PCB 111000_UNO Sample project commentaries

****

Proj_1A1: A Basic first sample program LEDs are used to generate a simple display that repeats endlessly.

It introduces The main routine Every program requires a main routine. It is the routine with which program execution starts when power is applied, after programming or after a reset.

Memory All routines require memory in which to save the results of logical or arithmetic operations. These locations are given names which are known as variables because they can hold a variable result.

All memory locations must must be declared. Those declared inside a routine are normally only available for use by the routine. Those declared outside a routine are available for use by all the routines that share the same C file.

Subroutines: A subroutine provides a basic service and is called by the main routine or another subroutine. It usually requires data from the calling routine and can return a single data value to it.

Project Firm WARE

1. "setup_UNO_extra" This is a code segment that sets up the UNO IC. For example: It determines which pins are to be used with the user switches.

2. The subroutine "I2C_Tx_2_integers()": This sends 32 data bits from the UNO to the PCB_A device. This hosts a program known as the mini-OS. It is one of the jobs of the mini-OS to use this data to control the display. Communication between the devices takes place over the I2C bus.

3. The subroutine "Timer_T0_10mS_delay_x_m(6)". This pauses program execution for 60mS.

****

Proj_1C: Target practice: Testing your reaction time. Another repetitive display. LEDS are disabled provided sw2 is pressed at the instant they are illuminated.

It Introduces 1. Interrupts: These can occur at any time, for example when a switch is pressed. Program flow jumps to a special routine known as the Interrupt Service Routine (ISR). When the ISR has been executed program flow control jumps back to the point at which the interrupt occurred.

2. The .h file ("Proj_1C_header_file.h") This is provided to simplify the program file. A lot of information that could be placed in the program file is removed and placed in this file. It gives details of the code segments and the locations of the subroutines. It also tells the compiler about certain WinAVR files that will be required.

3. Project Firm WARE a. config_sw1_and_sw2_for_PCI (Pin Change Interrupt) This is a code segment that enables switches to generate interrupts when they change an input from high to low and visa versa.

b. switch_2_up A code segment that returns a zero when the switch is pressed and a 1 when it is not. An "if(switch_2_up)" statement is present in the PCI ISR to specify the response when sw_2 has just been released. Note there is also "switch_2_down" and similar segments for sw1 and sw3.

c. SW_reset A code segment that enables the UNO Atmega 328 to be reset by the program. It initialises a special timer known as the watch dog timer (WDT) that triggers a reset when it times out.

Want a bit of help with the C and microcontrollers in general: Then google https://epdf.tips/c-programming-for-microcontrollers.html. for a great introductory read.

Operation Press sw2 and the LED illuminated at that time will transfer from the top row to the bottom. Continue until all LEDs have been shot down.

****

Proj_1G: Basic display with intensity control. A third simple pattern generator.

It introduces:

1. The project subroutines: initialise_display() and Inc_Display(). The purpose of these is to clarify the program operation. One initialises the LED display and the other increments it. No exchange of data takes place.

Note that memory locations required by both the main routine and a subroutine are declared outside the main routine.

2. The USART, the receiver/transmitter unit that enables the Atmega 328 to communicate with a PC. The USART ISR, the response to keyboard activity provided the USART has been configured to generate an interrupt.

3. Pointers: A subroutine is usually allocated some memory space that is only visible to it. Sometimes it is convenient for a subroutine to use memory space allocated to the routine that calls it. Where this is the case the '&' symbol is used to signify that the calling routine is providing the memory. The address of the memory location is passed to the subroutine rather than the contents. The '*' symbol signifies that the subroutine is using memory supplied by the calling routine.

4. Subroutine I2C_Tx(). This uses the I2C bus to communicate with the PCB_A Atmega 328. This subroutine is not normally accessed directly by user projects.

Several program segments are used: "waiting_for_I2C_master" and "send_byte_with_Ack(num_bytes)". These crop up quite frequently and have therefore been defined to clarify program operation.

