Introducing the 32 bit Micro Experimenter

In a 2010, Nuts and Volts introduced the 16 bit Micro Experimenter with a seven article series. The 16 bit Experimenter offered the readership a new and significant microcontroller capability when compared with current offerings of 8 bit technologies in both performance, and functionality. Well get ready this year, Nuts and Volts are going to raise the bar with an even more powerful capability the “32 bit Micro Experimenter” or, “Experimenter” for short.

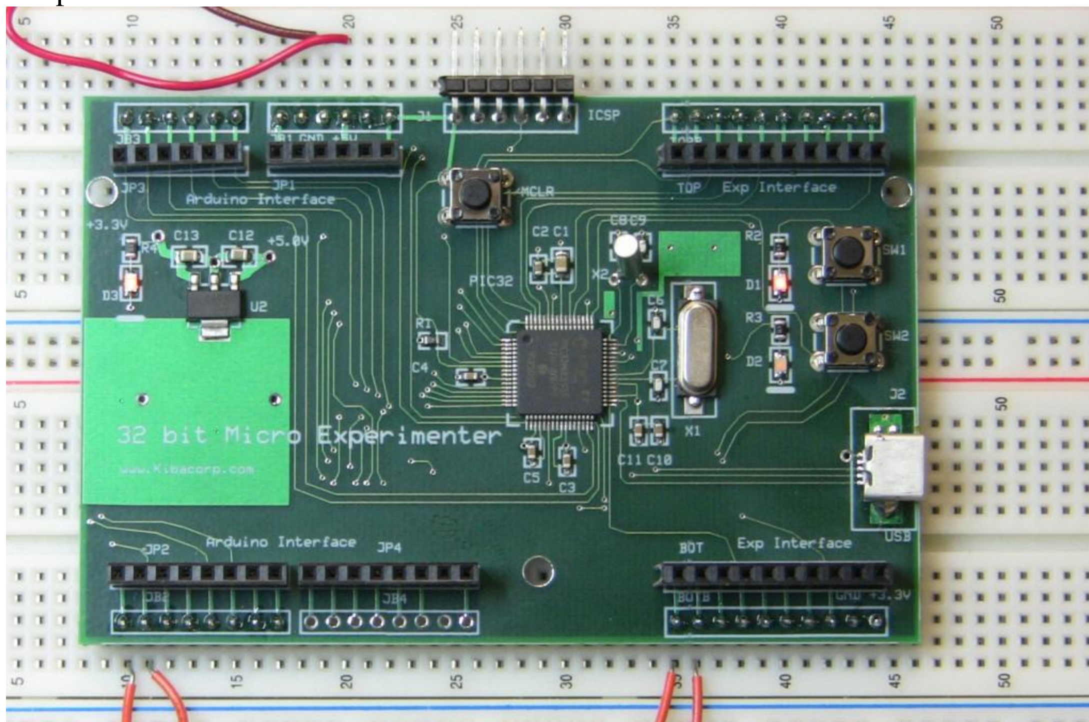


Figure 1 the 32 Bit Micro Experimenter

The Experimenter uses the Microchip 32 bit family of microcontrollers, leveraging what the followers of this series has learned so far with the 16 Bit Experimenter, making the transition to 32 bit easier. This makes sense because of the significant overlap between Microchip’s 32 bit family and their 16 bit family. In fact, the PIC32, Microchip’s new 32 bit offering, supports all the Experimenter’s 16 bit PIC24F peripheral operations as well as a new set of atomic bit operations (more on those in later articles). The 32 bit development tool suite like the MPLAB IDE, PICKIT3, and Microchip ‘C’ Compiler family are similar to those used with the 16 bit set. In addition peripheral programming has been simplified with a new Microchip ‘C’ 32 bit program peripheral library set. Again, as with the 16 bit Experimenter, the 32 bit Experimenter requires some familiarity with ‘C’ language at a high level.

With the Experimenter we will experience a whole new level of Applications. Look for embedded web control, use of Real Time Operating Systems (RTOS), USB, speech playback, high speed (100 MHz) Ethernet, and high resolution graphics—all of which we have come to expect in today's top end multi-media consumer's products. We will explore these together, in an upcoming series of articles, using the 32 bit Micro Experimenter.

The PIC32 Microcontroller–an Overview

Microchip's 32 bit Microcontroller, the PIC32, is the top performer in Microchip's product family, offering significant enhancements in speed, memory capability and performance over all other members of the Microchip product family. If you require high end performance for your microcontroller application then the PIC32 is the processor for you. Let's do some comparisons. Let's compare the Microchip PIC32MX695F512H microcontroller (PIC32 bit processor used with the 32 bit Micro Experimenter), to the PIC24FJ64GA002 (the 16 bit microcontroller used on the 16 bit Micro Experimenter), to the PIC16F887 (common 8 bit microcontroller).
In the table shown below, MIPS (Millions of Instruction per second) metric is used to capture execution performance. As shown, the PIC32 runs 5X faster than the PIC24F and 16X faster than the PIC16. The PIC32 also executes instructions with much larger data words 32 bits, versus 16 and 8 bits used by the other family members. Finally the PIC32 has the largest compliment of flash and ram, allowing it to tackle bigger program applications.

PIC32	PIC24F	PIC16
32 bit	16 bit	8 bit
512K Flash	64 K Flash	8k Flash
128K RAM	8K RAM	386 RAM
80 MHZ CLOCK	32 MHZ	20 MHZ
80 MIPS	16 MIPS	5 MIPS
3.3V operation	3.3v operation	5 V operation

.

A block diagram of the PIC32 shows the internals of the chip. The PIC32 as shown is the table is a +3.3V part, however it can accept +5V logic level inputs on certain pins (see Figure 2 +5V tolerant pins) without damage. The chip is organized around two internal buses. The top bus is the faster of the two and runs at the system clock CPU rate. It allows simultaneous communications for devices on this bus without any bus contention. Some of devices here are the 32 Core CPU, the 8 channel DMA (Direct Memory Access Controller), and USB. Other components on this bus are flash, ram, interrupt controller, all the digital ports, and a peripheral bridge (a connection to all the on –chip peripherals). It is interesting to note that digital ports reside on the high speed bus. This means they can be toggled (on/off) at the 80 MHz rate ---able to generate digital signals of up to 40MHz if needed. With the use of the DMA, this port (or any other peripheral), can

directly access memory directly without CPU (software) intervention for data transfers. We will show the DMA in action with one of the experiments listed in this article.

