



Microchip 16-bit Embedded Control Design Contest 2007  
Project Number: **MT2210**  
Microchip Components: **dsPIC30F3010, PIC24FJ64GA002**  
Bonus Components: **MCP6022, ENC28J60, 24LC512**  
**Full Documentation**



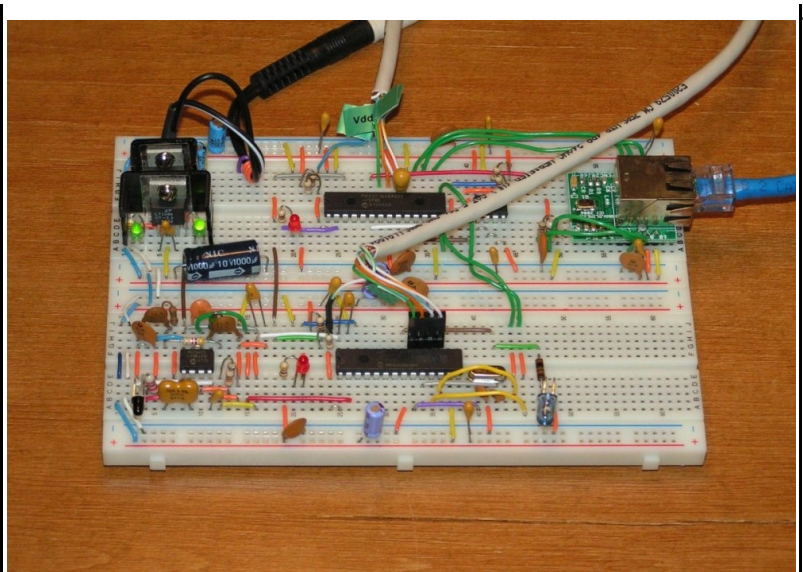
**UIRC:**  
*The Ultimate IR Controller*

## **FULL DOCUMENTATION**

### ***Introduction***

UIRC is a Ethernet Based Universal Remote IR controller designed to provide remote control to any IR device through the Internet, designed mainly to control home remote video equipment and appliances. It is able to recognize any IR signal in a infrared pass-band of 32 KHz up to 70 KHz in several protocols and report the signal via a TCP connection. It also is able to generate any IR command through the same interface. With hooks for IR learning and macros, the device would be able to recognize any IR command and generate macros from IR or TCP commands, converting any remote control in an advanced universal remote system.

This project uses a different approach to process the IR signal: instead of using common IR receiving modules I decided to use a photo-transistor and process the signal digitally using DSP techniques. This way I built a wide band sensor which can detect the frequency of the signal as part of the decoding, which is necessary for a fully learning IR remote system. Using DSP and the 10 bit ADC also improves considerably the detection sensitivity of the final device.



*Illustration 1: The UIRC Prototype, starting at top right clockwise: the voltage regulation chips, the only non-Microchip chips used; the PIC24FJ64GA002 Master Controller; Microchip 24LC512 EEPROM, a prototype board with the ENC28J60 and the Ethernet Magnetics; dsPIC30F3010 UIRC Sensor, MCP6022 IR amplifier and low-pass anti-alias filter.*

UIRC was implemented only with Microchip chips, except for power supply regulation. It has two major functional blocks:

- **The UIRC sensor**, or UIRCS, is an IR decoder/encoder Intelligent Sensor, which based on DSP technology implements a wide-band IR sensor (from a common photo-transistor instead of a IR sensor), with embedded IR command decoding and generation in nine formats (SIRC, JVC, NEC, Sharp, RCA, NRC17, RC5 and generic Pulse Width and generic Pulse Distance modulation). The UIRCS was implemented with a **dsPIC30F3010** which has enough RAM and processing abilities to be able to handle the digital capture, filtering, decoding and encoding in real time. The UIRCS is a standalone project by itself and interfaces with higher end system through a standard UART serial port. It will report protocol, frequency, number of bits, bit encoding, gap time and several other timing parameters depending on the mode (learning or receiving). Through the same interface it can be commanded to generate any IR command at any frequency in the allowed range.
- The UIRC Controller, which is the system that integrates the Intelligent Sensor with the Ethernet controller to provide basic remote IR receiving and sending from a TCP connection. This controller has also a web page interface to monitor the commands and a library needed to implement learning and macros inside the chip. This last feature however couldn't be finished for the contest but it will be very easy to include later. The UIRC controller sends human readable decoding of the received IR commands through a single TCP connection and it also permits the generation of IR command with an ASCII command interface.

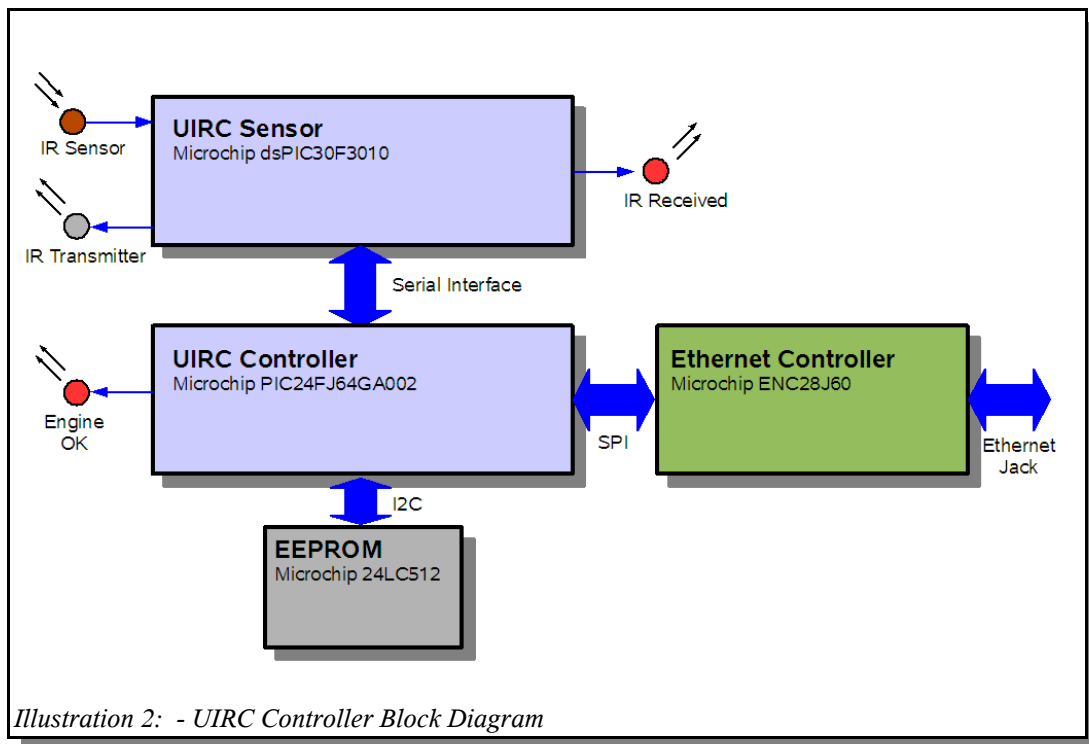
For this contest the submitted entry has the following fully functional features:

1. UIRC Intelligent Sensor is fully implemented, decoding and sending IR commands from a distance of at least 10 feet in nine protocols.
2. UIRC Controller with Ethernet interface is able to receive IR commands from the UIRCS and transmit them to a client connected via TCP at port 10000.
3. A web page interface for IR monitoring. The pages are stored in an external Microchip EEPROM. The pages can be updated through the FTP interface provided by the Microchip TCP/IP stack.
4. Library to save/recall IR events and commands from the EEPROM. Not used at this moment but implemented.

This development used the Microchip MPLAB IDE, the C30 compiler, the Microchip dsPICfd Lite, dsPIC Filter Works, the DSP library, the 24F and dsPIC 30F peripheral libraries and the TCP/IP Stack version 4.12. All the system was developed in C. A ICD2 programmer/debugger was a key tool to finish this on time.

## Hardware Design

The UIRC was designed in two components: the intelligent sensor, or UIRC Sensor (UIRCS) and the UIRC master controller. The sensor is responsible for the analog signal capturing, digital filtering, digital rectification, IR decoding and transmission to the master, via a serial port interface. It also receives commands through the interface to change the sensor mode (sensing or learning) and to generate a IR signal. The second component is the UIRC Master Controller, which is no more than an interface CPU, which in the case of this project will simply interface the UIRCS to a TCP connection. A block diagram explaining the connections is below as Illustration 2.

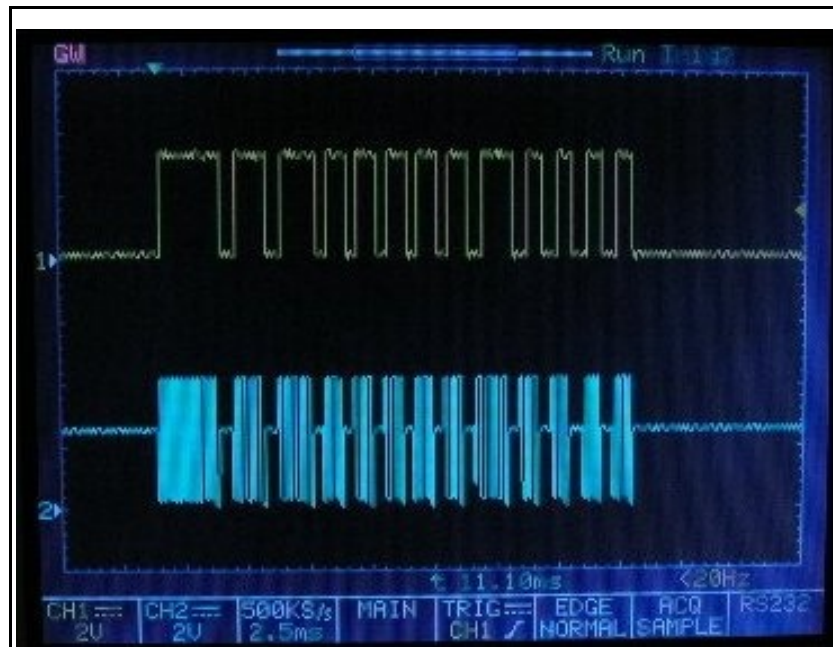


The UIRC Sensor consist of an analog part and a digital part (please reference the schematic at page 14). The analog part is responsible of the IR signal receiving and to prepare the signal for the dsPIC ADC. The IR detector is a LTR-4206E which includes an opaque filter to filter visible light. It is amplified with an operational amplifier (U1A) and then passed through a third order low pass anti-alias filter (U1B). The bonus chip, a Microchip MCP6022, a rail-to-rail operational amplifier was designed for this kind of job and was used in this part of the UIRC. The anti-alias filter was designed using the Online Simulation Tool at Microchip.com. After these analog operations, a 2.5 volt centered signal with frequency components less than 100 KHz, is feed to the AN0 pin of the dsPIC30F3010 (IC1). The dsPIC is where the digital part of the sensor is implemented. The chip will process digitally the signal, filter it to interest bandwidth, rectify it to then process the data digitally and decode one of the nine implemented IR protocols: SIRC, JVC, NEC, Sharp, RCA, NRC17, RC5 and generic Pulse Width and generic Pulse Distance modulation.

The dsPIC30F3010 was selected by its RAM amount and fast 10-bit ADC. The chip is configured to run with a 7.3728 MHz clock externally but internally it is bumped up to 29.5 MHz through the PLL. This speed was fundamental for the DSP algorithms at relatively high frequencies of the IR signals (32 KHz to 70 KHz). It captures the IR analog signal at about 246 KHz (more than twice the allowed frequency of the anti-alias filter), runs a 4<sup>th</sup> order Elliptic IIR 32-70 KHz band-pass filter and then process the raw signal to rectify it, to convert the stream of digital samples into a internal digital representation of the signal. Illustration 3 at page 5 is a capture of the resulting internal signal which is output to the debug pin after this process. The signal clearly shows the IR command envelope.

The dsPIC connects to the master controller (U2) through a UART interface. The dsPIC also has provision to output a IR signal at pin PWM3H. As there are enough PWM resources, future versions will permit the IR generation in several zones at the other PWM outputs.

The UIRC Master Controller is based on a PIC24FJ64GA002, which has enough power, memory and flash to implement the TCP/IP stack and serial communications with the peripherals. Although it passed through my mind to comprise both functions, the sensor and the TCP/IP stack in a single, higher-end dsPIC, I believe that this approach would have been much more difficult if not impossible as the DSP algorithms consume almost all CPU power of the dsPIC. Besides that the dsPIC is a fully standalone smart sensor chip which can be used in other applications.