5. Volatile variables

All variables used in the main() routine have permanent storage space in program memory. Variables only used by the subroutines share a block of memory. Every time a subroutine is called it has to initialise its variables. This is not always convenient. For example: two or more subroutines way need to share a variable. Memory space allocated to this variable cannot be re-initialised every time the subroutine is called. Memory space must therefore be permanently allocated to it. The volatile key word is used to ensure this happens.

There is another situation in which the volatile key word is required: A variable may be changed by in input from the hardware. For example a timer interrupt. This cannot be anticipated by the compiler. Even though the variable is defined in the main() routine the volatile key word is still required.

This can be a bit subtle: sometimes it is worth using the volatile keyword to start with and then trying to remove it.

Another tip: Click on Tools/Options/Environment/Fonts and Colors/VA Brace matching and select a realy bright color. This helps to clarify program flow. Make sure that VA Brace Error uses a different color.

Operation Press keys 1, 2, 3, or 4 to adjust the display intensity.

****

Proj_2A: Another reaction time tester

Introduces:

1. Project subroutine PRN_16bit_GEN(0): A 16 bit psuedo random number generator:

This randomly generates 2^16 -1 (= 65535) different numbers. Note there is a similar routine "PRN_8bit_GEN(0)" that can generate 255 different numbers.

The generators require (a seed) a number to start them off and then generate a new number which can become the seed for the following number. In this way a series of numbers is generated. A location in EEPROM is updated with the latest number. This location also provides the original seed when the generator is first called.

The original seed can also be determined in soft ware:

For example PRN_16bit_GEN(0xFFFF) defines a seed of 0xFFFF.

Google "LFSR" (Linear-Feedback Shift Register) for details of random number generation.

2. Using the EEPROM

This is simply an area of memory that is not reset when power is cycled. Here the PRN generator uses it so that there is no need to define the initial seed in soft ware and the sequence rarely if ever starts from the same place more than once.

It can also be used to store text strings and user prompts.

3. Code segments “Rotate_Port_1” and “Rotate_Port_2”. These are specific to this project. They are defined to de-clutter the code and make it more readable.

Operation

LEDs in the lower row are illuminated in order. When it increments the next segment is illuminated. At the same time the upper leds increment by 1,2 or 3 leds in random order. Therefore it is only occasionally that a pair of segments are illuminated. When this happens press sw2 and the LEDs will flash.

****

Proj_2F: Display numbers manually This shows how the numbers we read on the display are no more than simple patterns in the segments that go to make them up.

It introduces

1. The user prompt: a message that is sent to the PC when the program first runs.

2. The project variable "watch_dog_reset". When power is first applied or immediately after programming it is zero. However when the program re-starts following a SW_reset it is set to 1. This program uses the "watch_dog_reset" to generate different user prompts. The full prompt is used when the program first runs following power-up or programming. However following a SW_reset an abbreviated prompt is used.

3. The Watch Dog Timer (WDT): This is primarily a safety device to ensure that program code never comes to an indefinite standstill. Unless it is reset by the code it generates a reset after a preset time interval (30mS in this case). This type of thing never comes to order and therefore sw1 is used to generate an interruption. Note: The "SW_reset" code segment uses the WDT.

4. Project subroutines "isCharavailable()" and "receiveChar()". Obviously a user entering data at the keyboard cannot be constrained by the WDT. "isCharavailable()" waits several mS for a keypress and if one is not forth coming it resets the WDT and waits again. When the user eventually does make a key press "receiveChar()" gets the data from the USART and the program continue on its way.

5. Project subroutine "waitforkeypress()" is simpler to use and good for exercises, but not very good practice because it cannot be used with the WDT.

****

Proj_2G: A slightly better way of entering numbers Each digit is pre-defined in terms of the segments used to construct it. The segments are labelled "a" to "g".

For example: Digit six uses segments f, e, d, c and g Digit seven uses segments a, b and c The full list of definitions is given in the header file. Each definition is know as a string and is terminated in zero. Each string is stored in program memory and the address of the first letter of each string is given the name of the corresponding digit.

