16_PUSH YH     ,  YL            ;  store pivot location on stack
            U16_SUB  YH     ,  YL     ,  XH     ,  XL      ;  calculate lower array size in bytes
            LSR      YH                      ;  This number is guaranteed to be even
            ROR      YL                      ;  divide by 2 to get number of elements, ror is lsr w/ carry bit
            U16_SUB  r3     ,  r2     ,  YH     ,  YL      ;  calculate number of elements in upper half, including
pivot
            U16_SUB  r3     ,  r2     ,  r17    ,  r16     ;  r3:r2=(r3:r2)-1, get rid of the pivot
            U16_PUSH r3     ,  r2            ;  store upper array size on stack
            MOV      r3     ,  YH            ;  set high byte
            MOV      r2     ,  YL            ;  move array length into r3:r2 for next call to quicksort
            RCALL    quickSort               ;  lower half, X is equivalent for this call, r3:r2 holds new length
            U16_POP  r3     ,  r2            ;  move upper array length into r3:r2 for next call to quicksort
            U16_POP  XH     ,  XL            ;  restore X pointer to previous pivot
            U16_ADD  XH     ,  XL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_ADD  XH     ,  XL     ,  r17    ,  r16     ;  move X to element after previous pivot (lower byte of
first element in upper half array)
            RCALL    quickSort               ;  Upper half, X is at the element after previous pivot, r3:r2 holds upper
half length

qSortR:     RET

;  Array start address is X (array start and pivot are the same element), array length stored at r3:r2
part:       MOV      r0     ,  XL            ;  set low byte
            MOV      r1     ,  XH            ;  write down pivot address in r1:r0
            MOV      YL     ,  XL            ;  set low byte
            MOV      YH     ,  XH            ;  set y pointer to first (non-pivot) value in array
            LD       r4     ,  X+            ;  read low byte
            LD       r5     ,  X+            ;  load the pivot into r5:r4, X now points to second element in array
            CLR      r17                     ;  set high byte to 0x0
            LDI      r16    ,  0x1           ;  use r17:r16 to increment loop counter
            CLR      r7                      ;  set high byte to 0x0
            MOV      r6     ,  r16           ;  use r7:r6 for loop counter, start at 1
partL1:     U16_CP   r7     ,  r6     ,  r3     ,  r2      ;  stop loop when counter == r3:r2 (array length)
            BREQ     partR                   ;  return if loop counter is equal to array length
            LD       r12    ,  X+            ;  read low byte
            LD       r13    ,  X+            ;  load the current value into r13:r12
            U16_CP   r13    ,  r12    ,  r5     ,  r4      ;  compare value to pivot
            BRLO     partL2                  ;  swap values if value<pivot
            JMP      partL3                  ;  don't swap if value>pivot
partL2:     RCALL    qSwap                   ;  swap the pivot and value if value is less than pivot
partL3:     U16_ADD  r7     ,  r6     ,  r17    ,  r16     ;  increment loop counter
            JMP      partL1                  ;  repeat loop after incrementing counter
partR:      RCALL    qSwapPivot              ;  swap *Y and pivot
            RET                              ;  return from partitioning

qSwapPivot: LD       r14    ,  Y+            ;  set low byte
            LD       r15    ,  Y+            ;  store value to be swapped in r15:r14
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  send Y pointer back to address of value to be swapped
            MOV      XL     ,  r0            ;  set low byte
            MOV      XH     ,  r1            ;  restore X to pivot location
            ST       X+     ,  r14           ;  set low byte
            ST       X+     ,  r15           ;  store *Y at original pivot location
            ST       Y+     ,  r4            ;  set low byte
            ST       Y+     ,  r5            ;  store pivot at address of Y
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  send Y pointer back to address of value to be swapped
            U16_SUB  XH     ,  XL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_SUB  XH     ,  XL     ,  r17    ,  r16     ;  send X pointer back to original array start addr, used
for calculating upper/lower half length in qsort
            RET                              ;  return from swapping pivot with location of final value swapped into the
lower half
```