*Illustration 3: - The bottom trace is the capture at the AN0 pin of the dsPIC and the upper trace is the resulting internal signal after the digital filter and rectification. This signal is output in the debug pin of the dsPIC.*

The Master Controller has the job of receiving the IR events from the UIRC sensor and forwarding them to a TCP connection. It also has the ability to receive commands through that connection to change the sensor mode and also to send IR commands. A web page for IR monitoring was implemented. Additional programming is needed to implement the IR saving and restore and event

handling in the chip which was not ready for the contests. Even that, the system is very useful to implement remote IR communications from a cellular phone with TCP capabilities.

I used the Microchip TCP/IP Stack version 4.12, with the following features enabled: ICMP, classic HTTP, MPFS, DHCP client, FTP server for web page updates and NetBIOS name server. The PIC24FJ uses a Microchip 24LC512 for the web pages storage. A space is reserved at the beginning of the EEPROM (and the routines were written but not used) to implement a list based storage for IR events. A small AJAX page will display in a browser the current received IR events as soon as they happen. This was implemented for debugging and as a future interface for the fully implemented IR learning/macro capability planned for the system.

The Ethernet controller is a Microchip ENC28J60 (U3). This chip integrates very easily with the TCP/IP Stack and no much more knowledge is needed for the developer. To ease my development, I used a Olimex ENC28J60 board which includes the Microchip ENC28J60, supporting hardware, and the RJ-45 jack with the magnetics integrated (JP0026).

The rest of the hardware is the power supply section. Two voltage regulators (U4 and U5) supply the 5V and 3.3V needed for the chips.

## Software Design

The UIRC sensor software is the most complicated part of the project. It not only involves DSP algorithms but the code optimization is highly critical to achieve its goal in real time. The following block diagram shows the internal components developed in the dsPIC:

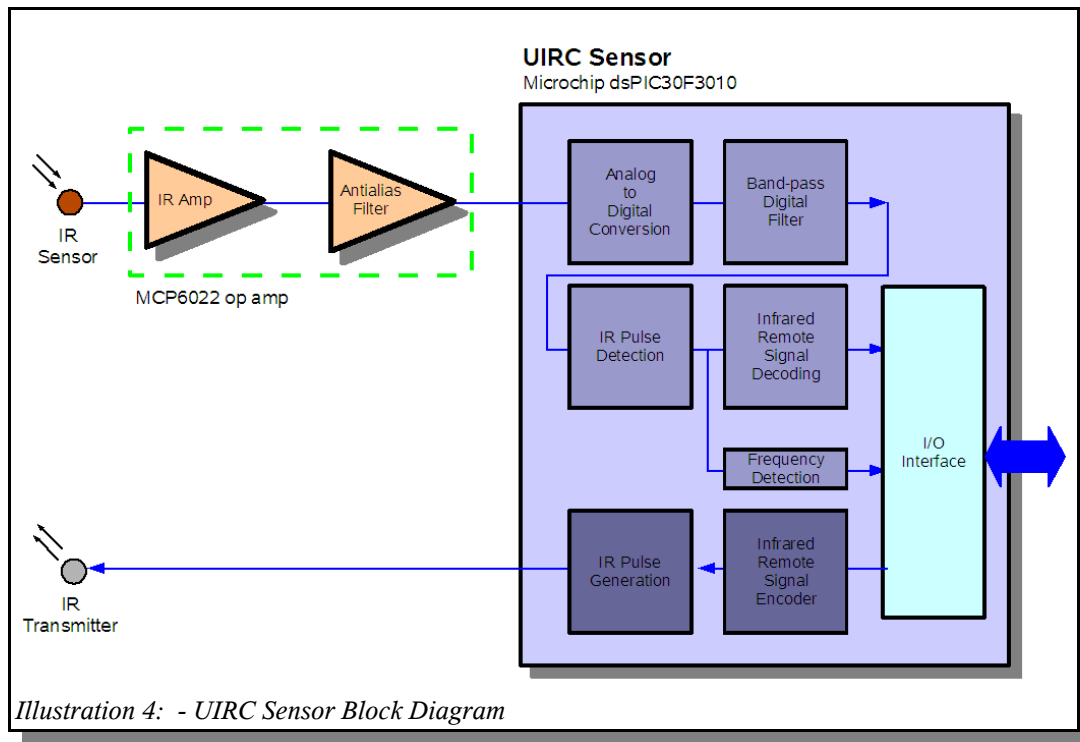


Illustration 4: - UIRC Sensor Block Diagram

The control system used in the dsPIC is through a Finite State Machine and Interrupts. A technique that permits a pseudo-task implementation between different parts without having to implement or use a real time OS (RTOS). This programming method is based on a state machine controlled by a variable. The variable tells the software which part of the code gains control. The variable is changed at the end of every task thus transferring the CPU time to other part of the code. The interrupt routines also change the state of the FSM.

The FSM in the UIRC sensor has five states: `FSM_STATE_START`, where the system prepares to receive a IR code. `FSM_STATE_CAPTURE` is the state where a signal is being digitized by the ADC. `FSM_STATE_BP_IIR` is activated when a block of ADC data is ready for filtering. In this state the IIR filter routine is invoked. `FSM_STATE_GET_RAW` is then activated to process the raw data to rectify the signal. Finally, the `FSM_STATE_IR_DECODE` is called to decode the rectified signal. These state run in that order: first the system is initialized, ADC capture is activated, band-pass filtering is performed every 16 captures (the size of the ADC buffer), and the rectification of the signal is performed.