More on pointers: Variable "string_ptr" is defined and will be loaded with the addresses of the strings. When the user keys in a digit from zero to 9, "string_ptr" is loaded with the address of the appropriate string. This address is passed to the subroutine "display_num_string ()" which reads the string one letter at a time and illuminates the display accordingly.

Note: The "&" symbol is not used because in "C" the name of a string is automatically a pointer that contains the address of its first character.

This method may seem an improvement on the job of entering segments manually but we still would not want it clogging up any of our programs. For future projects the details are all taken care of by the mini-OS.

****

Proj_3E: Leaving the mini-OS to deal with displaying the numbers

The following project subroutines are introduced:

1. Num_from_KBD() This requires a pointer to a string known in this case as "digits". As the user presses keys the subroutine uses the numerical ones to populate the string.

2. I2C_Tx_8_byte_array() Every time that a new numerical key press is made this subroutine is used to update the display. It sends the "digits" string to the PCB_A device and the mini-OS displays it.

3. I2C_displayToNum() When the user has finished entering the number and pressed the return key this subroutine requests that the mini-OS convert the display to an actual number and return it to the user program.

4. Num_to_PC() This converts a number to a string of characters and sends them to the PC.

A basic discussion of converting numbers to strings and strings to numbers is given in https://epdf.tips/c-programming-for-microcontrollers.html.

A bit of arithmetic is also carried out to confirm that the operations of entering and displaying data have been successful.

****

//Proj_3G: Demonstrates positive and negative binary numbers

A little program designed to illustrate how positive and negative 8 bit binary numbers work.
Note the most right hand bit is known as bit 0. The most left hand bit is known as bit 7.

8 bits can hold 2^8 (=256) different numbers. The unsigned numbers 0 to 255 or the signed numbers -128, -127. . .0, 1. . . 127

Reset the program for a demonstration of unsigned numbers and press the middle switch to increment the numbers. Notice how the binary pattern builds up as the numbers increase:
1 is 1 2 is 10 4 is 100 8 is 1000 16 is 1 0000 32 is 10 0000
64 is 100 0000 96 is 110 0000 112 is 111 0000 128 is 1000 0000 255 is 1111 1111

Release sw2 and press sw3 for a demonstration of signed numbers. Note that -1 is 1111 1111

Press sw1 to decrement the number and notice that -2 is 1111 1110 -4 is 1111 1100
-8 is 1111 1000 -16 is 1111 0000 -32 is 1110 0000 -64 is 1100 0000 -96 is 1010 0000
-112 is 1001 0000 -128 is 1000 0000

Pulse sw1 and notice that 127 equals 0111 1111 follows -128. Hold sw1 down and the numbers steadily decrement to zero.

Therefore as the binary number increases from 0 to 0111 1111 the decimal number increases from 0 to 127. Increasing the binary number by 1 switches the decimal number to -128. Steadily increasing the binary number to 1111 1111 results in a steady increase on the decimal number to -1.

For signed 8 bit numbers bit 7 is known as the sign bit. The signed bit is zero for positive numbers and 1 for negative numbers.

Consider the number -32 it is 1110 0000 which equals 1000 0000 plus 0110 0000 which equals -128 + 96 = -32.

Consider the number -112 it is 1001 0000 which equals 1000 0000 plus 0001 0000 which equals -128 + 16 = -112.

Therefore decode a signed 8 bit number in two stages: treat the first 7 bits as a positive number. If bit 7 is zero this is the answer and if bit 7 is 1 subtract 128.

Obtaining a negative number: Consider 96: it is 0110 0000 in binary. Invert all the bits and add 1 to get 1001 1111 + 1 = 1010 0000 = -96

Display any negative number:

Toggle sw 3 and if the negative sign is ignored notice that the numbers always add up to 256

****

Proj_4B_data_from_IO

Illustrates a process for entering numbers by pressing the user switches.

Subroutine "number_from_IO()" is introduced.