```
;  swaps values at (Y+1) and X, does not change X, Y=(Y+1)
qSwap:      LD       r15    ,  -X           ;  do this twice because uint16 is 2 bytes large
            LD       r15    ,  -X           ;  retract X back to the address of the value to be swapped
            U16_ADD  YH     ,  YL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_ADD  YH     ,  YL     ,  r17    ,  r16     ;  increment Y pointer
            LD       r14    ,  Y+           ;  read low byte
            LD       r15    ,  Y+           ;  load value to be swapped from Y pointer into r15:r14
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  send Y pointer back to address of value to be swapped
            LD       r18    ,  X+           ;  load low byte
            LD       r19    ,  X+           ;  load other value to be swapped into r19:r18
            U16_SUB  XH     ,  XL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_SUB  XH     ,  XL     ,  r17    ,  r16     ;  decrement X pointer to original location
            ST       X+     ,  r14          ;  store low byte
            ST       X+     ,  r15          ;  store the *(Y+1) value at the original location of X
            ST       Y+     ,  r18          ;  store low byte
            ST       Y+     ,  r19          ;  store the *X value at (Y+1)
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  do this twice because uint16 is 2 bytes large
            U16_SUB  YH     ,  YL     ,  r17    ,  r16     ;  Y now addresses (Y+1) from the original Y
            RET                             ;  return from swapping values
;  table - a 20 value table
table:      .DB      0x14   ,  0x0    ,  0x5e   ,  0x3e   ,  0x81   ,  0x9d   ,  0x2f   ,  0xff   ,  0x03   ,  0x6a   ,
0x07   ,  0x30   ,  0xda   ,  0x71   ,  0xd4   ,  0xb0   ,  0xec   ,  0x92   ,  0xad   ,  0xd1   ,  0xe7   ,  0xf8   ,
0x3e   ,  0xf1   ,  0x39   ,  0x64   ,  0x55   ,  0xdd   ,  0x4f   ,  0xe0   ,  0x20   ,  0x06   ,  0x55   ,  0x02   ,
0x4d   ,  0xaf   ,  0x63   ,  0x80   ,  0x6b   ,  0x33   ,  0x22   ,  0xbd
;  table20 - a 20 byte (10 value) table
table20:    .DB      0x0a   ,  0x0    ,  0xa2   ,  0x35   ,  0xfa   ,  0x94   ,  0x5c   ,  0xbe   ,  0x29   ,  0xb0   ,
0x3d   ,  0xe4   ,  0x62   ,  0x32   ,  0x9a   ,  0xb8   ,  0x9a   ,  0xfb   ,  0x87   ,  0x86   ,  0x91   ,  0x96
getDataDebug:LDI      XL     ,  0x0         ;  same as getData, but reads from program flash instead of receiving data
via uart
            LDI      XH     ,  0x1          ;  set X to start of sram
            LDI      ZL     ,  low(table*2) ;  set low byte
            LDI      ZH     ,  high(table*2) ;  set Z to starting address of table
            RCALL    getuint16Debug         ;  get the number of uint data elements in table (n)
            LDS      r0     ,  0x100        ;  load low byte
            LDS      r1     ,  0x101        ;  load n into r0,r1.
            CLR      YL                     ;  clear low byte
            CLR      YH                     ;  use Y for accumulator, and r1:r0 for compare
debugL1:    U16_CP   r1     ,  r0     ,  YH     ,  YL      ;  compare loop counter to array size
            BREQ     getDataDebugR          ;  stop loading when all values are loaded into sram
            RCALL    getuint16Debug         ;  get the next dataset number
            ADIW     YL     ,  1            ;  increment loop counter
            JMP      debugL1                ;  repeat loop
getDataDebugR:RET                           ;  return from loading values into sram

getuint16Debug:LPM      r16    ,  Z+        ;  same as getuint16, but loads uint16 from program memory for debugging
            ST       X+     ,  r16          ;  store n low byte
            LPM      r16    ,  Z+           ;  load high byte
            ST       X+     ,  r16          ;  store n high byte
            RET                             ;  return after getting both bytes of the uint16

