

***i*Cricket: A web enabled Handy Cricket using the MSP430 microcontroller**

Project Associates

Akshay Saxena
asaxena@cs.uml.edu

Kallol Par
kpar@cs.uml.edu

Project Guide

Dr. Fred Martin
Associate Professor
Department of Computer Science
University of Massachusetts Lowell

Project Site

The Engaging Computing Lab
Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854, USA

May 13, 2003

Abstract

This paper describes the *i*Cricket project which gives a web interface to the Handy Cricket [4] robot controller. The TI-MSP430 microcontroller in conjunction with the CS8900A Ethernet controller; is used to interface with the Handy Cricket using its own serial bus protocol.

1. Introduction

The *i* in *i*Cricket stands for “Internet controlled” or more specifically “IP controlled”. This project deals with making the Handy Cricket, web-enabled, which would make it useful to *remotely* or *IP-ly* control any robotic application created using the Handy Cricket. We have used the Handy Cricket’s bus interface to facilitate this. The web interface comprises of an ultra low-power microcontroller, the TI-MSP430 [2], and an Ethernet controller, the CS8900A [3].

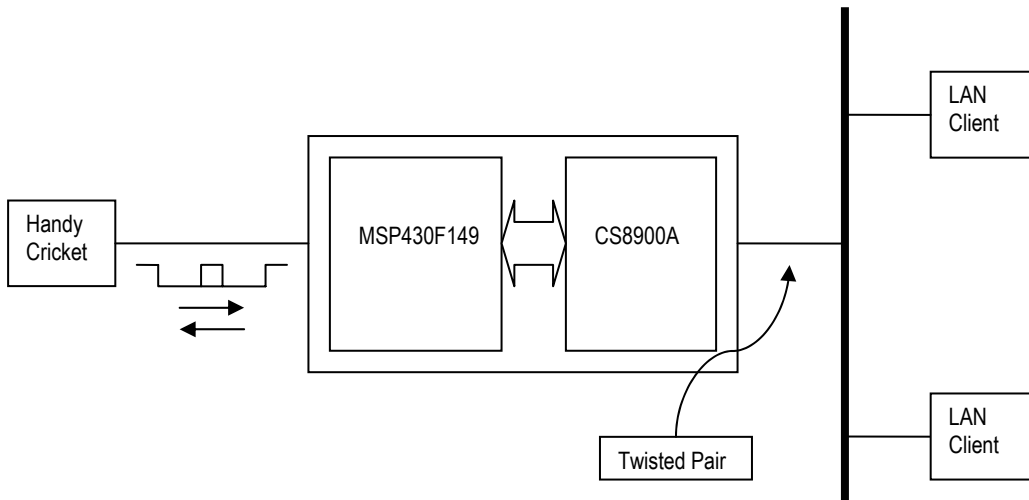


Figure 1.1: The *iCricket* Components

2. The Handy Cricket

The Handy Cricket is a programmable robot controller designed by Prof. Fred Martin.