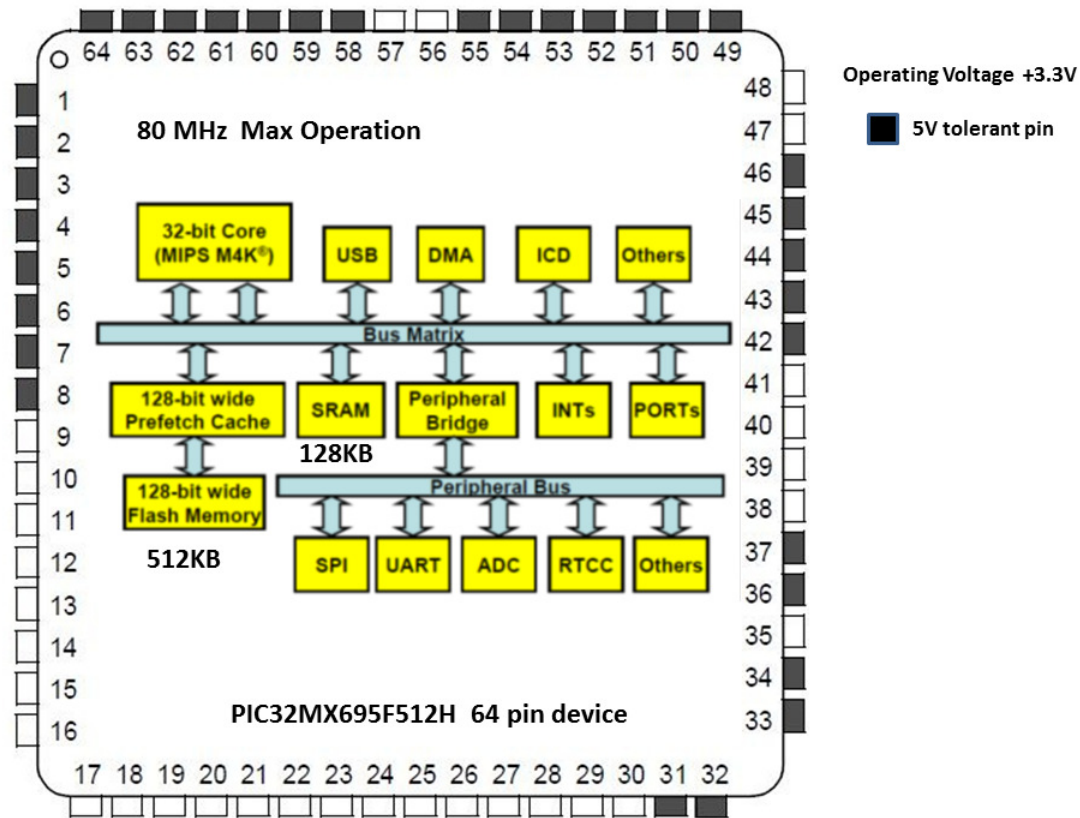


Figure 2 PIC32 Block Diagram

The other bus is the peripheral bus. This bus does not need to run as fast as the previous bus and can be slowed down to a factor of 1 to 8 (peripheral clock rates are not required to be run as high as 80 MHz). The peripheral bus connects to all the on-chip peripherals. With the PIC32MX695F512H there are 5 16 bit Timers, a RTCC (Real Time Clock Calendar), 6 UART (Universal Asynchronous Receiver Transmitter), 3 SPI (Serial Peripheral Interfaces) , 4 I^2C (Inter-Integrated Circuits) , 16 Channel 10 bit ADC (Analog-to –Digital Converter) , 1 PMP (Parallel Master Port), 5 CCP (Compare/Compare Ports), and 2 Analog Comparators. The Experimenter provides allows us interface access to all these capabilities.

An Overview of 32 Bit Experimenter Hardware

A block diagram of the Experimenter is shown; schematics are also included at the end of this article. The board has extensive expansion capabilities. This approach allows access to all the PIC32MX695F512H I/O for experiments.

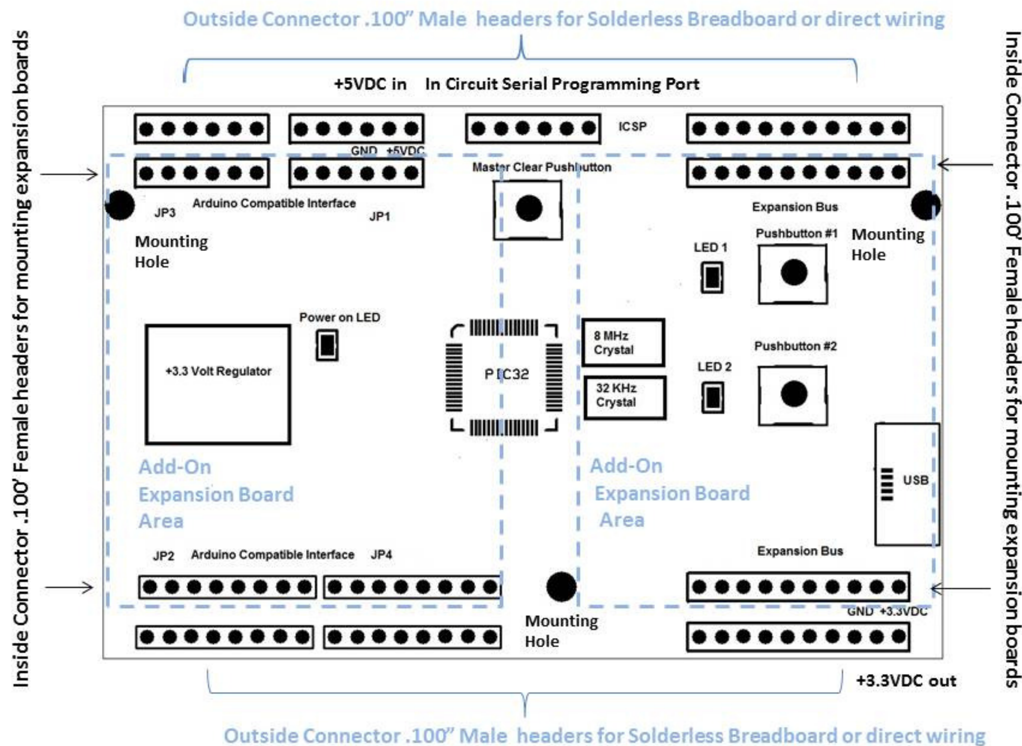
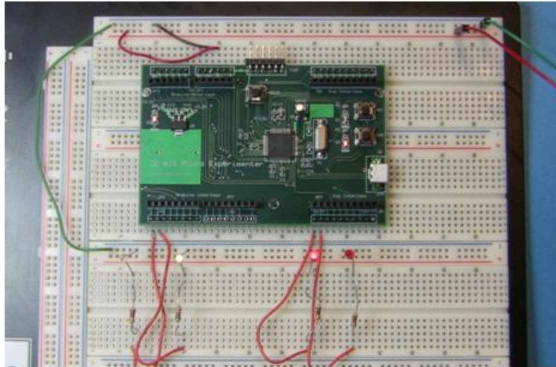


