PIC n' Mix

Mike O'Keeffe

Our periodic column for PIC programming enlightenment

Four-digit, seven-segment LED display – Part 4

THE all-on-one-pin interface solution we explored last month allowed us to add a (calculator) keypad to our original display design, which was first used for a 24-hour clock. Now we're ready to program the hardware to turn it into a calculator.

While the additional modifications to the hardware were minimal, the software will not be as trivial. As we add functionality to the code, it's good to take a step back and consider the current behaviour of the code before changes are made.

The code for the 24-hour clock continually rotates around the four digits. In the background there is a 500ms timer used to time seconds and minutes. The code then updates the display every minute to show the current time. This works well for the clock. However, we must ask ourselves a question – will it still work when we change the fundamental operation from a clock to a calculator?

Consider how the calculator should work. Initially, it will display a zero, indicating it is awaiting an input. Pressing any key will commence the equation. The equations for this calculator will be of the type `X ø Y = Z`, where `X` is the first number to be entered, `Y` is the second, ø is the mathematical operator and `Z` is the answer. (For our simple calculator, the ø can represent addition, subtraction, multiplication or division.) So, we need to capture the first number (`X`), the mathematical operator (ø), the second number (`Y`) and the equals sign (=). Once these have been captured, we have enough data to perform the basic equation.

The first number (`X`) in the equation will be complete once an operator key has been pressed, then we can start entering the second number (`Y`) in the equation. Finally, the equals key (=) will be pressed for the answer (`Z`) to be displayed on screen. In the 24-hour clock code, the display is updated every minute. This is far too long to wait for a simple answer, or to update the display with the pressed digit. Therefore, we must update the display much quicker.

Another issue is that we must continually check for a button press using the ADC input on RA0. In the original 24-hour clock code, we manually rotated around the digits, any delay in this would cause one digit to shine brighter than the other or even worse, cause flicker.

Swapping the fundamental behaviour

The answer lies in swapping the behaviour between the 24-hour clock code and the calculator. In the 24-hour clock, the timer is used to count the seconds. The main code then rotates around the digits. The timer uses an interrupt service routine (ISR), which is meant to be very lightweight. In any ISR, the goal is to minimise the interruption to the main code by checking the interrupt flag, perform a very quick operation, reset the flag and return to the main code. If we added the digit rotation into the timer interrupt routine, the seconds would no longer be timed correctly and the clock would have been wrong. However, with the calculator, clock timing isn't important, while performing prompt calculations and ADC captures are. For the calculator, it is better to move our digit rotation into the timer ISR. Now the timer can be set to 1ms for a 1kHz switching frequency. Every time the ISR is called, the digit will swap. This automatic change gives us much more freedom to perform calculations and capture and display the numbers we enter.

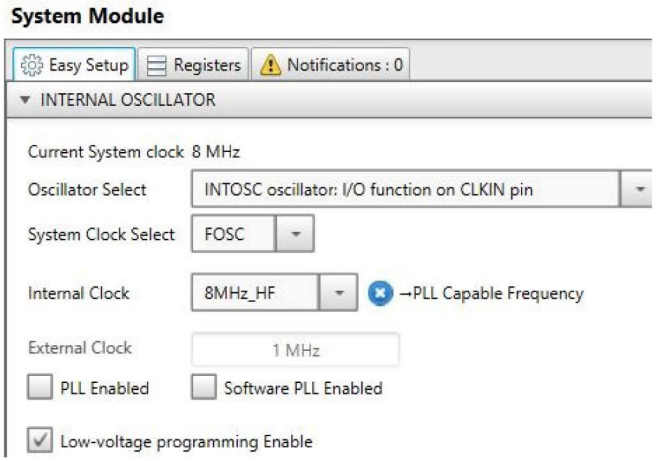

Fig.1. Changing the internal oscillator in MCC

Moving the digit rotation and display into the ISR adds significant delays inside the ISR. With the clock running at 500kHz and the ISR being called every 1ms, the ISR ends up taking 0.5ms to operate (this was measured during testing using the PICkit3 debugger). If the Timer is set to interrupt every 1ms, then our code only has 0.5ms operation time. The problem here is the main code is constantly interrupted, making it difficult to behave normally. This is a fine example of why ISRs need to be lightweight and fast. With this current clock speed, the calculator will not work properly and updating the display becomes near impossible. Increasing the clock speed to the internal clock at 8MHz improves this operating time to 0.2ms (See Fig.1). However, this is still not ideal and very slow for an ISR.

Entering numbers

It may seem trivial, but pressing the number keys on the keypad and expecting them to appear on the segment display correctly is a little tricky. Consider pressing the key '7' first, then '8' followed by '9'. We would expect the number 789 to appear on the display. In order to display this, we set up a four-digit array in the code and map these to the display's digits. In the array, we have four elements, numbered 0, 1, 2 and 3, where 0 is the first element in the array. In the four-digit seven-segment display, we can map this left-most digit to element 0 in the array and so on until element 3, which is the right-most digit. In our example, '7' would be entered into element 3 first. When we press key '8', the '7' on the display must be shifted into element 2 and the '8' will be stored in element 3. When key '9' is pressed, '7' and '8' will be shifted to the left and '9' will now be stored in element 3. In this example, element 0 does not have a number stored in it. There are two options here, we could display a zero or display nothing (blank LED). Storing a zero in element 0, will display a zero, so we need to choose another digit to represent nothing, which we'll cover further on.