The signal capture is interrupt based. Once the system is in the `FSM_STATE_CAPTURE` the FSM



will cycle continuously until the state is changed. This occurs when 16 samples are captured and the ADC interrupt is called. The samples are stored in the *adc\_buffer* and the state is change to *FSM\_STATE\_BP\_IIR*, forcing the FSM to call *bp\_filter()*. This routine calls *IIRTransposed* from the dsPIC30 DSP library, effectively performing a IIR digital filter for the samples stored at the *adc\_buffer*, and storing the filtered signal in the *output* array. The *bp\_iirFilter* structure has the filter parameters as designed with the dsPIC FD Lite software. This great tool generates the code for the filter coefficients and it is easily integrated in the C code. The following illustration shows the IIR filter design behavior:

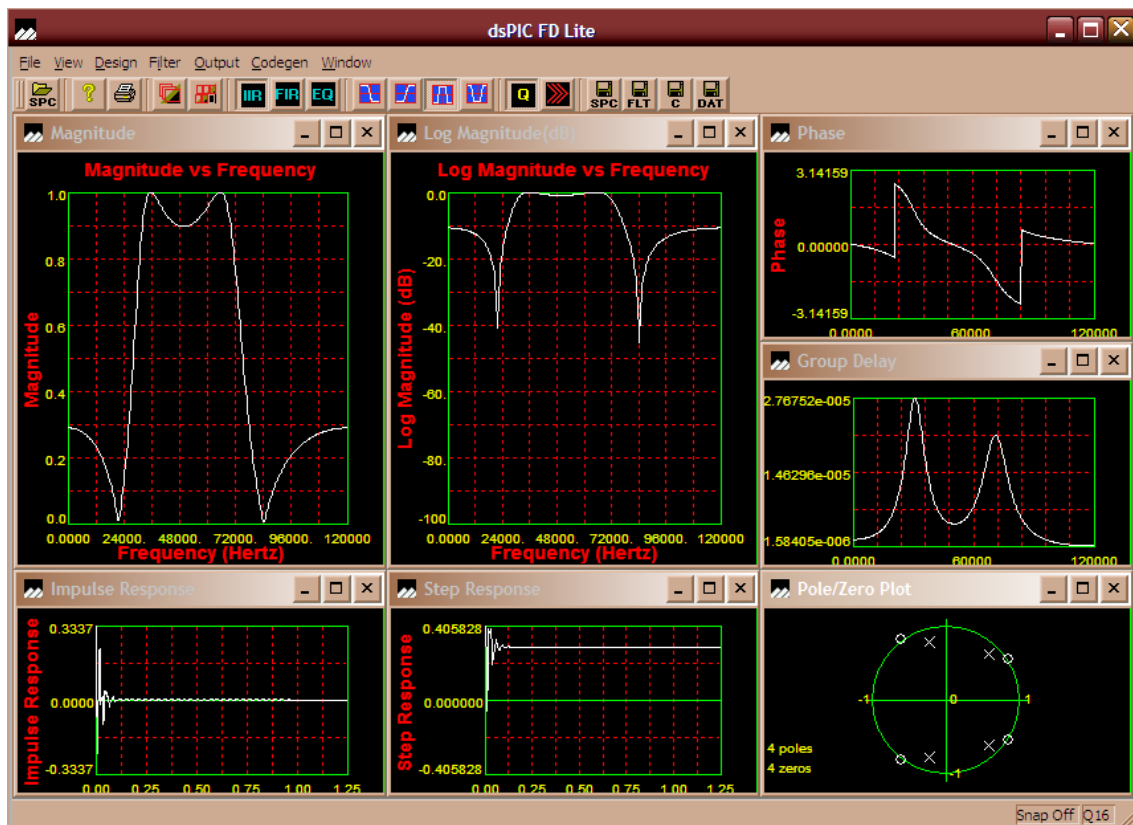


Illustration 5: The dsPIC FD Lite results for the IIR digital filter used in the UIRC sensor. This filter was designed as a 4th order, Elliptic band-pass filter for the band 30 KHz-70 KHz.

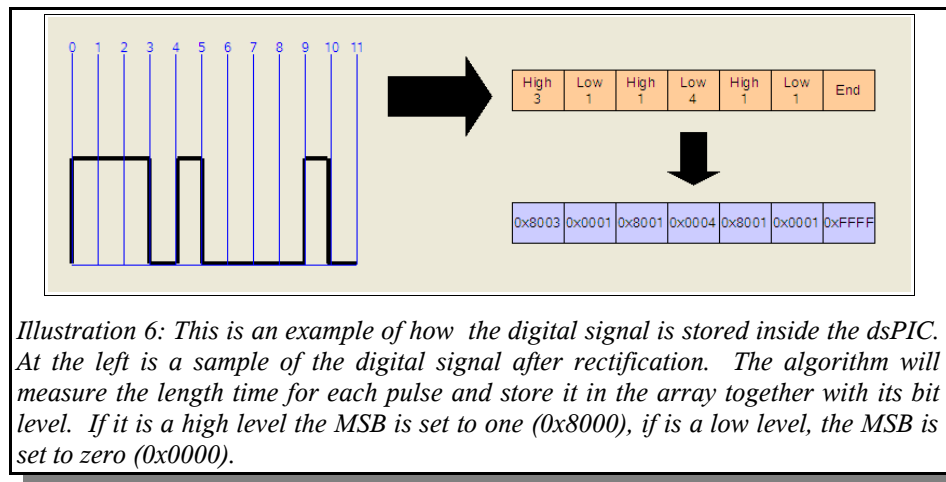
The IIR band-pass filter task (*bp\_filter*) finishes changing the FSM state to *FSM\_STATE\_GET\_RAW*, to run the next task, the *get\_raw\_signal* routine. This routine is responsible of detecting the high and low levels of the IR energy as captured in the *output* buffer.

Rectification here means the same as in the analog world: the positive and negative cycles are filtered to get the digital envelope of the digitized analog representation. This software implementation was tricky: the remaining time to perform these steps is very short before the next ADC interrupt. If not done quickly, the system won't be able to work in real time. The trick was to perform the filtering based on threshold events: every time that the signal exceeds a specific positive voltage (*THRESHOLD\_LEVEL*) a event is recorded. The events are saved in a unsigned integer as bits: every one bit signals a threshold exceeded event, zero bits are for the lower voltages. This “event stream”



will then work as a half-wave rectifier, recording the positive loops only at the threshold. Then an event-mask will compare the event stream to check if a lot of “zero events” have been detected. This is an indicator that no signal is present if the number of past events exceeds the cycle time of the lowest frequency allowed by the filter. With this technique it is possible to detect when a higher frequency is present, when is not and that “event stream” (stored at the *event* variable) will have a representation of the signal frequency. For instance a “0011100001” event stream is a representation of two low to high edges of the raw signal (the MSB is the oldest event). These are recorded as '01' bits in the stream. By counting the number of events between the transitions the frequency of the signal can be measured. This is done by the *measure\_freq()* function.

