Controlling an HC11-based Robot

Final Project Report
December 8, 2000
E155

Roy Pollock and Greg Matthews

Abstract:

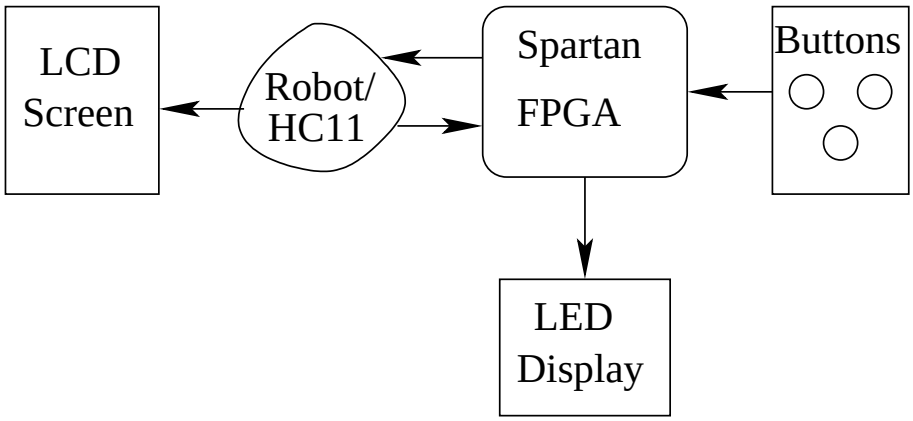
Motorized robot kits for hobbyists with some engineering background can be purchased to investigate the use of motors and sensors. The Rug Warrior Pro is one such robot that uses the HC11 as a microprocessor for motor and sensor control. This project facilitates control of the robot while it is detached from a host computer. A Spartan FPGA supplies glue logic for communicating button presses to the robot's HC11, allowing navigation of a menu displayed on the robot's LCD screen. The FPGA, buttons, and LCD display are mounted on a small breadboard that is attached to the robot, and this extra hardware successfully controls the tasks running on the robot's HC11.

Introduction

The Rug Warrior Pro robot can be programmed to operate its motors and sensors to perform a variety of actions. The user downloads tasks to the robot using a host computer, but once disconnected from the host the set of tasks being run cannot be changed. It is desirable to be able to control the robot both when attached and detached from the host computer. The user should be able to load many different tasks onto the robot, and while in operation choose from among those tasks using additional hardware.

We decided to make use of an LCD display that the robot already knows how to control. On this display we present the user with a menu consisting of a list of the task groups currently loaded on the robot. Each task group will typically represent a program that uses motors and sensors to perform some action, such as seeking light. The user is allowed to scroll through this list and select which task group is running, and then stop those tasks and return to the menu with another button press (which may of course require chasing after the robot).

Figure 1. Block Diagram of Overall Hardware



The breakup and interaction of the hardware components is shown in Figure 1. The FPGA handles the user input portion of the user interface, detecting button presses and communicating these to the robot. LED's that are on the same board as the FPGA are controlled by the FPGA and used to signal when a button press has been detected. The robot's HC11 responds to these button presses by starting and stopping the appropriate tasks and updating the LCD screen.

New Hardware

The Rug Warrior Pro is the only substantial piece of new hardware used in our design. It offers many advantages, including a two-line, fifteen character per line LCD display that is accessible through a *printf* formating routine and automatic A/D conversion routines on two externally accessible lines. It also comes with a fairly comprehensive library of motor control routines including velocity and tracking management.

The Rug Warrior can be programmed in two ways. The first is to use the Interactive C language. This was developed by Newton Labs, Inc. and comes with the Rug Warrior Pro kit. Alternatively, a freeware version is available although it is slightly out of date. However the freeware version does have the advantage of being open source. Interactive C has many of the same basic features as C, though a few important features are missing. Pointers are severely limited–they can not be used to arbitrarily access memory. Also, function pointers are completely non-existent. On an annoying if minor note, the switch/case statement is also not implemented.

