2313Temper8

please read

I'll gladly answer questions regarding software that I've written, but I cannot help with homework, class assignments, projects and unrelated AVR programming questions... especially since I'm a newbie to the AVR. Thanks.

overview

One way to learn a new processor's architecture and assembly language is to port a well understood program from a different architecture.
I decided that a good way of learning the AVR's instruction set and architecture would be to port **LCDTemper8**, my MC68HC11 assembly language program to read multiple DS18S20s. The result is **2313Temper8**.
One unforeseen side benefit of this approach was finding and correcting bugs in the original code, and finding a new way of calculating the DS18S20's extended temperature.

2313Temper8 demonstrates:

- Extracting the 64 bit ROM ID for up to 8 DS1820/DS18S20 devices and calculating the extended temperature for each device
- Simple interrupt driven serial I/O

The program starts by resetting the DS 18S20 bus and reading the 64 bit ROM ID for each device. When a character is received by the UART, it causes an interrupt and is promptly read by the software.

There are only two commands: R (upper or lower case) sends the number of devices and their ROM IDs via the UART. T (upper or lower case) followed by a number from 0 through 7 returns the extended temperature for that device, followed by a CR and LF.

```
r <- character sent to program
4
0800080007CC0010
C000080007728A10
0D00080007455E10
3300080007B60510
0000000000000000
0000000000000000
0000000000000000
0000000000000000

t0
+023.50C

t1
+023.57C

t2
+023.82C

t3
+022.94C
```

hardware

2313Temper8 was tested on an Atmel AT90S2313-10 running at 8MHZ, using a ceramic resonator.
The program's hardware requirements are minimal; the program uses only one bit - Port B bit 2 (PB2) and the UART.
The Dallas 1-Wire bus requires a pull-up to +5V. Typically, a single 4.7K Ohm resistor on the DS18S20 DQ line (PB2 in this case) is all that's required. In my code, I've turned on the pull-up feature of the AVR's ports; this is enough for the 1-Wire bus and eliminates the need for the external 4.7K pull-up resistor. I've tested this method with up to four DS18S20s, and they operate flawlessly. I also tested the same code with an external 4.7K pull-up and it works as well.
Either the internal port bit pull-up or an external 4.7K resistor can be used with this program, but regardless of the method used the DS18S20s won't work properly without a pull-up!

code description

2313Temper8 starts by initializing the AVR stack, 1-Wire data line (PB2) and UART before doing a ROM search on the 1-Wire bus.
Most of the program was translated directly from the 68HC11 code, so I didn't have to write much from scratch. The original 68HC11 ROM search routines took some effort to write; luckily, the Dallas/Maxim 1-Wire software development download page has a Public Domain assembly language example for the Motorola 6805 called ML6805. It includes a ROM ID search routine that I converted to 68HC11 assembly language; it's called **ROM1st** in both versions. I didn't do much to optimize the code for either the 68HC11 or the AVR... I just made it work. Optimization - as my calculus book used to say - *is left as an exercise for the reader*.
This routine extracts one device ID from the bus, and indicates if this is the only device on the bus or if other devices were found. It also returns a failure if no devices are found.
I used the C code in the Dallas/Maxim Application Note 162: INTERFACING THE DS18X20/DS1822 1-WIRE TEMPERATURE SENSOR IN A MICRO-CONTROLLER ENVIRONMENT as a guide to writing the

GetROMS routine.
GetROMS calls the **ROM1st** routine up to 8 times. Each time a ROM ID is found the 8 ID bytes are copied to one of 8 locations allocated in RAM for ROM ID storage. The routine then calls **ShowROM**. This routine steps through the 8 device ID locations in RAM and converts them to ASCII hex before sending them out the serial port. Empty locations (where no devices were found) are shown as 00000000.
The 68HC11 ShowROM routine has a bug (or feature) that causes it to list the ROM IDs in reverse order. Since the program puts the IDs codes onto the ROM bus in the correct order, it's not a big deal. In any case, the AVR version shows the IDs in the order presented in the Dallas data sheet for the DS18S20.