After this process is finished, an array with the timings between each rising or falling edge of the digital signal is stored in the *raw[]* array. Each position in this array stores the time to the next edge. The most significant bit (15) is set to 1 if the current level is positive or to 0 if the level is negative. To clarify this representation please see the following drawing:

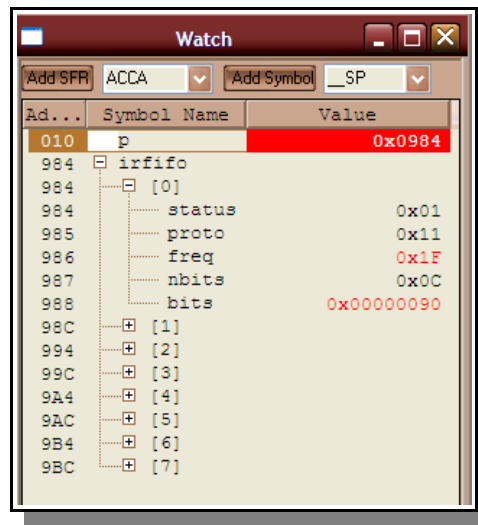


*Illustration 6: This is an example of how the digital signal is stored inside the dsPIC. At the left is a sample of the digital signal after rectification. The algorithm will measure the length time for each pulse and store it in the array together with its bit level. If it is a high level the MSB is set to one (0x8000), if is a low level, the MSB is set to zero (0x0000).*

The routines *count\_high()* and *count\_low()*, called inside the *get\_raw\_signal*, actually perform these measures. Every step in the *output* array corresponds to a sampling interval of about 4.069 us (246 KHz). So each count is a 4 us time step. All the measures of the UIRC sensor are in sampling interval units.

After the rectification process, the decode routine is run which decodes the internal representation. Based on the start bit and the normal bit lengths a protocol of the possible nine is recognized. The software then calls a specific protocol analysis routine which converts the measured time intervals into a stream of 0s and 1s that correspond to the digital command received. The final IR command is queued into the *irfifo* queue and transmitted to the UIRC Master Controller serially for further analysis. If the sensor is in learning mode, all of the measured parameters are sent. If the mode is in the sensing mode, only the frequency, protocol, number of bits and the bit commands are sent.

The following illustration shows a MPLAB capture of the irfifo structure just after a Sony Channel + command is received:



It can be seen that the protocol correspond to pulse width and SIRC (0x11), that the frequency is 31 KHz, 12 bits and the command is 000010010000 which corresponds to TV (00001) and Channel + (10000).

## The Master Controller and the Ethernet Interface

The master controller is based on a PIC24FJ64GA002 and in the demo code of the TCP/IP stack. It implements a TCP server at port 10000 to report received IR commands and to send IR commands to the UIRC sensor (this last feature was not ready for the contest). Several IP services are enabled: HTTP for a IR monitor web page for debugging, FTP for web page update in the external EEPROM, DHCP client for auto configuration, NetBIOS name service to get the assigned DHCP IP (the UIRC has the name 'uircsvr') and a ICMP client. The demo code was heavily modified to implement the communication with the UIRC sensor.

The receiving IR part of the Master Controller was finished in time for the contest. It receives any IR command from the UIRC sensor, converts it to ASCII and transmit it to a TCP connection established at port 10000. The following is a capture of a telnet session to the UIRC system:

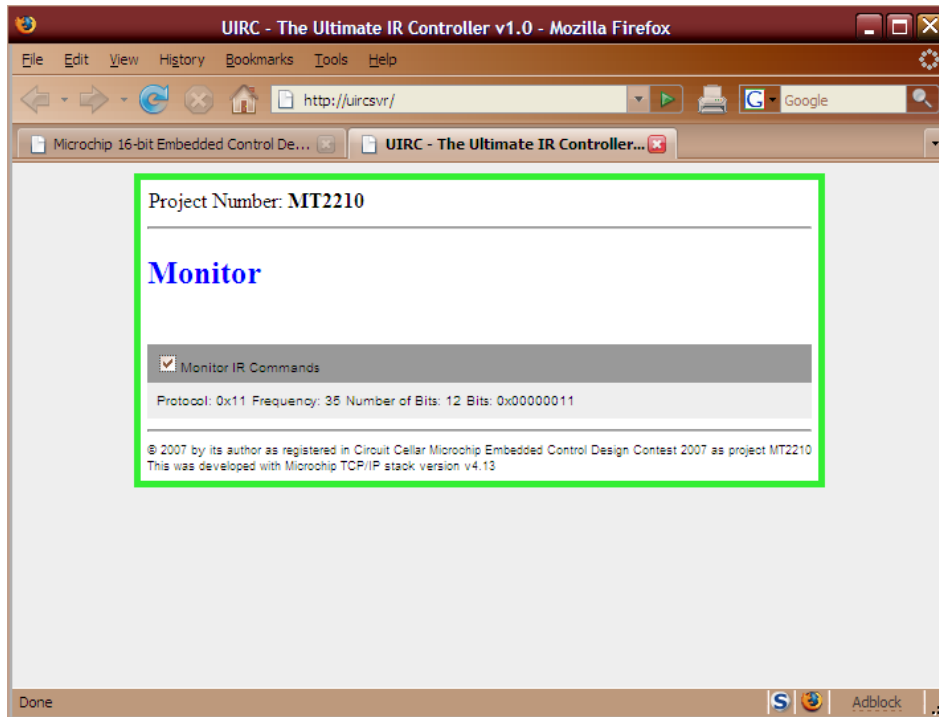
```
nas@qube:~$ nmblookup uircsvr
querying uircsvr on 172.16.255.255
172.16.1.102 uircsvr<00>
nas@qube:~$ ping uircsvr
ping: unknown host uircsvr
nas@qube:~$
nas@qube:~$ nmblookup uircsvr
querying uircsvr on 172.16.255.255
172.16.1.102 uircsvr<00>
nas@qube:~$ ping 172.16.1.102
PING 172.16.1.102 (172.16.1.102) 56(84) bytes of data.
64 bytes from 172.16.1.102: icmp_seq=1 ttl=100 time=2.06 ms
64 bytes from 172.16.1.102: icmp_seq=2 ttl=100 time=1.22 ms

--- 172.16.1.102 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 1.227/1.648/2.069/0.421 ms
nas@qube:~$ telnet 172.16.1.102 10000
Trying 172.16.1.102...
Connected to 172.16.1.102.
Escape character is '^]'.
E:11:35:12:00000015
E:11:35:12:00000015
E:11:35:12:00000015
E:11:35:12:00000015
E:11:35:12:00000015
E:11:35:12:00000015
E:11:35:12:00000015
E:11:35:12:00000001
E:11:35:12:00000001
E:11:35:12:00000003
E:11:35:12:00000003
E:11:35:12:00000003
E:11:35:12:00000003
E:11:35:12:00000003
E:11:35:12:00000003
E:11:35:12:00000003
E:11:35:12:00000003
E:11:35:12:00000005
E:11:35:12:00000005

^]
telnet> q
Connection closed.
nas@qube:~$
```