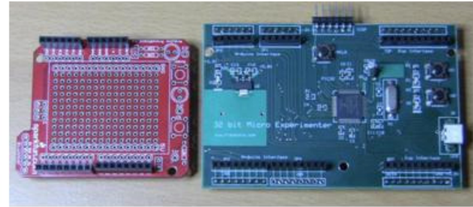
Figure 3 EXP32 Block Diagram

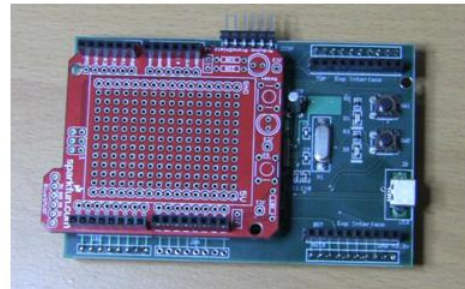
The Experimenter inside connectors are .100" female headers. This allows mounting expansion cards right on top of the Experimenter (vertical expansion). The left side female header top and bottom set has an "Arduino Expansion " footprint that is mechanically and electrically compatible with Arduino expansion cards, the minor exceptions to interface are shown in red (see Figure 5). These expansion cards are available from a number of vendors and this feature allows users the opportunity to integrate a number of existing Arduino expansion cards with their Experimenter applications. Keep in mind that new software drivers will have to be rewritten in Microchip 'C' to accommodate any of these boards. The Experimenter also uses an additional set of female connectors top/bottom on right side to accommodate non-Arduino expansion boards.

For horizontal expansion (typical of Solderless bread boarding) the outside connector rows sets support .100" male headers. These are used to mount the Experimenter to a large Solderless breadboard (3260 contacts) to accommodate direct wiring prototyping. Finally, the board has three mounting holes that provide mechanical support for vertical expansion or mounting the entire Experimenter to a chassis.

Expansion Capabilities (48 Pins)







Using Solderless breadboard

Vertical stacking

Figure 4 Experimenter Expansion

The Experimenter requires a +5VDC input (500 ma max current) and does on-board regulation of this +5VDC to generate the necessary +3.3VDC for the PIC32. A power up LED signifies that +3.3VDC is up and running. The +3.3VDC is also made available for off board use.

The Experimenter board is programmed and debugged using the standard Microchip six pin ICSP (In Circuit Serial Programming Interface). This interface directly accepts a PICKIT3 programmer/debugger or any of the other Microchip programmers/debuggers (with exception of the PICKIT2).

An 8MHz crystal serves as a CPU clock source and is electronically multiplied inside the chip for 80MHz operations and for USB clock requirements. The Experimenter also has a fully USB 2.0 compliant interface. USB applications will be covered in detail in subsequent articles.

An optional 32 KHz crystal can be added to serve as a precision clock source for the PIC32 Real Time Clock Calendar (RTCC) peripheral for accurate data/time operations.

There is a minimal user interface for manual reset, two software controlled LEDs and readable pushbutton switches to support debugging and experiments.

Power	JP1	ARDUINO	PIC32 EXP
	1	Reset	/MCLR (7)
	2	3.3V	3.3V
	3	5V	5V
	4	gnd	gnd
	5	gnd	gnd
	6	vin	not used

	JP3	ARDUINO	PIC32 EXP
Analog	1	ADC0/PC0	AN0 (16)
	2	ADC1/PC1	AN1(15)
	3	ADC2/PC2	AN2(14)
	4	ADC3/PC3	AN3(13)
	5	ADC4/SDA/PC4	SDA3A(31)
	6	ADC5/SCL/PC5	SCL3A(32)

TOP		PIC32 EXP
1	TSP1	RE7(3)
2	TSP2	RE4(64)
3	TSP3	RE3(63)
4	TSP4	RE2(62)
5	TSP5	RE1(61)
6	TSP6	RE0(60)
7	TSP7	RF1(59)
8	TSP8	RF0(58)
9	TSP9	RD7(55)
10	TSP10	RD6(54)
BOT		**PIC32 EXP**
1	BSP1	RB9(21)
2	BSP2	RB10(22)
3	BSP3	RB14(29)
4	BSP4	USBID(33)
5	BSP5	RD9(43)
6	BSP6	RD11(45)
7	BSP7	RD1(49)
8	BSP8	RD5(53)
9	GND	GND
10	3.3V	3.3V