The program's main loop is pretty simple - it just waits until incoming data causes a UART interrupt. I should probably put the AVR in SLEEP mode to conserve power; maybe in the next version.
Any character received at the serial port will case an interrupt. The serial I/O interrupt handler first disables all further interrupts, then parses (in **ParseIn**) the incoming character for commands that it recognizes (only R and Tn, where n is a number from 0 to 7). I used an interrupt routine to provide a quicker response to incoming data. After the commands have been processed, the interrupts are re-enabled and the program picks up where it left off.
When a valid command to read the temperature is received by the UART, the program reads the device temperature, then does an extended temperature calculation before sending the temperature (as ASCII) out the UART.
The DS18S20's TempMSBand TempLSB registers only provide a basic temperature reading, limited to +/- 0.5 degrees C.
However, by using the values found in the Counts_Remaining and Counts_per_C registers, the DS18x20 has a way of extending the resolution to about +/- 0.1 degrees C. The DS18s20 data sheet gives the equation for calculating the extended temperature:

Temperature = Temperature_read - 0.25 + ((count_per_c - Count_remain) / count_per_c)

There are a couple of ways of implementing this equation in assembly language:

1- write a series of math libraries to do floating (or at least fixed) point arithmetic
2- cheat

I chose #2.
I simplified the equation to make it 68HC11 assembly language friendly. When I ported the program to the AVR, I discovered that both of the devices that I was using to develop the code (the AT90S8515 on the STK500 development board and the AT90S2313)lacked integer divide instructions. The equation had to be simplified even further, because I didn't want to use an integer divide routine.
Look at the equation. It has an integer part (Temperature_read, which is the integer part of the temperature read from the DS18x20) and a fractional part (everything to the right of the minus side).
The first step in making this equation integer arithmetic friendly is to rearrange it slightly to eliminate the fractional subtraction.

Temperature = Temperature_read - 0.25 + ((count_per_c - Count_remain) / count_per_c)

becomes
Temperature = (Temperature_read - 1) + 0.75 + ((count_per_c - Count_remain)/count_per_c)
In the DS18S20, count_per_c is always 16, so this allows us to simplify even more:

Temperature = (Temperature_read - 1) + 0.75 + ((16 - Count_remain) / 16)

We now have a simple integer subtraction followed by a factional addition. If you multiply the fractional part by 100, you're left with:

TFrac = 75 + 100*((16 - Count_remain) / 16)

This can be simplified further by multiplying everything out:

TFrac = 75 + (1600 / 16 - 100*count_remain / 16)

which results in:

TFrac = 75 + (100 - 100*count_remain / 16)

which is equivalent to:

TFrac = 175 - (100*count_remain / 16)

so the final equation is:

Temperature = Temp_Read-1 + TFrac / 100

The TFrac result doesn't have to be divided by 100 - just put a decimal point in the output string, followed by TFrac.
There are no divisions required, just a simple 8 bit multiply routive that I borrowed from an Atmel App note. You can see how this is done in the **CvtExt** (Convert Extended) routine. If the fractional result (TFrac) is over 100, we just subtract 100 from the fractional result and add 1 to the integer temperature.

Even though the program displays the complete TFrac result as part of the temperature, it's obvious that the DS18S20 is not accurate to 0.01° C. Since **count_per_c** is fixed at 16, the smallest change in temperature that the DS18S20 will detect is 1/16° C, or 0.0625° C; in reality, it's probably not this accurate. You should round the returned temperature to 0.1° C in the host program on the PC.

comments

Having a 1-Wire ROM search routine is useful if you can only spare one bidirectional pin in your design, or if you need to use more DS18S20s than you have bidirectional pins available. Otherwise, please consider other options that may be easier to implement. One way is to use one bidirectional I/O pin per DS18x20. This would allow you to dispense with the ROM search and ROM matching routines and would match a port bit/DS18x20 with a particular location. With my code, all you get is a list of ROM ID numbers, not the physical location of the device. For instance, assume that device 0 is an outdoor temperature device and Device 1 is measuring the indoor temperature. If you replace the outdoor device with a new DS18x20, it may no longer appear as device 0 in the ROM list if its ROM ID number is higher than Device 1...which may now become Device 0.
You can also skip the extended temperature calculations if you use a DS18B20, which provides up to 12 bits of temperature data.
Why don't I display the temperature in degrees F? I just couldn't be bothered to write the code! Also note the almost total lack of error checking in the routines; for example, the software reads the DS18S20's CRC byte, but it's not used at all. I was going to write a table based CRC routine (there are several examples on the 'net and in the Dallas app notes) but my 68HC11 hardware - with a DS1820 at the end of an 8 meter cable - has worked continuously, without error for over a year. This isn't production code, just "experimenter's" code; you may want to add the CRC error check if accurate, trusted data is critical for your project.
I used the Atmel STK500 development kit on this [project. It was only $79 from Digikey. The STK500 comes with an excellent assembler/debugger IDE, and supports a wide range of AVR devices and programming modes. The code was originally developed on the AT90S8515 that comes with the STK500. Porting the code to the '2313 took less than 15 minutes - including the time required to remove the '8515 and insert the '2313 into the STK500 socket.

Download 2313Temper8

Frank Henriquez frank@ucla.edu Updated: May 10, 2003