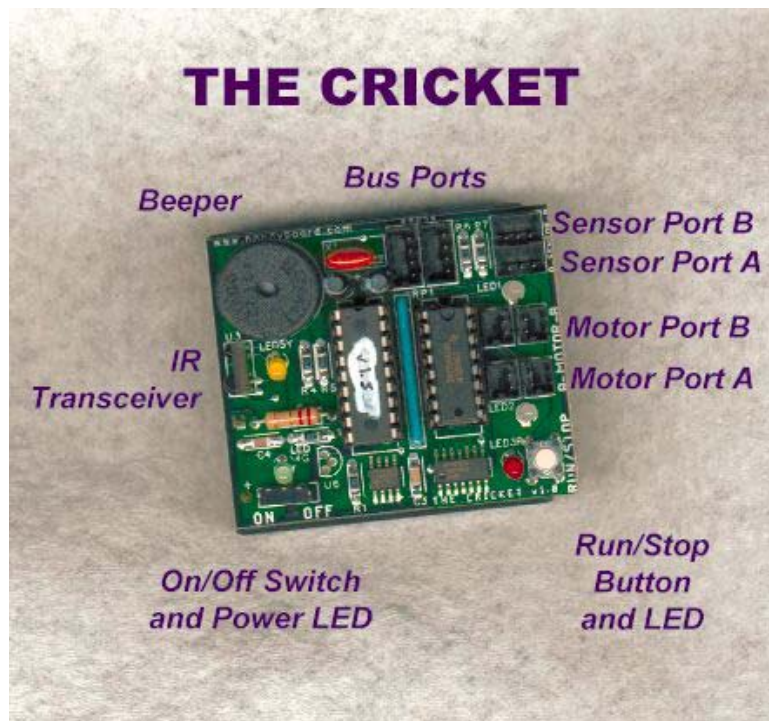


Figure 2.1: The Handy Cricket

It has a Microchip PIC microprocessor with built-in Logo byte-code interpreter. It has 4096 bytes of user program and data memory. It has two DC motors for output. There are two plugs and one bi-color LED on each output. There are inputs for two sensors. The sensor value may be read as true/false or converted to a number from 0 to 255. There are two bus ports, which allow the Handy Cricket to interact with a large collection of other devices. We have used one these bus ports to implement the interaction between the Handy Cricket and its web interface. There is a built-in infrared transceiver with raw data rate of 50k baud. The device operates on a power supply of four AA cells.

3. The TI-MSP430 microcontroller

The MSP430 microcontroller is based on the von-Neumann architecture i.e. all memory and peripherals are in one address space. It has ultra low-power architecture with the following specifications:

- 0.1 microA – 250 microA operating current @ 1MHz.
- 1.8V – 3.6V operation
- 6 micro-sec wakeup from standby mode.

It has seven source-address and four destination-address modes. It also has an integrated 12-bit A/D converter, multiple timers, integrated USARTs and a watchdog timer. The microcontroller also has a built-in programmable oscillator.

The MSP430 has masked ROM, in-system programmable Flash and EPROM. It has 64K addressing space.

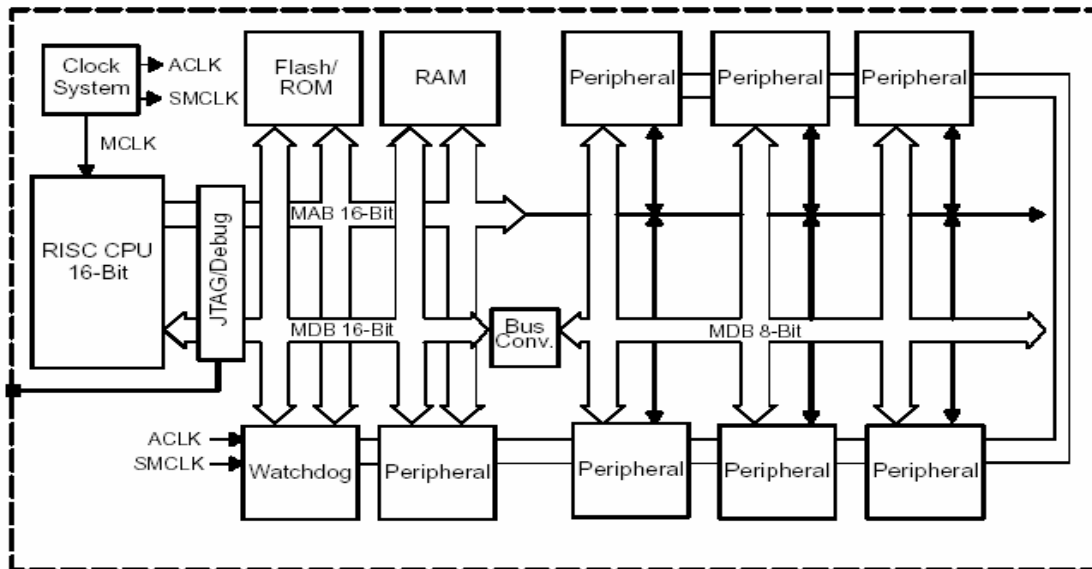


Figure 3.1: MSP430x1xx family architecture

CPU

The CPU has RISC architecture with 27 instructions and 7 addressing modes. It has sixteen 16-bit registers (R0-R3, R4-R15). It has a 16-bit address and data bus.