The second method is to write assembly language routines. These are written in HC11 assembly for a processor in Special Test Mode. They can be integrated into the Interactive C code that serves as the "operating system" for the robot by using a modified assembler called as11_icb which produces relocatable object code which can then be loaded into the robot. Various mechanisms can be used to declare global variables, make subroutines, and specify code to be executed during the boot sequence of the Rug Warrior. This last is the most important feature, as it can be used to install custom interrupt handlers. These features are fully documented in the Interactive C User's Guide [3].

Most importantly, the Rug Warrior can communicate with peripheral devices through an 8-bit multiplexed memory bus connected to Port C on the HC11 that controls the Rug Warrior. To avoid contention, the connections to this bus must be in a high impedance state when the peripheral does not have control of the bus. Bus signaling for write is accomplished by means of four input select lines. When a device is permitted to assert data to the bus, the corresponding line goes low for 250 ns, during which time the peripheral has control. To meet these constraints, we used a 74HC244 octal buffer chip. On the software side, reading from the bus is accomplished by performing a memory access to \$4xxx,\$5xxx,\$6xxx, and \$7xxx for the four lines (the x's represent don't cares).

To avoid the overhead associated with polling, we used an interrupt based method. The IRQ* signal is also externally accessible. It generates an interrupt vector that is not used by anything on the main Rug Warrior board, and so is well suited for use as a peripheral interrupt mechanism. Furthermore, the input

select signal provides a convenient acknowledgment mechanism. The Rug Warrior services the interrupt by reading from the peripheral bus. The peripheral detects this signal edge and uses it as a reset to the flop which asserts IRQ*, thus clearing the interrupt.

FPGA Design

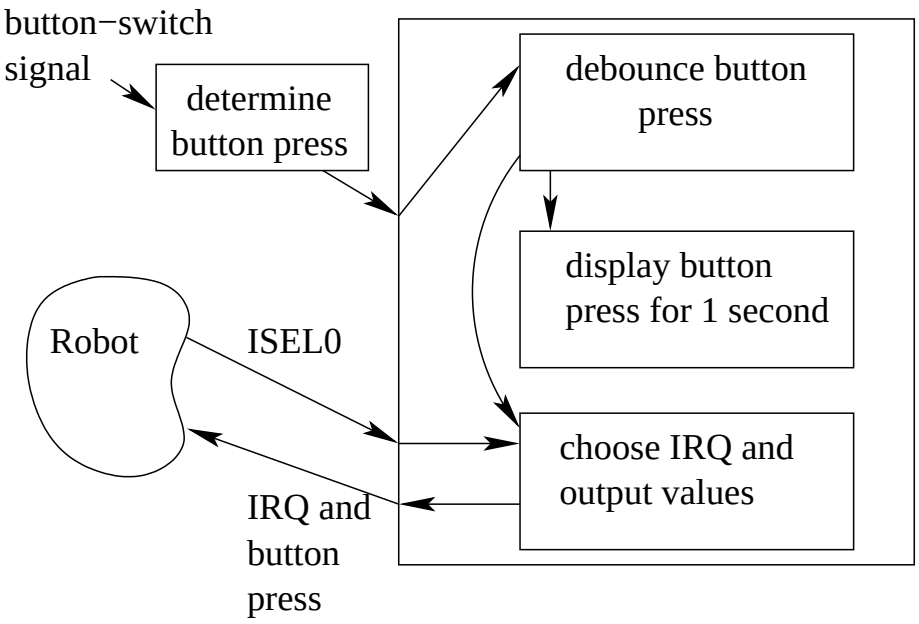
The FPGA is responsible for accepting user input through button-switch presses. This has been implemented with three push-button switches that close an open circuit when pressed. The FPGA detects when a button is pressed, and debounces this signal to avoid multiple detections of the same button press event. The multiple button press events resulting from a non-debounced signal are undesirable because each button press event must result in one input to the menuing system on the robot. Our debouncing logic is based heavily on the lab 4 solution Verilog code, and experimentation revealed that a period of about 2 milliseconds for the debounce sampling was effective. See Figure 2 for Verilog pseudo-code of the debouncing logic.