	JP2	ARDUINO	PIC32 EXP
Digital1	1	RXD/PD0	U1ARX(50)
	2	TXD/PD1	U1ATX(51)
	3	INT0/PD2	INT0 (46)
	4	INT1/PD3	INT1(42)
	5	T0/PD4	NOT0 RE5(1)
	6	T1/PD5	NOT1 RE6(2)
	7	AIN0/PD6	ADN0 C1IN+ (11)
	8	AIN1/PD7	ADN1 C1IN-(12)

	JP4	ARDUINO	PIC32 EXP
Digital2	1	ICP/PB0	IC3(44)
	2	OC1A/PB1	OC5(52)
	3	OC1B/SS/PB2	/SS2A (8)
	4	MOSI/OC2/PB3	SDO2A (6)
	5	MISO/PB4	SDI2A (5)
	6	SCK/PB5	SCK2A (4)
	7	GND	gnd
	8	AREF	NAREF AN8 (21)

Red –not totally compatible

Figure 5 Experimenter 48 pin Expansion Interface

The Experimenter is available in both kit form and fully assembled from the Nuts and Volts web store. The Experimenter is largely surface mount so in both kit and other offerings, the Surface Mount assembly is complete. The only “kit” aspects are mounting and soldering the female and male headers onto the board, as well as the 32 KHz crystal.

Setting up your Tool Suite

This couldn’t be any easier. Go to the Microchip web site and download their latest version of MPLAB. You will also need to download their free PIC32 ‘C’ Complier and install both software packages. All supplied demo should be easily built and compiled. The only choices you have are the method of programming and debugger hardware. It comes down to the following choices: PICKIT3, MPLAB ICD3, or REALICE. All choices are viable with PICKIT3 being the cheapest. The Experimenter comes pre-configured with an LED blink program that alternately turning off/on each of the two on-board LEDs. This should run “right out of the box” once you apply +5VDC and Ground to JP1 or JB1 connectors. Demo Source code, in the form of Microchip projects (.MCP files), is available on the CD-ROM supplied with the kit or downloadable from the Nuts and Volts web site. The Demos allow you to exercise your tool suite to build, compile and download working code into your Experimenter.

Hello Word Demos

If you have never heard of ‘C’ language, you might have heard of the famous “Hello World!” programming example. This is a single line program execute by beginner programmers to display “Hello World!” on a computer screen. So let’s be realistic; we are programming a microcontroller that does not have a computer screen associated with it, so we are limited initially with the Experimenter digital I/O. Fortunately, some of the I/O is connected to its on board LEDs. In other parts of this series we will show the use of USB to write to the computer screen: however, for now we will use LEDs. To make our “Hello World” meaningful we will start with the on-board LEDs and then advance to a

5x7 LED Matrix for more exciting light shows that demonstrate the use of a Real Time Operating System and the PIC32 DMA controller. Here's the run down:

- Demo 1 - blinky LEDs –alternate blinking on board LEDS (BLINKY.MCP source)
- Demo 2 - Turn on LED with pushbutton (PUSHBTN.MCP source)
- Demo 3 - RTOS individual LED control (RTOS.MCP source)
- Demo 4- DMA transfer to LED (DMA.MCP source)

The particular 5x7 matrix is shown for Demos 3 and 4, along with the hook-up diagram for the Experimenter. Let's start working through the different experiments.