This spends most of the time in a loop waiting for a flag to be set to inform it that data entry is complete. In the absence of the flag being set, switch interrupts enable program flow to temporarily exit this loop.

The first interrupt occurs when switch 2 is pressed. The integers 0 to 9 are displayed in sequence on digit[0] of the display. The user releases the switch when the correct integer is being displayed.

An interrupt from sw1 shifts the display one place to the left so that a second integer can be entered.

When the number has been entered in full an interrupt from sw3 flashes the display and sets the flag to inform subroutine "number_from_IO()" that data entry is complete.

This subroutine then calls project subroutine "I2C_displayToNum()" to covert the display to a binary number which it then returns to the main routine. Finally some arithmetic is performed to check that the process is working correctly.

Operation Make the following switch presses sw2 to populate digits[0] sw1 to shift display left sw3 to enter the number sw2 to multiply the number by 10 sw1 to divide it by 10 press sw3 before sw1 or 2 to repeat.

****

Proj_4D_int_num_ops

A slightly different method of entering numbers from the user switches. This time the mini-OS uses with them to calculate roots and powers.

Subroutine "int_plus_short_from_IO()" is introduced. It is used to enter two data items from the user switches. The first has up to 6 characters and the second only 1.

Program flow sits in an infinite loop awaiting interrupts from sw3 to set two flags "Return_key" and "data_counter".

When sw3 is pressed for the first time digit[0] of the display is populated as for Proj_4B (integers 0 to 9 are displayed in sequence).

When sw3 is released the display automatically scrolls one place left so that digit[0] can be populated using a new integer when sw3 is pressed again. Subroutine "I2C_Tx_2URNs_from_IO()" keeps the display up to date.

When entry of the number is almost complete and digit[0] is being populated for the last time sw1 must be pressed. The "Return_key" flag will now be set to 1 when sw3 is released.

Subroutine "enter_number()" is called to prepare the display for entry of the next number and increment the flag "data_counter".

This time when sw3 is pressed entry of only one integer is allowed. Subroutine "enter_number()" is called again and "data_counter" is incremented to 2.

Data entry is now complete and program flow exits subroutine "int_plus_short_from_IO()".

The main routine now calls "I2C_Tx_Uarithmetic_OP()" and the mini-OS returns the answer.

Where data is to be entered from the switches it is unlikely that any two cases will be the same. Subroutines used in Projects 4B and D are particular to the projects. They are not general purpose and have not been included among the Project resources. It was found easier to write bespoke subroutines than use general purpose ones.

Operation Make the following switch presses

Press sw1 for roots or sw2 for powers Hold down sw3 to populate the first digit of the display Release sw3 when the correct number is being displayed to move on to the next digit When the final digit is being entered press sw1 before releasing sw3 then release sw1. Press sw3 for next number (only one digit is allowed). Release sw3 and root or power will be calculated. Pulse sw3 to repeat calculation. Hold sw1 or sw2 down while pulsing sw3 then press it again to reset the device.

****

Proj_5E: A stop watch application

Introduces the timer application provided by the mini-OS

Operation At power up or post programming every digit of the display has segment "d" illuminated. Pulse sw1 for a 10mS clock or sw2 for a 100mS clock Press sw1 to pause the clock and save the time Press sw2 to read back the save times press sw1 while sw2 is held down to return to the stop watch Press sw3 to reset the stop watchdog Note: There is memory for 10 times to be saved before the oldest is overwritten.

****

Proj_6E_simple logic and bitwise operations

Illustrates the and, or, exclusive or, not and shift operations that can be used to

set, clear, reverse or test individual bits of a memory location

These are very important: The Atmega device does not come with external switches or lights to enable the user to control it. It does however have registers that do enable the user code to interact with it: In particular there are control registers that enable the hardware (the USART for example) to be set-up. Status registers that can be read to determine whether the USART is ready to transmit a character to the PC for example. Address registers than enable the code to reach the required hardware

Operation:

At the user prompt “R…...R…...” pressing

R or r and follow the instructions.