Figure 2. Debouncing Verilog Pseudo-Code

```
always @ (2 millisecond clock tick) {
   if (no button pressed) {
      continue polling;
   }
   else if (button pressed && no button pressed at last check) {
      //valid button press;
      set IRQ;
      set button press;
   }
}
```

The FPGA must also speak the robot's bus protocol in order to transfer information about the button presses. This involves an active-low IRQ signal, as well as an encoding of which button has been pressed. The IRQ signal from the FPGA must only stay low until the interrupt has been serviced, otherwise the same button press will be acted upon twice by the HC11. To achieve this we wire into the FPGA a signal used by the HC11 to enable input on its port C. This signal, called ISEL0, goes low for one clock tick on the HC11, and is picked up by the FPGA as an indication that the current IRQ has been serviced.

Figure 3. FPGA Block Diagram



LEDs on the FPGA board are utilized to inform the user that a button has been pressed. Each button press is displayed for 1 second, and then the display goes blank. This is useful because insufficient pressure on the buttons may result in a clicking sound but will occasionally not actually close the circuit. In addition the manner in which our hardware is affixed to the robot leaves it vulnerable to damage as the robot moves around, and the LEDs will allow a quick check that no damage was done to the switches or their connections if a collision or other such accident occurs.

Microcontroller Design

The code on the Rug Warrior is conceptually simple. The assembly routine has several functions. The first runs as the robot boots and copies the address of the interrupt handler into the appropriate vector. The interrupt handler itself reads the keypress data from the bus, stores it in a global variable, and increments an interrupt count.

The Interactive C routine causes a function *do_menu()* to start when the robot is turned on. This function polls the global variable set by the assembly routine, waiting for it to change to a non-zero value, indicating a keypress.

When this happens, the robot enters a mode where it kills any running processes and stops the motors. It then displays an entry in its task menu. It also resets the keypress variable and waits for it to change again. On seeing an up keypress it clears the variable and scrolls the menu display up, and does the corresponding scroll if down is pressed. If select is pressed, the *toggle_x()* task for menu choice x is run. This starts all the appropriate subtasks needed to execute the behavior. It also stores their Process ID numbers in a table so that they can be killed later.

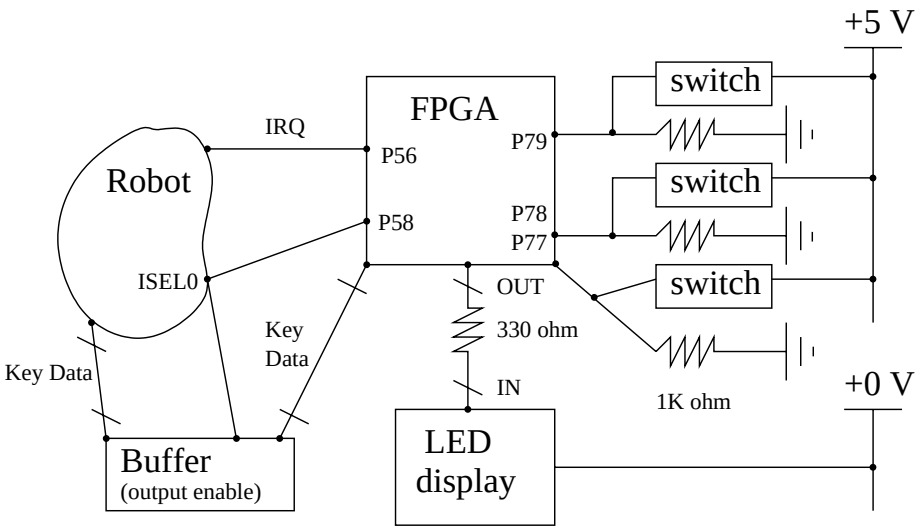
After the tasks have been started, the *do_menu()* task exists the display loop and begins polling for another keypress. While doing this, it uses the minimum possible timeslice by calling the *defer()* builtin. This reduces its impact on the CPU time available to other tasks to an insignificant amount.