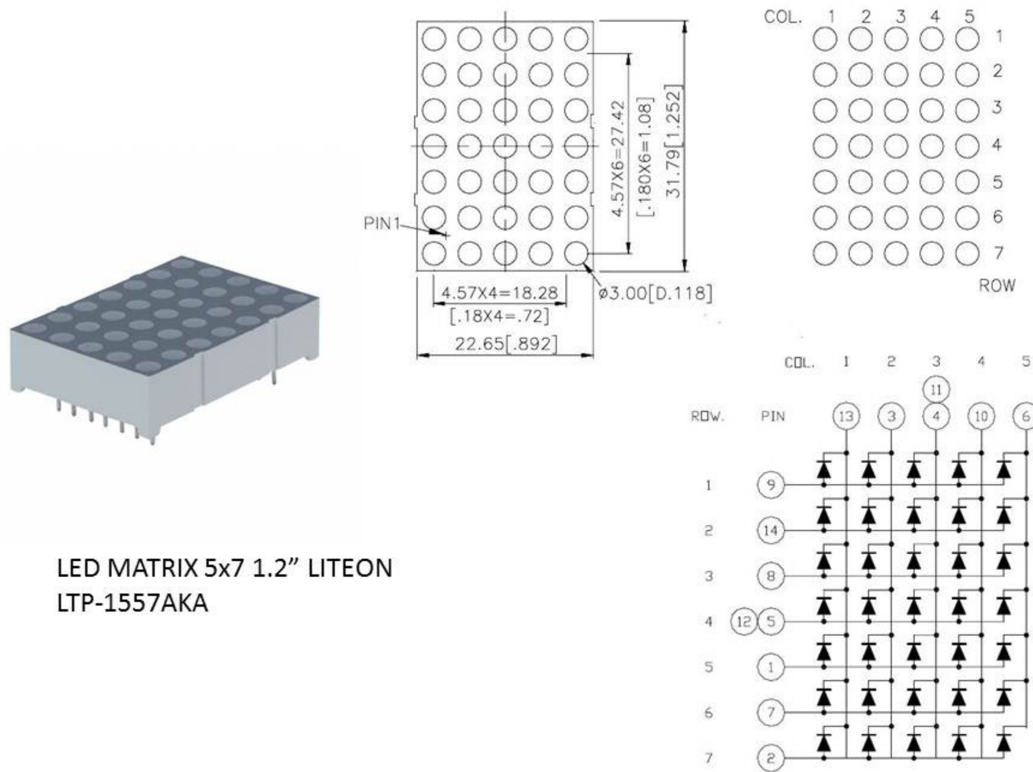


Figure 6 LED Matrix

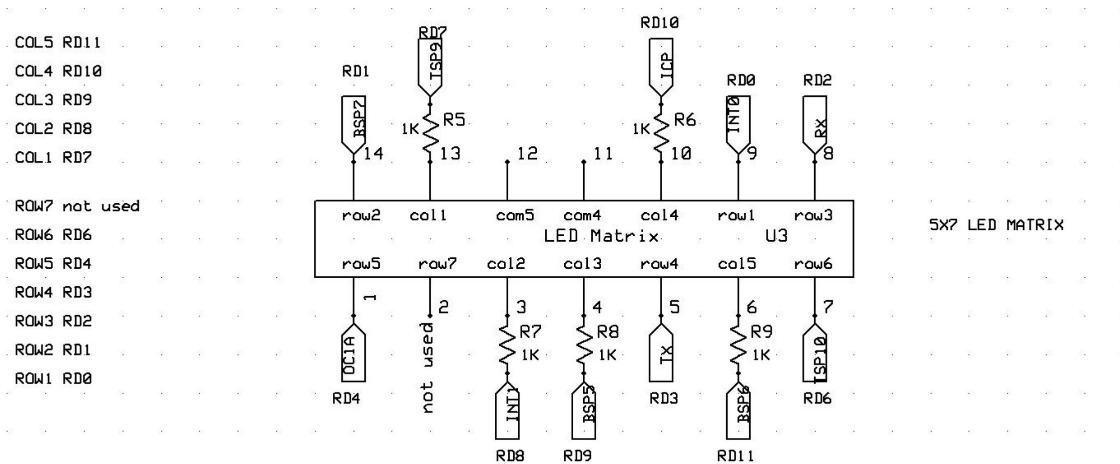


Figure 7 LED Matrix Hook up

Demo 1 “Hello World” - blinky LEDs

The first Demo is found in the Blinky folder, so open the folder and double click BLINKY.MCP . This should invoke MPLAB and bring up the Blinky project. Select your programmer/debugger, compile the project, and download the code to the Experimenter with your programmer/debugger hook to ICSP. You should see the two LEDs on the Experimenter blink alternately. Let’s examine the code. All these programs use **#include <plib.h>** to reference the PIC32 library and part. We then set the fuses for 80MHZ operation and the peripheral bus to 40MHZ, then we invoke a library function “**SystemConfig()”**to optimize PIC32 operation. This function sets pre-fetch cache, ram, and flash wait states for the 80MHZ clock. Finally, we configure the digital I/O for LEDs and enter a continuous loop using toggle function and a software delay for blink.

```
#include <plib.h>
// Configuration Bit settings
// SYSCLK = 80 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care
#pragma config FNOSC = PRIPLL, POSCMOD = HS, FPLLIDIV = DIV_2, FPLLMUL = MUL_20, FPBDIV = DIV_2, FPLLODIV = DIV_1
#pragma config FWDTEN = OFF
#define SYS_FREQ    80000000        // frequency we're running at

int main(void)
{   int i;
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // Configure the device for maximum performance but do not change the PBDIV
    // Given the options, this function will change the flash wait states, RAM
    // wait state and enable prefetch cache but will not change the PBDIV.
    // The PBDIV value is already set via the pragma FPBDIV option above..
    SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

    // L2 RB11 an output
    // L1 RB12 an output-- Turn them off before changing
    mPORTBClearBits(BIT_12 | BIT_11 );
    mPORTBSetBits(BIT_11 );
    mPORTBSetPinsDigitalOut( BIT_12 | BIT_11 );

    // Now blink all LEDs ON/OFF forever.
    while(1)
    {
        mPORTBToggleBits(BIT_12 | BIT_11  );

        // Insert some delay
        i = 1024*1024;
        while(i--);
    }
}
```

PIC32 and Peripheral Library

Set Fuse Configurations

Library function to configure system for max performance at 80 MHZ

Configure Digital I/O connected to LEDs

Toggle LEDs

Loop

Software Delay

Figure 8 Blinky Code

Demo 2 “Hello World” -Pushbutton LED

This demo is similar in structure to demo 1 but responds to either of the two Experimenter pushbuttons being depressed by lighting the corresponding LED associated with the button. The code is shown. The project is located in the PUSHBUTTON Folder. Note that we set a “weak pull up” feature internal to the chip that provides a resistor to +3.3V to each switch input.

```
#include <plib.h>
#pragma config FNOSC = PRIPLL, POSCMOD = HS, FPLLIDIV = DIV_2, FPLLMUL = MUL_20, FPBDIV = DIV_2, FPLLODIV = DIV_1
#pragma config FWDTEN = OFF
#define SYS_FREQ    80000000        // frequency we're running at

int main(void)
{
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    // Configure the device for maximum performance but do not change the PBDIV
    // Given the options, this function will change the flash wait states, RAM
    // wait state and enable prefetch cache but will not change the PBDIV.
    // The PBDIV value is already set via the pragma FPBDIV option above..
    SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);
//setup L1 and L2
    mPORTBClearBits(  BIT_12 | BIT_11 );
    mPORTBSetPinsDigitalOut( BIT_12 | BIT_11 );
//setup S1 RB15/CN12
    mPORTBSetPinsDigitalIn( BIT_15 );
//setup S2 RD5/CN14
mPORTDSetPinsDigitalIn( BIT_5 );
    /* enable pullups on change notice pins 12,14 */
ConfigCNPullups(CN14_PULLUP_ENABLE | CN12_PULLUP_ENABLE);
    while(1)
    {   if(PORTDbits.RD5 == 0)
        mPORTBSetBits( BIT_11)  ;
        else
        mPORTBClearBits( BIT_11);

            if(PORTBbits.RB15 == 0)
        mPORTBSetBits( BIT_12)  ;
        else
        mPORTBClearBits( BIT_12);
    }
}
```

PIC32 and Peripheral Library

Set Fuse Configurations

Library function to configure system for max performance at 80 MHZ

Configure Digital I/O connected to LEDs and connect to switches Set weak pull up resistors to switches

Check Switch1 and turn on LED1

Loop

Check Switch2 and turn on LED2

Figure 9 Pushbutton Code

Demo 3 "Hello World" using an RTOS

RTOS (Real Time Operating System) for microcontrollers are similar to Operating Systems like Windows and Linux for personal computer and workstations, are now making an impact in high performance microcontrollers like the PIC32. In fact, with the computational power and memory of the PIC32, an RTOS is needed to efficiently process lots of simultaneous complex multi-task software operations at the same time. The best way to visualize this is to appreciate the types of applications that these advanced microcontrollers are handling today. In this demo we will introduce you to a free Open Source Real Time Operating system (RTOS) for the PIC32 developed by Richard Barry (see http://www.freertos.org/).