Memory

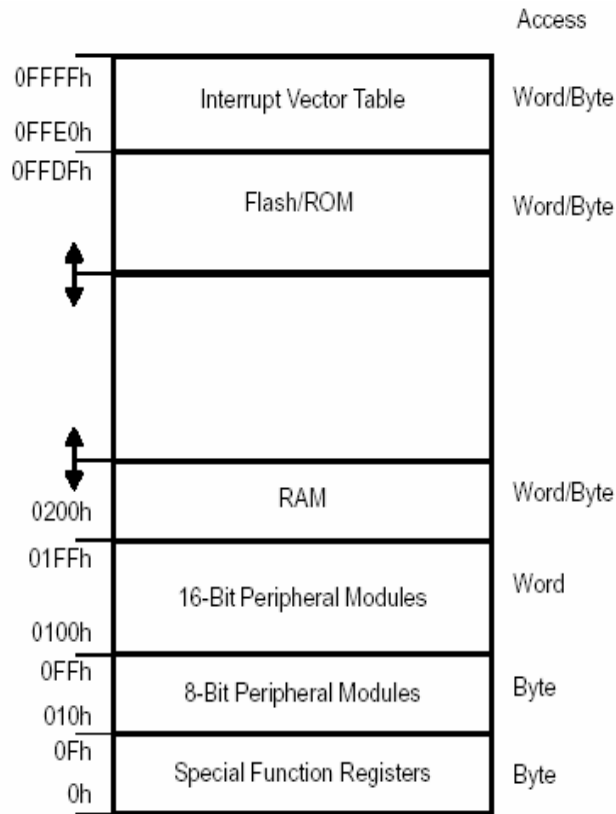


Figure 3.2: MSP430x1xx family memory map

The memory map of the MSP430 has physically separated memory areas for ROM, RAM, Special Function Registers & peripheral modules. These areas are mapped into a common address space.

Digital I/O

MSP430 has up to 6 ports (P1 – P6), each with eight I/O pins. Ports P1 and P2 have interrupt capability. The I/O pins are individually configured for input or output direction and there are independent input and output data registers.

Clock Module

MSP430 includes three clock sources viz. LFXT1CLK for Low / High – frequency oscillator, XT2CLK for High-frequency oscillator and DCOCLK for digitally controlled oscillator with RC-type characteristics.

Flash Memory

The flash memory module of MSP430 is divided into segments (main and information segments). It is bit, byte and word-addressable and programmable and has internal programming voltage generation.

4. The CS8900A ethernet controller [3]

The CS8900A is a low cost Ethernet controller with 4K RAM and on-chip 10Base-T filters. It has an ISA bus interface and operates in either memory or I/O mode (default mode – I/O). The MAC address and the IP addresses is fully configurable by the host (the MSP430). In I/O mode the host accesses the internal registers through the PacketPage architecture. The MSP430 toggles the IOR and IOW pins of CS8900 to communicate with it. For this project we have used a CS8900A development board with an RJ45 ethernet connector.

5. Implementation

5.1 Embedded web server [1]

We interfaced the MSP430 microcontroller with the CS8900A Ethernet controller using Ports 3 and 5 of the MSP430. The MSP430 runs an open source embedded web server that uses a downsized TCP/IP protocol stack. Its functionality is encapsulated by an easy-to-use application programming interface (API). By utilizing these APIs a dynamic HTTP server is implemented. The web server consists of the following major modules:

Ethernet

The main task of the ethernet module is the encapsulation of functions for data transmission by easy-to-use C functions. The ethernet module also generates the clock scheme used for accessing the internal registers of the CS8900A. This module allows for user-configurable MAC addresses for the network interface.

TCP/IP

This module implements the protocols for transferring data over a TCP/IP connection. It incorporates a set of event-handling procedures. The functions of the ethernet module are used to send and receive data. They provide simple, easy-to-use API to the upper application layer. Essential parts of the standards RFC 791, 792 and 793 are implemented in this module.