Schematics

Our breadboard layout differs from previous labs in that our space constraints have been tightened to avoid having to attach the typical large breadboard to the robot. Power and ground are now supplied by the robot through convenient pinouts on the robot's board, and the robot's batteries provide power for the entire system for many hours.

The button switches supply an active-high signal to the FPGA, and the corresponding pins on the FPGA are tied low to avoid floating inputs. An 8 bit latch acts as a buffer for sending of keypress data to the robot's HC11. The HC11's ISEL0 signal described in FPGA Design controls the output enable of the buffer, placing the signals held by the buffer onto the HC11's port C.

Figure 4. Breadboard Schematic



Results

Our project accomplished its goals. There were no significant discrepancies between the proposal and the finished project. The automatic generator for the menuing system has not yet been generated, but it is extremely easy to make the modifications by hand.

The most difficult part of the design was the interrupt handling. First, we had difficulty getting the assembly code to interact with the Interactive C code correctly. This turned out to be due to a limitation in the as11_icb program which only allows 16-bit integer values to be declared in assembly files. Second, we had to figure out how to get the FPGA to acknowledge that the interrupt had been cleared without using a lot of extra lines. The breakthrough came when we found that the CLB's had set/reset lines for their flops that are independent of the GSR signal.

References

The success of this project is due in large part to the excellent documentation available for the Rug Warrior Pro robot system. Here are the citations.

[1] Jones, Seiger, Flynn, *Mobile Robots, 2ncd ed.*, A K Peters, Natick, Mass., 1999.

[2] Jones, *Rug Warrior Pro Assembly Guide*, A K Peters, Natick, Mass., 1999.

[3] Newton Research Labs, *Interactive C User's Guide*, http://www.newtonlabs.com/ic/manual.html

Parts List

Part	**Source**	**Vendor Part #**	**Price**
8-bit latch	Radio Shack	74HC244	$.75
Rug Warrior Pro Kit	Acroname	R34-RWPRO	$580