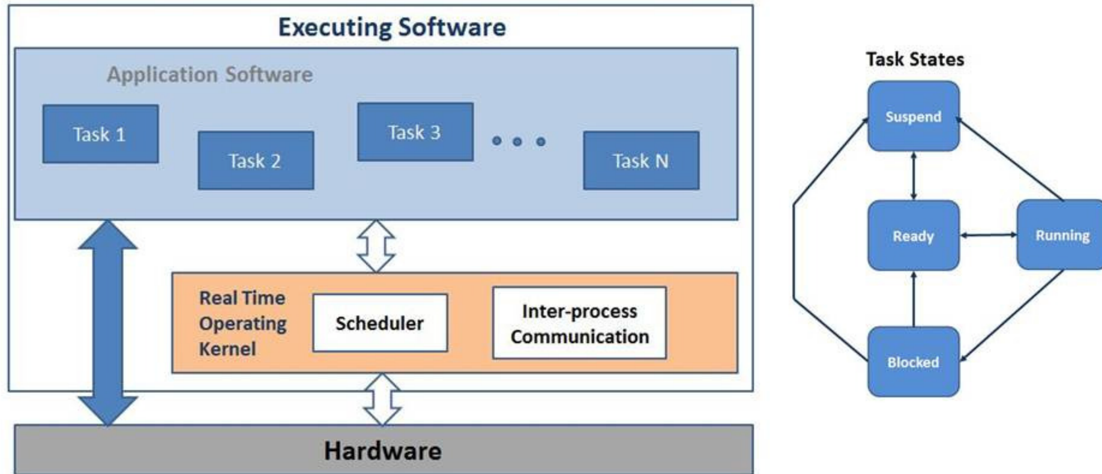
FreeRTOS allows PIC32 applications to be organized as a collection of independent tasks operating under control of a real time scheduler (see figure 9). The scheduler decides which task should be executing by examining the priority assigned to each task by the application designer. Tasks can move from running to block, to ready or suspended, depending on priority, and how you design the task. In this demo we will introduce you to task creation, scheduling, removal, and reinstatement as well as interrupt handling. We will use FreeRTOS to perform a "Hello World" function. It is a modest start but highlights the powerful capabilities of organizing your code around a RTOS. To make things interesting we will create in this demo the ability to simultaneously handle

independent LED blinking tasks –each task assigned to blink an individual LED. This demo requires use of the LED matrix and the hook-up diagram mentioned earlier. In this demo as you push and release SW1 an individual LED blinking task will be dynamically eliminated until the max of 8 tasks is reached. At this point as you continue to depress SW1, tasks that were previously eliminated are restored in reverse order. The main code is shown. The functions used are described in the on line documentation associated with FreeRTOS. Here's a quick overview:

- **vSetupEnvironment()**- encapsulates the code to setup the hardware (in this case 8 LEDs and Switch)
- **xTaskCreate(vTask1,"Task 1",240,(void*)image,1,&xTask1Handle)** – creates a "task 1" and passes it an image (digital port value of row/column for specific Led in led matrix). This task can be referenced later by its handle. The task is written as a led blink function.
- **vSemaphoreCreateBinary (xBinarySemaphore)**- creates a semaphore or binary flag/control that will be used uniquely by the switch interrupt service to invoke its task handler.
- **xTaskCreate(vHandlerTask, "Handler", 240, NULL, 3, NULL)** –creates a task to handle the switch once an interrupt occurs. The interrupt invokes this task using the above semaphore and this handler adds and removes tasks by their handles.
- **vTaskStartScheduler() –** start RTOS scheduler so that the tasks created so far start executing.

It is interesting to watch the LED activity as the FreeRTOS scheduler spreads and de-spreads the PIC32 processing power across the blinking tasks. This demo is included in the folder named RTOS example. Open this folder and navigate to examples and then open RTOSEXAMPLE.MCP. The demo is also available for download from the Nuts and Volts Web site.

RTOS Structure



- *"Hello World" Demo (8 tasks total)*
  - Task ---------------Read Switch and change a task priority
  - 7 Tasks--- Blink Lights

Figure 10 RTOS Structure Overview

```
int main( void )
{
    /* Configure both the hardware and the debug interface. */
    vSetupEnvironment();
    int image = LED11; //image is col row setting for col 1 row 1 LED
    xTaskCreate( vTask1,"Task 1",240,(void*)image,1,&xTask1Handle );
    image = LED12;
    xTaskCreate( vTask2, "Task 2", 240,(void*)image, 1, &xTask2Handle );
    image = LED13;
    xTaskCreate( vTask3, "Task 3", 240, (void*)image, 1, &xTask3Handle );
    image = LED14;
    xTaskCreate( vTask4, "Task 4", 240, (void*)image, 1, &xTask4Handle );
    image = LED15;
    xTaskCreate( vTask5, "Task 5", 240, (void*)image, 1, &xTask5Handle );
    image = LED16;
    xTaskCreate( vTask7, "Task 7", 240, (void*)image, 1, &xTask7Handle );
    image = LED17;
    xTaskCreate( vTask8, "Task 8", 240, (void*)image, 1, &xTask8Handle );
    /* Before a semaphore is used it must be explicitly created.  In this example
    a binary semaphore is created. */
    vSemaphoreCreateBinary( xBinarySemaphore );
    /* Check the semaphore was created successfully. */
    if( xBinarySemaphore != NULL )
    {   /* Create the 'handler' task.  This is the task that will be synchronized
        with the interrupt.  The handler task is created with a high priority to
        ensure it runs immediately after the interrupt exits.  In this case a
        priority of 3 is chosen. */
        xTaskCreate( vHandlerTask, "Handler", 240, NULL, 3, NULL );
        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
        /* If all is well we will never reach here as the scheduler will now be
    running the tasks.  If we do reach here then it is likely that there was
    insufficient heap memory available for a resource to be created. */
    for( ;; );
    return 0;
}
```