HTTP Server

The TCP/IP stack is used to implement an HTTP server. This server provides a web page that is stored in MCU flash memory. The module waits for an incoming connection, transfers the web page, closes the connection and waits for another client to connect. The web page consists of buttons that can be used to manipulate the controls on the Handy Cricket.

5.2 MSP430 – Handy Cricket Interface

The Handy Cricket communicates via its serial bus using a proprietary serial protocol. Therefore unlike hardware-based USART processing, any kind of interpretation of the signal data has to be done through software. Accordingly, we have designed the driver routines to understand and generate these signals. Our approach in writing the driver code was to make them interrupt driven for data reception. The MSP430 has six 8-bit bi-directional I/O ports for building various embedded control applications. Ports 1 & 2 have separate interrupt vectors associated with them. It is easy to configure any of the port pins to be negative-edge triggered and interrupt-driven. When a byte is sent from the Handy Cricket, the interrupt vector associated with that pin is used to allow for interrupt-driven data reception.

There exists a *bus send* (“bsend”) primitive available for the Logo language on the Handy Cricket, but there is no implementation of a direct bus receive primitive. The only way to receive data on the Handy Cricket is to ensure that the Handy Cricket application uses the *bus send/receive* or “bsr” primitive. This primitive works by first sending a byte of data on the serial bus and waiting for $\frac{1}{4}$ of a second for reception of data.

It is clear from the above discussion that the Handy Cricket always operates in the master mode, initiating both transmission and reception of data.

Data reception from the Handy Cricket is done using an interrupt service routine (ISR), invoked in the presence of a negative edge on the chosen I/O pin. The ISR performs bit sampling of that pin at regular intervals to acquire information regarding the start bit, eight data bits and the control bit.

Data transmission is also done by the ISR discussed above. This is due to the absence of a bus read mechanism on the Handy Cricket’s Logo language. For transmission of data to the Handy Cricket, the interrupt-driven receive part of the driver code ensures that the data is of a known value, agreed upon by the Logo application and the MSP430 driver code. If there is such a data value received, the ISR interprets that as an indication that the Handy Cricket is waiting to receive data. Correct data transmission is done on the same serial wire, by first generating the appropriate pre-start pulse, the start bit, eight data bits and the command bit.

```

interrupt (PORT1_VECTOR) CricketRead(void)
{
    unsigned short preStartCount = 1000;

    while(preStartCount > 0)
    {
        if(P1IN & 0x80)
            asm("JMP WaitForFirstBit\n\t");
        preStartCount--;
    }

    asm("JMP NoStartBit \n\t"
        "WaitForFirstBit: \n\t"
        "MOV #35, R15 \n\t" // 2 cycles
        "WaitFor15: \n\t"
        "DEC R15 \n\t" // 5 cycles for INV +
        // 2 cycles for ADD + 2 cycles
        "JNZ WaitFor15 \n\t" // 2 cycles
        "MOV.B #7, R14 \n\t");

    cData = P1IN;
    asm("SampleBit: \n\t");
    cData >>= 0x01;

    asm("MOV #20, R15 \n\t"
        "WaitFor10: \n\t"
        "DEC R15 \n\t"
        "JNZ WaitFor10 \n\t");

    cData |= (P1IN & 0x80);
    asm("DEC R14 \n\t"
        "JNZ SampleBit \n\t");

    // sample the pin again for control bit value
    asm("MOV #20, R15 \n\t"
        "WaitForControlBit: \n\t"
        "DEC R15 \n\t"
        "JNZ WaitForControlBit \n\t");

    controlBit = (P1IN & 0x80);
    if (cData == 0x01)
    {
        if (motorAControl == 0x01) // Motor A - ON
        {
            CricketWrite(0x55);
            motorAState = 0x01; // Status value - for updating page
        }
        else if (motorAControl == 0x02) // Motor A - OFF
        {
            CricketWrite(0xAA);
            motorAState = 0x02; // Status value - for updating page
        }
    }

    asm("NoStartBit: \n\t");
    P1IFG &= ~(0x80); // Clear the interrupt pending bit
}