Appendix A

Verilog code for accepting and debouncing button presses. Multiple counters are used to control various flops with different periods and maintain synchronous design.

```
/* Greg Matthews
     Verilog to debounce button presses, display which button is pressed for
     one second, and control an IRQ to an HC11
*/

module overall(clk, reset, switches, resetIRQ,
               LEDs, IRQ, swPress);
  input                 clk, reset, resetIRQ;
  input          [2:0]  switches;

  output         [7:0]  LEDs;
  output         [7:0]  swPress;
  output                IRQ;

  reg                   state;
  reg                   IRQ;
  reg            [7:0]  swPress, displayPress;
  reg            [20:0] dispCount;
  reg            [14:0] dbCount;

  // determine which switch is pressed
  parsesw parse(reset, switches, swPress);

  always @(posedge clk or posedge reset or negedge resetIRQ)

    // asynchronous reset for all registers
    if (reset)
      begin
        dispCount <= 21'b0_00000_00000_00000_00000;
        dbCount <= 15'b00000_00000_00000;
        displayPress <= 8'b000_00000;
        state <= 1'b0;
        IRQ <= 1'b1;
      end

    // asynchronous reset for IRQ, this is run every time ISEL0 off the robot
    // goes low
    else if (~resetIRQ) IRQ <= 1'b1;
```

```
    // debouncing and displaying of button press
    else
      begin

        // button press is checked every 2 ms (assuming a 2 MHz clock)
        if (dbCount == 15'b00100_00000_00000)
          begin

            // if no button is pressed -> state goes low
            if (~(|swPress))
              begin
                state <= 1'b0;
              end

            // if a button is pressed and state is low then we have a valid
            // button press
            else if (~state)
              begin
                state <= 1'b1;
                displayPress <= swPress;
                IRQ <= 1'b0;
                dispCount <= 21'b0_00000_00000_00000_00000;
              end
            dbCount <= 15'b00000_00000_00000;
          end
        else dbCount <= dbCount + 1;

        // display button press for approximately 1 second
        if (dispCount == 21'b1_11111_11111_11111_11111)
          begin
            displayPress <= 8'b000_00000;
            dispCount <= 21'b0_00000_00000_00000_00000;
          end

        // only adjust timer if we're actually displaying a button press
        else if (|displayPress) dispCount <= dispCount + 1;
      end

  // determine which LEDs should be lit
  decoder decode(reset, displayPress, LEDs);
endmodule
```

```
/* Greg Matthews
     Determine which switch is pressed
*/

module parsesw(reset, switches, pressed);
  input                 reset;
  input         [2:0]   switches;
  output        [7:0]   pressed;

  reg           [7:0]   pressed;

  parameter             UP =     3'b100;
  parameter             DOWN =   3'b010;
  parameter             SELECT = 3'b001;
  parameter             BLANK =  3'b000;

  // determine which key is pressed
  always @(switches or reset)
    if (reset) pressed <= 8'b000_00000;
    else
      case (switches)
        UP:      pressed <= 8'b000_00010;
        DOWN:    pressed <= 8'b000_00100;
        SELECT:  pressed <= 8'b000_01000;
        default: pressed <= 8'b000_00000;
      endcase
endmodule


/* Greg Matthews
     LED display decoder for display of switch press, w/ async
     reset to blank out the display
*/

module decoder(reset, switch, leds);
  input                 reset;
  input         [7:0]   switch;
  output        [7:0]   leds;

  reg           [7:0]   leds;

  parameter     BLANK   = 8'b0000_0000;
  parameter     UP      = 8'b0000_1100;
  parameter     DOWN    = 8'b0000_0011;
  parameter     SELECT  = 8'b1100_0000;
```

```
  always @(switch or reset)
    if (reset) leds <= BLANK;
    else
      case (switch)
        8'b000_00000: leds <= BLANK;
        8'b000_00010: leds <= UP;
        8'b000_00100: leds <= DOWN;
        8'b000_01000: leds <= SELECT;
        default:      leds <= BLANK;
      endcase
endmodule
```