Initialize I/O

Create tasks – each task has its own LED

Create a semaphore for use between button interrupt and button task handler

Create button handler task

Start real time OS

Default loop

Figure 11 RTOS Hello World Code

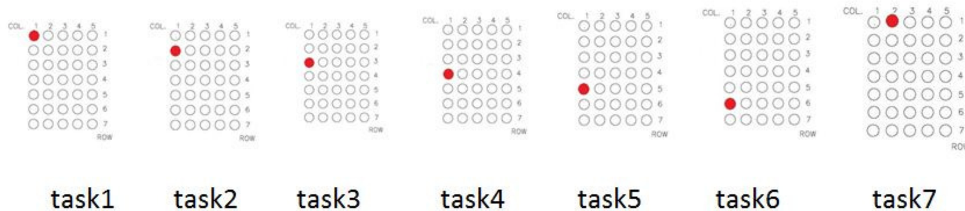


Figure 12 RTOS Hello World (7 tasks shown)

Demo 4 "Hello World" using PIC32 DMA

We now move to demonstrate a different "Hello World" using the DMA Controller to refresh the LED Matrix. Here we are servicing all 35 LEDs where each LED is individually controlled using the DMA (Direct Memory Access Controller) in conjunction with digital I/O and timer. The I/O port is configured to drive all row and columns of the LED matrix for 35 individual LEDs. These individual LED patterns are stored as a 1500 word "C" array (LED_PATTERN) in flash. DMA allows us to bypass the CPU and transfer directly to the I/O port. The DMA transfers occur as part of a timer 23 (32 bit timer) peripheral interrupt (which occurs every 10 milliseconds). DMA updates the LEDs with the specific array pattern synchronized to this interrupt. This demo is a

Microchip project named DMA.MCP and is available for download from the Nuts and Volts. The demo creates a dynamic light show with individual control of 35 LEDS to display random patterns between a distinct “d”, “M”, “A” display on the LED matrix (figure 13). The main code is shown and draws heavily on DMA peripheral library. Note the final loop -all of DMA activities occur while the CPU is simply executing the original blinky “Hello World” demo, truly a powerful demonstration of the PIC32 DMA.

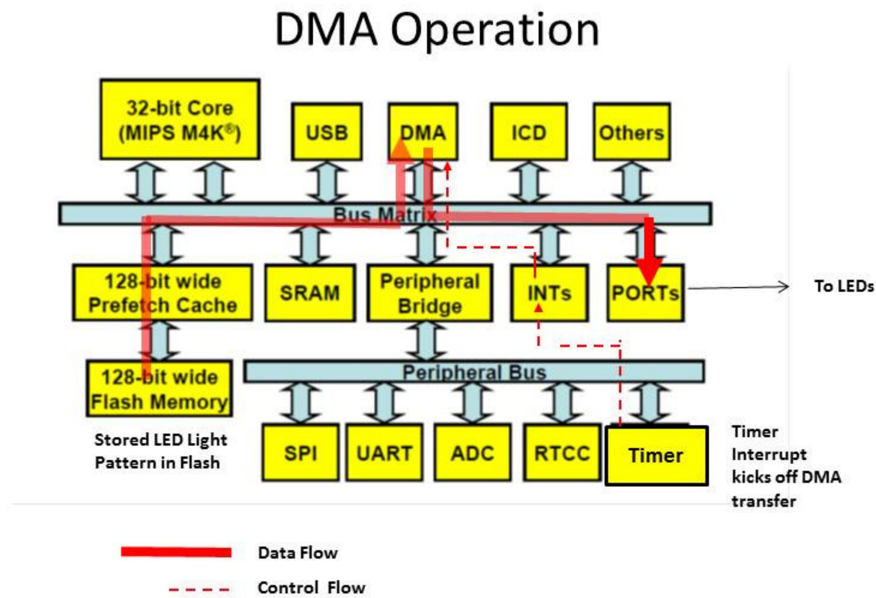


Figure 13 DMA Demo

```c
int main(void)
{
    int dmaChn=0;        // the DMA channel to use
    //set up outptus for on-board blinking LEDs
    mPORTBClearBits(BIT_12 | BIT_11 );
    mPORTBSetBits(BIT_11 );
    mPORTBSetPinsDigitalOut( BIT_12 | BIT_11 );

    //set the digital outputs for the matrix
    mPORTDSetBits(BIT_11 | BIT_10 | BIT_9 | BIT_8 |BIT_7  ); // turn off columns
    mPORTDClearBits( BIT_6 |BIT_5 | BIT_4 | BIT_3 | BIT_2 |BIT_1 | BIT_0 ); //clear rows
    mPORTDSetPinsDigitalOut(BIT_11 | BIT_10 | BIT_9 | BIT_8 |BIT_7 | BIT_6 |BIT_5 | BIT_4 | BIT_3 | BIT_2 |BIT_1 | BIT_0 );

    // Open the desired DMA channel.
    // We enable the AUTO option, we'll keep repeating the sam transfer over and over.
    DmaChnOpen(dmaChn, 0, DMA_OPEN_AUTO);

    // set the transfer parameters: source & destination address, source & destination size, number of bytes per event
    DmaChnSetTxfer(dmaChn, LED_pattern, (void*)&LATD, sizeof(LED_pattern), 2, 2); //desination =2 and tranfer =2
    // set the transfer event control: what event is to start the DMA transfer
    DmaChnSetEventControl(dmaChn, DMA_EV_START_IRQ(_TIMER_3_IRQ));
    // once we configured the DMA channel we can enable it
    // now it's ready and waiting for an event to occur...
    DmaChnEnable(dmaChn);
    // now use the 32 bit timer to generate an interrupt at the desired LED_BLINK_RATE
    {
        int pbFreq=SYS_FREQ/(1<<mOSCGetPBDIV());    // get the PB frequency the timer is running at
        // use 1:1 prescaler for max resolution, the PB clock
        OpenTimer23(T2_ON | T2_SOURCE_INT | T2_PS_1_1, (pbFreq/1000/4)*LED_BLINK_RATE);
    }
    while(1)
    {
        // do some other useful work
            mPORTBToggleBits(BIT_12 | BIT_11  );
            delay();
    }

}
```