```

Figure 5.1: Code fragment for ISR - CricketRead

```

void CricketWrite(unsigned char data)
{
    unsigned char bitMask;
    unsigned char dataBitCount;
    unsigned char bitTimeCount;

    _DINT();    //disable interrupts

    bitMask = 0x01;
    dataBitCount = 0;
    bitTimeCount = 0;

    //prepare pin 7 for write
    P1DIR |= 0x80; // Initialize Port 1.7 direction to output

    P1OUT &= ~(0x80); //output a '0' on Port 1.7

    //wait for 100 usec
    for(bitTimeCount = 0; bitTimeCount < 158; bitTimeCount++);

    //change the pin back to input mode
    P1DIR &= ~(0x80);

    //generate start bit - 10 usec
    for(bitTimeCount = 0; bitTimeCount < 13; bitTimeCount ++);

    //generate 8 data pulses - 10 usec each
    while(dataBitCount < 8)
    {
        if(!((data & bitMask)==bitMask)) // check if that bit position is '0'
        {
            P1DIR |= (0x80);
            P1OUT &= ~(0x80);
            for(bitTimeCount = 0; bitTimeCount < 9; bitTimeCount++);
        }
        //the bit position is '1'
        else
        {
            P1DIR &= ~(0x80); //return back to input mode
            //delay for exactly one bit time
            for(bitTimeCount = 0; bitTimeCount < 13; bitTimeCount++);
        }

        bitMask <<= 0x01;
        dataBitCount++;
    }

    P1IFG &= ~(0x80); //clear the interrupt pending flag
    P1DIR &= ~(0x80); // set P1.7 to input mode

    _EINT(); //enable interrupts
}

```

Figure 5.2: Code fragment for CricketWrite

```
to ReceiveData
loop
[
    ifelse (bsr 1) = $55
    [a, on]
    [a, off]
]
end
```

Figure 5.3: The Handy Cricket Logo Code

5.3 The Handy Cricket - MSP430 - Web server Interface

The web server communicates with the Handy Cricket driver using global variables. Initially, an HTML page is served when the user opens the *iCricket* URL (<http://icricket.cs.uml.edu>). When the user clicks on a control button, such as the “Motor control”, an event is triggered and sent across to the web server as a form submission. The form submission is done through the *GET* method. The HTTP server receive routine is modified to parse the URL value for the appropriate name-value pair. Based on the name-value pair information appropriate global variables are set/reset, capturing the event generated by the remote user. In its simplest form the Handy Cricket application constantly executes “bsr”, to receive control data from the MSP430. When any remote user-event information is properly captured and available, the receive ISR sends the corresponding control data byte across. On receiving that byte the Handy Cricket code performs the operation based on that data.

6. Future Work

Our future work would involve implementing a broader range of controls for the *iCricket* device for some basic robot applications. We would also be working on the next generation of Handy Cricket device which would have a 16-bit microcontroller (like MSP430) as a replacement for the PIC microcontroller. This version of the Handy Cricket would also have an Ethernet controller integrated with it. It would require us to port a virtual machine on the newer 16-bit processor and implement a network stack with broader range of protocols suiting the small system.

7. Summary

We started by introducing the *iCricket* project. We gave an overview of the Handy Cricket device. In the next sections, we described in some detail the architecture and working of the MSP430 microcontroller and the CS8900A Ethernet controller. Section 5

describes the implementation of this project including key code fragments. We end by giving some directions of our future work.

The complete suite of the programs discussed in this paper can be found online at:

<http://www.cs.uml.edu/~kpar/Robotics/iCricket/>

<http://www.cs.uml.edu/~asaxena/Robotics/iCricket/>

Bibliography

1. MSP430 Internet Connectivity, Andreas Dannenberg, Texas Instruments
2. MSP430x1xx Family – User’s Guide, Texas Instruments
3. CS8900A – Product Data Sheet, Cirrus Logic
4. Inside the Handy Cricket, Dr. Fred Martin
5. Handy Cricket Programming Reference