Appendix B

Here is the Interactive C code for the menuing system

```
/* Menu sample code */
/* Roy Pollock */
/* This code is intended to serve as a basis for what will
   eventually be generated code */

#define UP 2
#define DOWN 4
#define SELECT 8

#define num_items 4  /* constant after link time */

struct menu_entry {

  char message[31]; /* 30 char description of function (looks like
                       two 15 char lines */
  int started;

};

struct menu_entry menu[num_items] =
                          {{"Seek Light",0},
                           {"Seek Dark",0},
                           {"Sonic Commander",0},
                           {"Echo",0}};
/*
struct menu_entry menu[num_items] =
                     {{"Seek Light",0},
                      {"Seek Dark",0},
                      {"Follow wall",0},
                      {"Wimp",0},
                      {"Follow",0},
                      {"Sonic Commander",0},
                      {"Echo",0},
                      {"Bugle",0},
                      {"YoYo",0},
                      {"Theremin",0}};
*/

int pids[10] = {0,0,0,0,0,0,0,0,0,0}; /* each task can have up to
                                         10 sub-tasks */
```

```
void toggle_0() {
  printf("Toggling 0\n");
  if(pids[1]) {
    kill_process(pids[1]);
    kill_process(pids[2]);
    kill_process(pids[3]);
    pids[1] = 0;
  }
  else {
    pids[1] = start_process(moth_point(1));
    pids[2] = start_process(moth_bump());    /* Stop on collision */
    pids[3] = start_process(moth_drive());
  }
}

void toggle_1() {
  printf("Toggling 1\n");
  if(pids[1]) {
    kill_process(pids[1]);
    kill_process(pids[2]);
    kill_process(pids[3]);
    pids[1] = 0;
  }
  else {
    pids[1] = start_process(moth_point(-1));
    pids[2] = start_process(moth_bump());    /* Stop on collision */
    pids[3] = start_process(moth_drive());
  }
}

void toggle_2() {
  printf("Toggling 2\n");
  if(pids[1]) {
    kill_process(pids[1]);
    kill_process(pids[2]);
    kill_process(pids[3]);
    kill_process(pids[4]);
    pids[1] = 0;
  }
  else {
    pids[1] = start_process(snc_sample_sound());
    pids[2] = start_process(snc_capture_command(),1);
    pids[3] = start_process(sonic_control(),1);
    pids[4] = start_process(snc_rpt(),1);
  }
```

```
}

void toggle_3() {
  printf("Toggling 3\n");
  if(pids[1]) {
    kill_process(pids[1]);
    kill_process(pids[2]);
    kill_process(pids[3]);
    kill_process(pids[4]);
    pids[1] = 0;
  }
  else {
    pids[1] = start_process(sample_sound());
    pids[2] = start_process(capture_command(),1);
    pids[3] = start_process(echo_control(),1);
    pids[4] = start_process(rpt(),1);
  }
}

void do_toggle(int i) {
  if(i == 0) toggle_0();
  else if(i == 1) toggle_1();
  else if(i == 2) toggle_2();
  else toggle_3();
}

void do_menu() {
  int selection_num = 0;
  int i, stay_in_menu;
  int egg = 0;
  int eggpid = 0;
  while(1) {
    if(!keypress) {
    defer();
    }
    else {
      keypress = 0; /* so select doesn't "select" the first time */
    /* kill any running process.  Boy I wish I had function pointers*/
      for(i = 0; i < num_items; ++i) {
        if(menu[i].started) {
          do_toggle(i);
          menu[i].started = 0;
        }
      }
      stop();  /* stop the motors */
      stay_in_menu = 1;
```

```
      printf("%s\n",menu[selection_num].message);
      while(stay_in_menu) {
        if(keypress == UP) {
          keypress = 0;
          egg = egg+2;
          selection_num = (selection_num+1)%num_items;
          stay_in_menu = 1;
          printf("%s\n",menu[selection_num].message);
        }
        else if(keypress == DOWN) {
          keypress = 0;
          egg = egg - 1;
          selection_num = (selection_num-1+num_items)%num_items;
          stay_in_menu = 1;
          printf("%s\n",menu[selection_num].message);
        }
        else if(keypress == SELECT) {
          keypress = 0;
          stay_in_menu = 0;
          if (egg == 11) {
            if(eggpid) {
              kill_process(eggpid);
              eggpid = 0;
            }
            else { /* cof() plays the theme from Chariots of Fire */
              eggpid = start_process(cof(),1);
            }
          }
          menu[selection_num].started = 1;
          do_toggle(selection_num);
        }
      }
    }
  }
}

void main() {
  start_process(do_menu(),1);
}
```

Appendix C

Here is the assembly interrupt handling code. The variable_xxx labels make a global variable visible to the C programs. The subroutine_initialize_module function is called as part of the Interactive C initialization sequence.

```
* Roy Pollock
* Interrupt code to poll port C for the keypress data

#define IRQV #$BFF2

    ORG    MAIN_START

* keypress is the global used to store the data
variable_keypress:
    FDB 0

* irqcount stores a count of the interrupts
* (in its MSB, so looks like 256*no. interrupts)
variable_irqcount:
    FDB 0

* routine to install interrupt handler

subroutine_initialize_module:

    LDX IRQV  ; load address of vector
    LDD #interrupt_code_start ; load address of code start
    STD 0,X  ; store in the vector
    RTS


* Here is the code that will be installed

interrupt_code_start:
    LDAB $4000   ; read from expansion bus (should clear interrupt)
    LDAA #$0
    STD variable_keypress
    INC variable_irqcount ; this actually changes the MSB and so does *256
    RTI
```