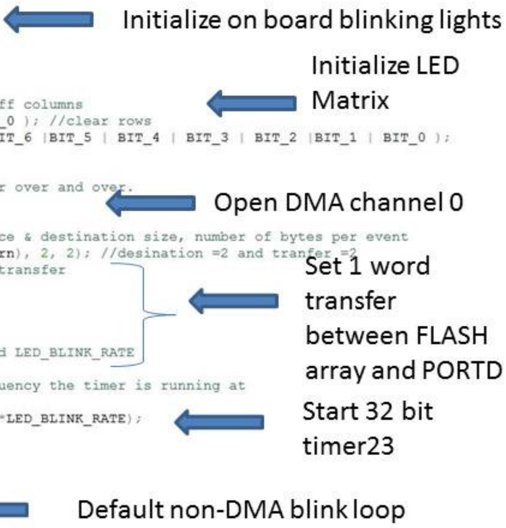


Figure 14 DMA Hello World Code

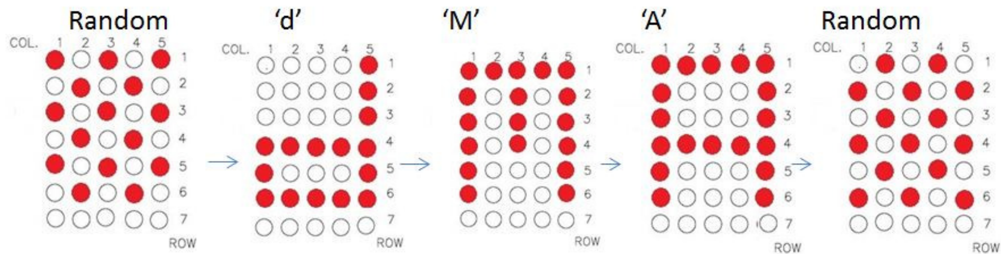


Figure 15 DMA Hello World Demo

What's next?

The Experimenter represents a new, affordable, and exciting way of getting involved in 32 Bit Microcontrollers. The PIC32 is currently recognized as one of the best in class, so what is a better way to get started than 32 bit microcontrollers? It leverages off of your 16 bit Experimenter experience, and a lot of what was learned and covered during the 16 bit series there is still pertinent here.

There is still a lot of ground to cover to help realize efficiency with this new technology. Given the expandability of the Experimenter there are many ways to realize this potential.

In upcoming articles we hope to at least pave the way with directions to USB, Ethernet, speech and high resolution graphics to name a few.

Figure 16 Experimenter Schematic 1 of 2

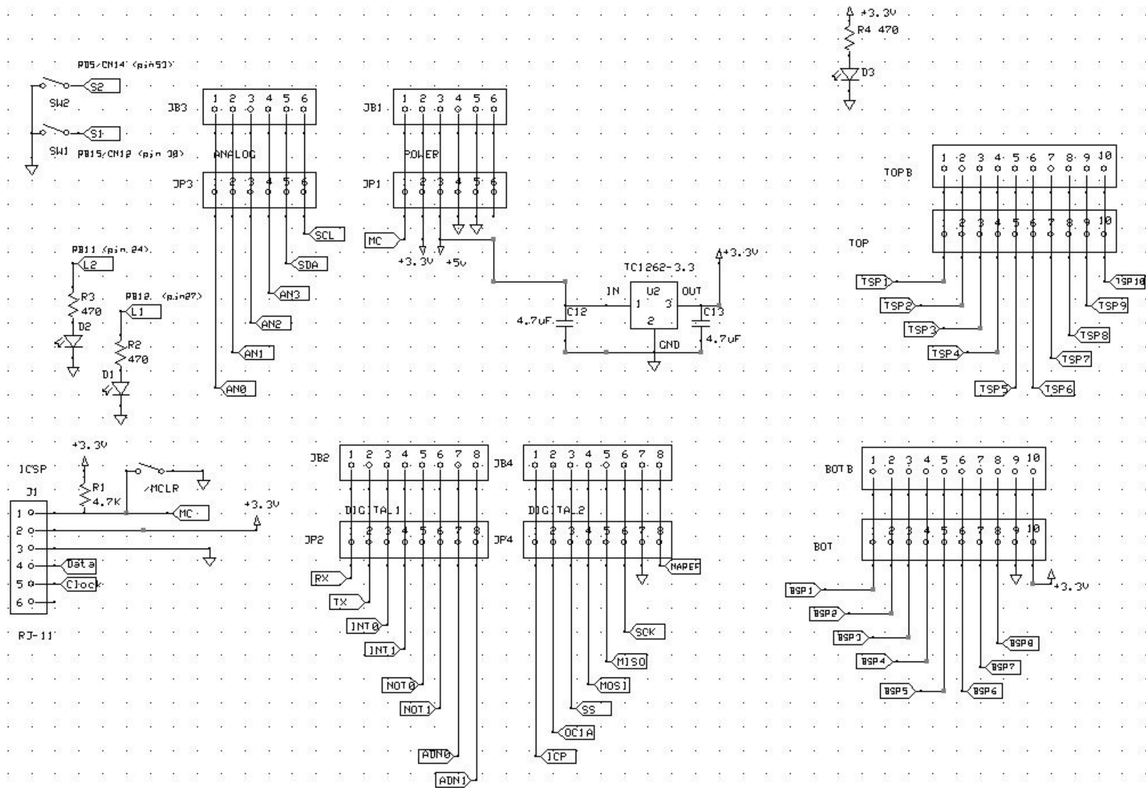


Figure 17 Experimenter Schematic 2 of 2