zeroSRAM:   LDI      r16    ,  0x0          ;  zero the first 0x500 values in sram so the sorted values are easy to see
in the memory viewer
            LDI      YH     ,  0x5          ;  set high byte
            LDI      YL     ,  0x1          ;  use Y for loop stop condition
            LDI      XL     ,  0x0
            LDI      XH     ,  0x1          ;  set X to start of sram

zeroSRAML1: U16_CP   XH     ,  XL     ,  YH     ,  YL      ;  stop loop at X=Y=0x300
            BREQ     zeroSRAMR              ;  return from zeroing
            ST       X+     ,  r16          ;  zero the current byte
            JMP      zeroSRAML1             ;  repeat loop
zeroSRAMR:  RET                             ;  return from zeroing sram
```

Main, Signed Char

```c
signed char Global_A;    //1-byte global symbol A
signed char Global_B = 1;//1-byte global symbol B
signed char Global_C = 2;//1-byte global symbol C
int main(void)
{
    Global_A = Global_C + Global_B;//A=B+C
}
```

Main Listing, Signed Char

```
000000f2 <main>:
signed char Global_A;
signed char Global_B = 1;
signed char Global_C = 2;
int main(void)
{
 Global_A = Global_C + Global_B;
  f2:   90 91 00 01    lds     r25, 0x0100    ; 0x800100 <__DATA_REGION_ORIGIN__> this is Global_A
  f6:   80 91 01 01    lds     r24, 0x0101    ; 0x800101 <Global_B>
  fa:   89 0f          add     r24, r25    ; add 1-byte values C and B
  fc:   80 93 02 01    sts     0x0102, r24    ; 0x800102 <__data_end>, write back to Global_C
}
 100:   80 e0          ldi     r24, 0x00      ; 0, Zero the working regs
 102:   90 e0          ldi     r25, 0x00      ; 0, Zero the working regs
 104:   08 95          ret ; return from main
```

Main, Unsigned Char

```c
unsigned char Global_A;    //1-byte global symbol A
unsigned char Global_B = 1;//1-byte global symbol B
unsigned char Global_C = 2;//1-byte global symbol C
int main(void)
{
Global_A = Global_C + Global_B;//A=B+C
}
```

Main Listing, Unsigned Char

```
unsigned char Global_A;
unsigned char Global_B = 1;
unsigned char Global_C = 2;
int main(void)
{
 Global_A = Global_C + Global_B;
  f2:    90 91 00 01    lds     r25, 0x0100    ; 0x800100 <__DATA_REGION_ORIGIN__> this is Global_A
  f6:    80 91 01 01    lds     r24, 0x0101    ; 0x800101 <Global_B>
  fa:    89 0f          add     r24, r25    ; add 1-byte values C and B
  fc:    80 93 02 01    sts     0x0102, r24    ; 0x800102 <__data_end>, write back to Global_C
}
 100:    80 e0          ldi     r24, 0x00      ; 0, Zero the working regs
 102:    90 e0          ldi     r25, 0x00      ; 0, Zero the working regs
 104:    08 95          ret ; return from main
```

Main, Signed Integer

```
signed int Global_A;      //2-byte global symbol A
signed int Global_B = 1; //2-byte global symbol B
signed int Global_C = 2; //2-byte global symbol C
signed int main(void)
{
Global_A = Global_C + Global_B;//A=B+C
}
```

Main Listing, Signed Integer

```
000000f2 <main>:
signed int Global_A;
signed int Global_B = 1;
signed int Global_C = 2;
signed int main(void)
{
Global_A = Global_C + Global_B;
  f2:    20 91 00 01    lds     r18, 0x0100    ; 0x800100 <__DATA_REGION_ORIGIN__>, A low byte
  f6:    30 91 01 01    lds     r19, 0x0101    ; 0x800101 <__DATA_REGION_ORIGIN__+0x1>, A high byte
  fa:    80 91 02 01    lds     r24, 0x0102    ; 0x800102 <Global_B>, B low byte
  fe:    90 91 03 01    lds     r25, 0x0103    ; 0x800103 <Global_B+0x1>, B high byte
```