The commands received are sent in the format *E:proto:frequency:number of bits:bits*. So for instance the first received IR command above is *E:11:35:12:00000015*, which corresponds (look into the UIRC.h file for the protocol definitions) to protocol 0x11 (pulse width + Sony IR command SIRC), 35 KHz frequency, 12 bits and the command is 000000010101.

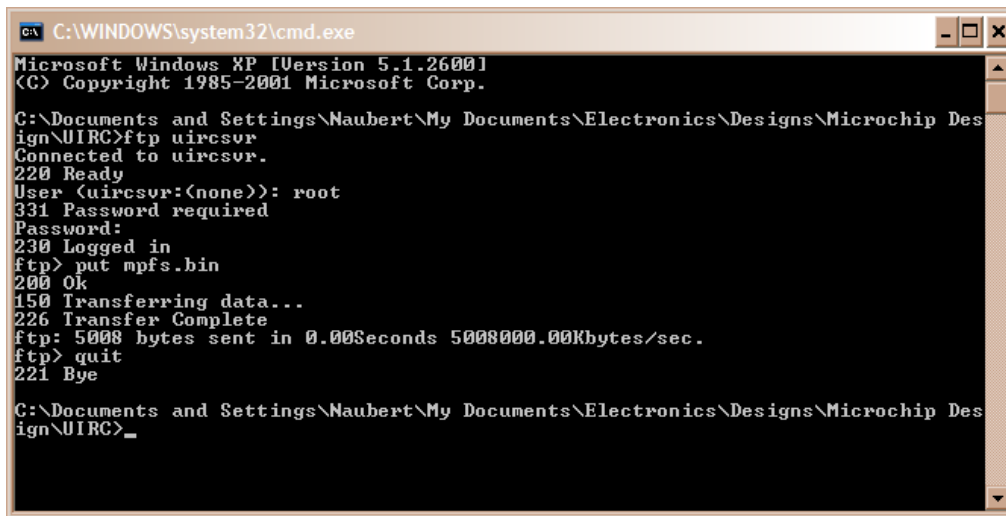
Besides the telnet interface, a web based monitor interface was developed with AJAX. A single page with a monitor interface will display the last IR command received in the screen:



*Illustration 7: UIRC Web Server for monitoring IR commands*

Parts of the demo code were modified to build this controller. The I2CEEPROM.c was taken from the version 4.02 of the TCP/IP stack, fixed and used in the 4.12 version of the stack to communicate with the I2C Microchip EEPROM. The main code routines of HTTP, ICMP, FTP were copied and modified to fit the project. Obviously all TCP/IP stack configuration files were adapted to enable the required features and to update the pinout. A custom board was defined in the TCPIPConfig.h. In the main code the initialization routine was modified to assign the required pins and enable the hardware resources. A couple of new tasks were defined: *ConnectionTask()* that handles the TCP connection, and *ProcessIO()*, which reads the IR events from the UIRC sensor. The HTTPGetVar was modified to return the last IR command for the first variables, so the AJAX code can report in the web page the last received IR command.

The following is a screen capture of the web page update via FTP:



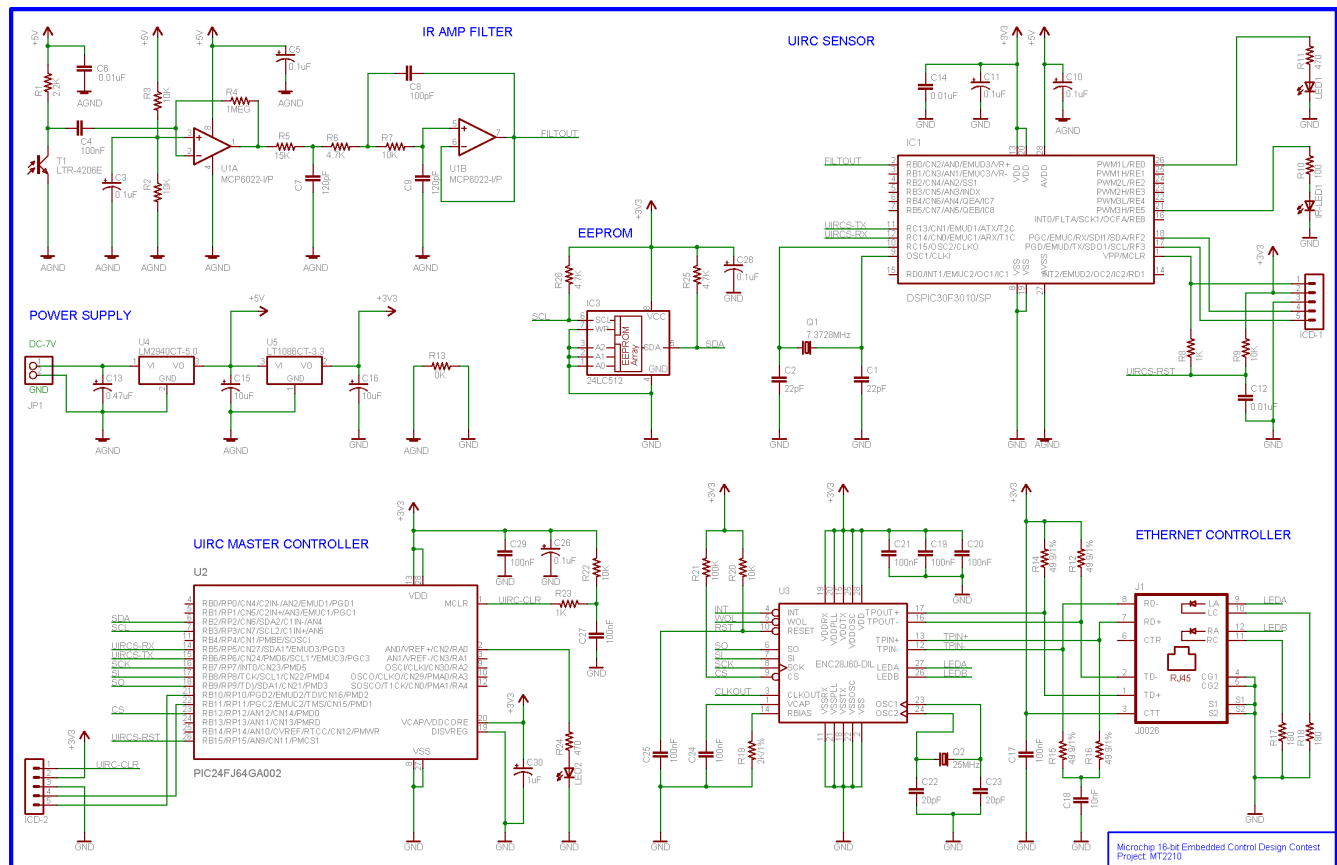
```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Naubert\My Documents\Electronics\Designs\Microchip Design\UIRC>ftp uircsvr
Connected to uircsvr.
220 Ready
User (uircsvr:(none)): root
331 Password required
Password:
230 Logged in
ftp> put mpfs.bin
200 Ok
150 Transferring data...
226 Transfer Complete
ftp: 5008 bytes sent in 0.00Seconds 5008000.00Kbytes/sec.
ftp> quit
221 Bye

C:\Documents and Settings\Naubert\My Documents\Electronics\Designs\Microchip Design\UIRC>_
```