```
 102:   82 0f         add    r24, r18    ; add the low bytes
 104:   93 1f         adc    r25, r19    ; add the high bytes and carry flag
 106:   90 93 05 01   sts    0x0105, r25    ; 0x800105 <__data_end+0x1>, store high byte
 10a:   80 93 04 01   sts    0x0104, r24    ; 0x800104 <__data_end>, store low byte
}
 10e:   80 e0         ldi    r24, 0x00      ; 0, Zero the working regs
 110:   90 e0         ldi    r25, 0x00      ; 0, Zero the working regs
 112:   08 95         ret            ; Return from main
```

Main, Unsigned Integer

```
unsigned int Global_A;      //2-byte global symbol A
unsigned int Global_B = 1;  //2-byte global symbol B
unsigned int Global_C = 2;  //2-byte global symbol C
int main(void)
{
Global_A = Global_C + Global_B;//A=B+C
}
```

Main Listing, Unsigned Integer

```
000000f2 <main>:
unsigned int Global_A;
unsigned int Global_B = 1;
unsigned int Global_C = 2;
int main(void)
{
Global_A = Global_C + Global_B;
  f2:   20 91 00 01   lds    r18, 0x0100    ; 0x800100 <__DATA_REGION_ORIGIN__>, A low byte
  f6:   30 91 01 01   lds    r19, 0x0101    ; 0x800101 <__DATA_REGION_ORIGIN__+0x1>, A high byte
  fa:   80 91 02 01   lds    r24, 0x0102    ; 0x800102 <Global_B>, B low byte
  fe:   90 91 03 01   lds    r25, 0x0103    ; 0x800103 <Global_B+0x1>, B high byte
 102:   82 0f         add    r24, r18    ; add the low bytes
 104:   93 1f         adc    r25, r19    ; add the high bytes and carry flag
 106:   90 93 05 01   sts    0x0105, r25    ; 0x800105 <__data_end+0x1>, store high byte
 10a:   80 93 04 01   sts    0x0104, r24    ; 0x800104 <__data_end>, store low byte
}
 10e:   80 e0         ldi    r24, 0x00      ; 0, Zero the working regs
 110:   90 e0         ldi    r25, 0x00      ; 0, Zero the working regs
```

```
 112:   08 95         ret          ; Return from main
```

Main, Demonstration

```
unsigned char Global_A;    //1-byte global symbol A
unsigned char Global_B = 1;//1-byte global symbol B
unsigned char Global_C = 2;//1-byte global symbol C
void main(void)
{
 Global_A = (Global_C^2) - Global_B; //bitwise xor on C^(char)0x02, then subtract B
}
```

Main Listing, Demonstration

```
000000f2 <main>:
unsigned char Global_A;
unsigned char Global_B = 1;
unsigned char Global_C = 2;
void main(void)
{
 Global_A = (Global_C^2) - Global_B;
  f2:   90 91 00 01   lds    r25, 0x0100   ; 0x800100 <__DATA_REGION_ORIGIN__>, Global_C
  f6:   82 e0         ldi    r24, 0x02     ; 2, load constant 2
  f8:   89 27         eor    r24, r25; xor with C and 2
  fa:   90 91 01 01   lds    r25, 0x0101   ; 0x800101 <Global_B>, load B from SRAM
  fe:   89 1b         sub    r24, r25    ; Subtract B from result
 100:   80 93 02 01   sts    0x0102, r24   ; 0x800102 <__data_end>, store at A
 104:   08 95         ret ; return form main
```

Schematics (Hardware)

None

Analysis

Call Stack

Program organization methodologies were examined to determine the effects of different program structures while executing on the ATmega328PB. Programs with shallower call stacks required less SRAM usage for the stack, however programs utilizing the stack for calls to subroutines were easier to conceptualize during debugging. Programs requiring a single call also reduced the risk of developer error in missing calls to subroutines, errors in the ordering of subroutine calls, or errors that could be the result of the processor changing state between calls of subroutines.

Sorting Speed

Sorting speed was evaluated by implementing the algorithms bubblesort and quicksort on the ATmega328PB. The UART protocol that was used, along with the software stack used for running the tests showed a limitation on the lower bound of testing speed. Tests faster than ~2 milliseconds could not be accurately profiled by the host PC. It was still shown that running quicksort on the ATmega328PB was much faster than bubblesort, for 16-bit unsigned integers on datasets larger than n=50. The implementation of quicksort used is reliant on the dataset being truly random, however a random pivot selection mechanism could be implemented to support using quicksort on non-randomly arranged datasets.

AVR-GCC Data Types

It was found that while signed vs. unsigned datatypes produce equivalent assembly code when performing addition, with the -O0 compiler option. The differences between integer data types of different sizes, 8-bit and 16-bit, was examined and it was found that the larger data types required more assembly instructions to complete the same addition operation.

Conclusion

The ATmega328PB processor was examined by implementing and examining the execution of various algorithms. It was found that utilizing the stack and nested subroutine calls was better for program development, however utilized more stack space than calling subroutines directly by a parent caller. Additionally, sorting algorithms were tested, revealing that the speed-ups and code simplicity gained by utilizing the stack and recursion must be analyzed on a per case basis, as the tradeoff of possible stack overflow may not be large enough risk to warrant using other methods. It was also discovered that the smallest possible data type needed to store and process data should be used when writing C programs for the ATmega328PB, as unnecessarily large data types result in slower, larger programs than using appropriately sized data types.

References

Application Note AVR220 – Atmel Corp. -1997

Appendix A: Test Data

Table includes dataset size (n), and Quicksort/Bubblesort execution times in milliseconds. Each execution time is the average of running the algorithm on 10 random datasets of size n.

n	Quicksort	Bubblesort
10	1.022625	1.068377
20	1.069379	1.116872
30	1.05772	1.022029
40	1.110387	1.152492
50	1.140022	2.177668
60	1.074433	3.158998
70	1.14212	4.418945
80	1.568246	5.98464
90	1.864409	7.455087
100	2.491426	9.484696
110	2.338958	11.43827
120	2.594733	13.53207
130	2.926826	15.64696
140	3.320432	18.04266
150	3.587079	20.33722
160	3.761697	23.47202
170	4.097199	26.37973
180	4.529738	29.4414
190	4.483128	33.18357
200	5.075097	36.86824
210	5.143642	40.33396
220	5.469131	44.30442
230	5.592632	48.62924
240	6.221747	52.62089
250	6.040621	56.94597

260	6.669784	61.53121
270	6.89106	67.05508
280	7.557344	71.46881
290	7.176352	76.90942
300	7.921195	81.67624
310	7.871604	87.70761
320	8.649492	93.47034
330	8.252239	99.29912
340	9.163046	104.8598
350	8.979416	111.6251
360	9.579992	118.8722
370	10.04477	124.8174
380	10.20293	131.0553
390	10.70578	138.9437
400	10.97214	145.4863
410	11.69012	154.3027
420	11.51657	159.6095
430	11.71858	167.5161
440	11.96353	176.8732
450	12.48105	183.7112
460	13.4155	192.8695
470	13.24301	200.3046
480	14.15331	209.6532
490	13.76834	217.3068
500	13.77735	226.4575
510	14.3137	236.4048
520	13.92183	246.0294
530	15.16771	254.2226
540	14.84203	263.4984
550	15.29396	275.2499
560	15.75627	286.2888
570	15.73591	293.4321
580	17.89646	304.8417
590	16.84911	315.5914
600	17.34705	327.0012
610	17.44723	337.9479
620	17.2605	348.8396
630	18.55497	358.7631
640	17.94243	373.6752
650	18.44742	385.1466
660	20.52121	392.9457

670	19.94927	406.353
680	20.55314	420.3301
690	20.93909	432.1551
700	20.41662	444.6342
710	21.01231	457.3825
720	21.24152	469.7065
730	21.59438	480.9461
740	22.12207	497.2909
750	22.69318	511.6277
760	23.34635	522.4308
770	22.81175	540.0388
780	22.75095	550.2101
790	22.64681	565.9402
800	23.49834	580.4157