## UIRC Schematic

Below is the full schematic of the UIRC. A higher resolution image is in the Documents folder in the ZIP file of the contest entry.



*Illustration 8: - The UIRC System Schematics: the five sections are, from the top left clockwise: the IR amplifier and anti alias filter; 12C EEPROM for web page storage; a dsPIC30F3010 is the UIRC Intelligent Sensor decodes and encodes IR signals in real time, the Ethernet Controller uses a ENC28J60 and the UIRC Master Controller is a PIC24FJ64GA002 which integrates everything to provide IR receive and send functions through TCP.*

## UIRC Code Sample

The following code shows three routines of the UIRC Smart Sensor. The first one is the ADC interrupts, which is invoked at the end of 16 ADC conversions. It transfers the capture data to the buffer to prepare the data for the DSP filtering. The second one, *bp\_filter()*, is the digital filter routine. This routine is called by the finite state machine (FSM) once data is in the buffer. The *get\_raw\_signal()* routine then processes the raw data after the digital filtering and splits the IR signal in its components: start bits, maximum and minimum bit lengths, etc. After all of these parameters are measured, the FSM calls the *decode()* routine which will analyze the timings and process the corresponding IR protocol.

```
// ADC interrupt handler: read the ADC result into the corresponding buffer block
// this requires at least -O1 optimization and loop unroll
void _ISR_ADCInterrupt(void)
{
    int i;
    fractional *dst;

    // if sampling is disabled and an interrupt arrived, signal error
    if (!sampling) {
        StopSampling();
        IFS0bits.ADIF = 0;
        while (1) {
            // turn on debug LED
            LED0 = 1;
            // turn off debug LED
            LED0 = 0;
        }
    }

    // read the 16 ADC buffer positions and copy them to the ADC buffer
    dst = adc_buffer;
    for (i = 0; i < 16; ++i)
        *dst++ = *((fractional *)&ADCBUF0+i);

    // ADC block is filled, then signal the IIR process
    fsm = FSM_STATE_BP_IIR;
    sampling = SAMPLING_OFF;                                /* this is for security: if the interrupt arrives */
                                                            /* before IIR process is ready, a fault is detected */

    // clear the ADC interrupt flag
    IFS0bits.ADIF = 0;
} // _ADCInterrupt

// bp_filter
// band-pass IIR filter for wideband IR decoding. This IIR will pass the band from 30 KHz to 80 KHz
inline void bp_filter(void)
{
    // call IIR filter
    IIRTransposed(MAX_ADC_BUFFER, output, adc_buffer, &bp_iirFilter);

    // restart capture
    sampling = SAMPLING_ON;
    fsm = FSM_STATE_GET_RAW;
}

// ADC interrupt handler: read the ADC result into the corresponding buffer block
// this requires at least -O1 optimization and loop unroll
void _ISR_ADCInterrupt(void)
{
    int i;
    fractional *dst;

    // if sampling is disabled and an interrupt arrived, signal error
    if (!sampling) {
        StopSampling();
        IFS0bits.ADIF = 0;
        while (1) {
            // turn on debug LED
            LED0 = 1;
            // turn off debug LED
            LED0 = 0;
        }
    }

    // read the 16 ADC buffer positions and copy them to the ADC buffer
    dst = adc_buffer;
    for (i = 0; i < 16; ++i)
        *dst++ = *((fractional *)&ADCBUF0+i);
}
```



```

// ADC block is filled, then signal the IIR process
fsm = FSM_STATE_BP_IIR;
sampling = SAMPLING_OFF;                                /* this is for security: if the interrupt arrives */
                                                         /* before IIR process is ready, a fault is detected */

// clear the ADC interrupt flag
IFS0bits.ADIF = 0;

} // _ADCInterrupt

// bp_filter
// band-pass IIR filter for wideband IR decoding. This IIR will pass the band from 30 KHz to 80 KHz
inline void bp_filter(void)
{
    // call IIR filter
    IIRTransposed(MAX_ADC_BUFFER, output, adc_buffer, &bp_iirFilter);

    // restart capture
    sampling = SAMPLING_ON;
    fsm = FSM_STATE_GET_RAW;
}

// get_raw_signal
// rectify the digital signal and perform the low-high timings on it
void get_raw_signal(void)
{
    unsigned char i;
    unsigned char f;

    // continue capturing if the IR signal is not completely received
    fsm = FSM_STATE_CAPTURE;

    // check if energy in the pass band
    // test that two consecutive absolute value of samples exceeds threshold
    for (i = 0; i < MAX_ADC_BUFFER; ++i) {
        // check if threshold exceeded
        f = ((int)output[i] > THRESHOLD_LEVEL)?1:0;
        // count the number of last zero events
        if (f) zeros = 0; else ++zeros;
        // remember number of threshold exceeded, if one event exceeds threshold
        // in a EVENT_N event window then signal energy (this is like a bitwise low pass filter)
        event = (event << 1 | f);

        if (event & EVENT_MASK) {
            // output signal for debugging
            DEBUG_OUT = 1;

            // count high signal
            count_high();

            // capture event bitmap for frequency detection
            // this capture occurs when a 01 transition is detected twice
            if (!event_freq && (event&0xfffc) && (event&0x03) == 0x01)
                event_freq = event;

        } else {
            // output signal for debugging
            DEBUG_OUT = 0;

            // count low signal
            count_low();

        } // if sum */
    } // for */
} /* get_raw_signal */

/*
 * decode:
 * analyzes the timing and decodes the IR signal into an IR command according
 * to a protocol
 */
void decode(void)
{
    int ok;

    // if # of bits < 3, ignore command
    if (raw_pos < 6) {
        fsm = FSM_STATE_START;
        return;
    }

    // fix second to last high levels
    if (high2_max == 0) high2_max = high_max;
    if (high2_min == 65535) high2_min = high_min;
    if (low2_max == 0) low2_max = low_max;
    if (low2_min == 65535) low2_min = low_min;

    // decode signal
    freq = measure_freq();

    // check IR protocol

```

```

// first, look for biphase protocols
if (compare(start_high, FROM_US(2666)) && compare(start_low, FROM_US(889)))
    protocol = IR_PROTO_RC6|IR_ENCODING_BI;
else if (compare(start_high, FROM_US(500)) &&
        (compare(start_low, FROM_US(2500)) || compare(start_low, FROM_US(3000))))
    protocol = IR_PROTO_NRC17|IR_ENCODING_BI;
else if (compare(start_high, FROM_US(889)) && compare(start_low, FROM_US(889)))
    protocol = IR_PROTO_RC5|IR_ENCODING_BI;

// check for PW protocol
else if (compare(start_high, FROM_US(2400)) && compare(start_low, FROM_US(480)))
    protocol = IR_PROTO_SIRC|IR_ENCODING_PW;
else if (compare(start_high, FROM_US(4000)) && compare(start_low, FROM_US(4000)))
    protocol = IR_PROTO_RCA|IR_ENCODING_PW;

// check for PD protocol
else if (in_range(start_high, FROM_US(8400), FROM_US(9000)) && compare(start_low, FROM_US(4200)))
    protocol = IR_PROTO_JVCNEC|IR_ENCODING_PD;
else if (compare(start_high, FROM_US(320)) &&
        (compare(start_low, FROM_US(1680)) || compare(start_low, FROM_US(680))))
    protocol = IR_PROTO_SHARP|IR_ENCODING_PD;

// if still unknown, check for generic protocols
else if ((protocol&IR_PROTO_MASK) == IR_PROTO_UNKNOWN) {
    // pulse distance
    if (compare(high2_max, high2_min))
        protocol = IR_PROTO_PD|IR_ENCODING_PD;
    else if (compare((high2_max+low2_min), (high2_min+low2_max)))
        protocol = IR_PROTO_PW|IR_ENCODING_PW;
}

// process the corresponding protocol and fill the output structure
ok = 0;
switch (protocol&IR_ENCODING_MASK) {
    case IR_ENCODING_PW:      process_pw(); ok = 1; break;
    case IR_ENCODING_PD:      process_pd(); ok = 1; break;
    case IR_ENCODING_BI:      process_bi(); ok = 1; break;
    default: break;
}

// if ok then fill structures for output
if (ok == 1) {
    // if sensing and enough buffer, queue it
    if (mode == SENSING && ir_buf_n < IRFIFO_LEN) {
        IRevent *p = &irfifo[ir_buf_last];
        p->status = UIRCS_RES_IREVENT;
        p->proto = protocol;
        p->freq = freq;
        p->nbits = nbits;
        p->bits = bits;
        ir_buf_last = (ir_buf_last+1)%IRFIFO_LEN;
        ++ir_buf_n;
    }

    // if in learn mode, fill learn structure and stop
    if (mode == LEARNING) {
        IRlearn *p = &ir_learn_evt;
        p->status = UIRCS_RES_IRLEARN;
        p->proto = protocol;
        p->freq = freq;
        p->nbits = nbits;
        p->bits = bits;
        p->flags = 0;
        p->start_high = start_high;
        p->start_low = start_low;
        p->high_max = high2_max;
        p->high_min = high_min;
        p->low_max = low2_max;
        p->low_min = low_min;
        p->gap = gap;

        // set mode to idle to leave time to transmit
        mode = LEARNING_OK;
    }
}

// restart
fsm = FSM_STATE_START;
}

```

## References

- **Microchip dsPIC30F3010/3011 Data Sheet.** High Performance 16-bit Digital Signal Controllers. DS70141D
- **Microchip PIC24FJ64GA004 Family Data Sheet.** 28/44-Pin General Purpose, 16-bit Flash Microcontrollers. DS39881B.
- **Microchip ENC28J60 Data Sheet.** Stand-Alone Ethernet Controller with SPI Interface. DS39662B
- **Microchip MCP6021/1R/2/3/4,** Rail-to-Rail Input/Output, 10 Mhz Op Amps. DS21685C.
- **Microchip 24AA512/24LC512/24FC512,** 512K I<sup>2</sup>C CMOS Serial EEPROM. DS21754G.
- **Microchip AN699,** Anti-Aliasing, Analog Filters for Data Acquisition Systems. DS00699B.
- **Microchip 16-Bit Language Tool Libraries,** DS51456C.
- **Microchip AN833,** The Microchip TCP/IP Stack. DS00833B.
- **Olimex ENC28J60 Header Board Schematic** (<http://www.olimex.com/dev/enc28j60-h.html>)
- San Bergmans's IR protocol web pages: <http://www.sbprojects.com/knowledge/ir/ir.htm>
- Circuit Cellar's Microchip 16-Bit Embedded Control Design Contest CD.