We've now entered a number onto the display. The only problem is this number is split up into an array. We actually have 3 numbers, '7','8' and '9', but what we want for calculations is '789'. We need to convert '7','8' and '9' into a number that our microcontroller can mathematically process. A small function will provide this. The function

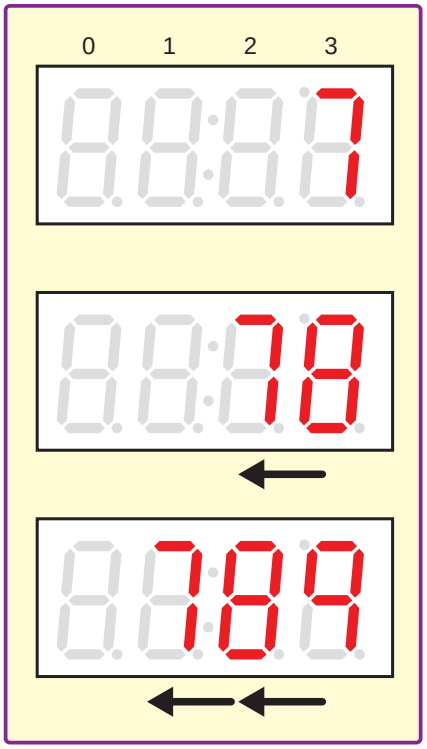

Fig.2. LED display digit shifting

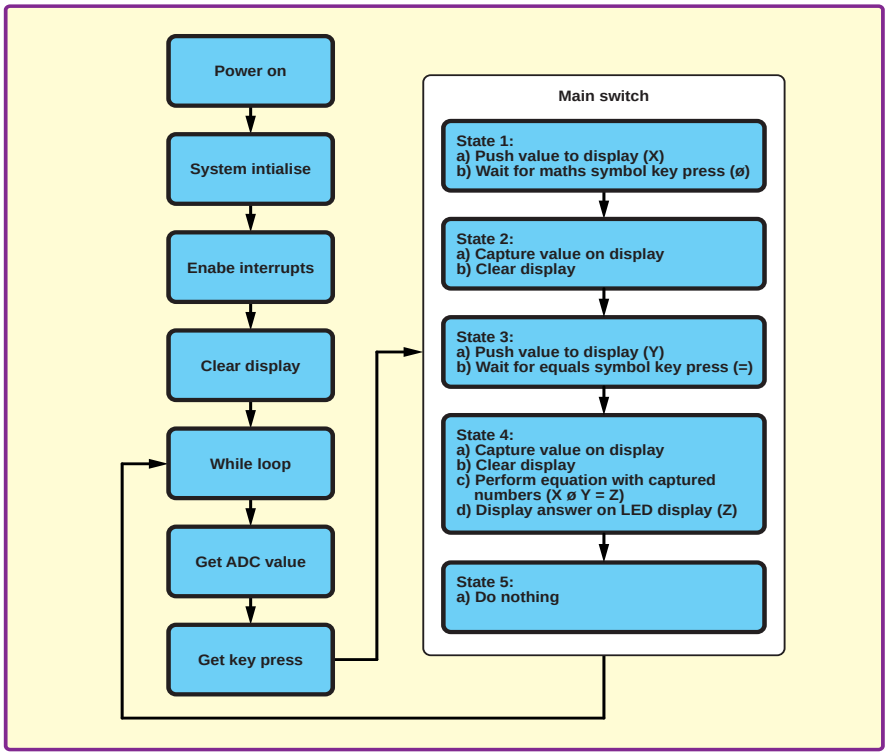

Fig.3. Calculator from chart

simply multiplies whatever number (0-9) is in element 0 by 1000. Element 1 will be multiplied by 100, element 2 will be multiplied by 10 and element 4 will be left alone. The results are added to give the result of 789. Now we have our number to work with.

Fig.2 shows the display as each digit is entered, shifting each previous number to the left.

Extra digits

In the 24-hour clock, we only needed the numbers 0-9 to be displayed on the display. This is still the case. However, the issue here is that we are using code from the original program, which is capturing a key press and then displaying it on the display. The normal digits will be mapped but the mathematical operator key presses now need to be added for the calculator to be able to work.

The original code uses a switch statement to swap between the various numbers to be displayed. The function `displayNumber()` is used to display the numbers on the display. In reality, it is used to convert key presses to exact pins connected to the segments to be controlled. To build upon that, we need to be able to capture when the multiply, divide, subtract, add, equals and clear buttons are pressed. We also want to be able to show a blank digit (ie, no LEDs lit) as it is easier to read a number on the display with no leading zeros. For improved calculator functionality it's a good idea to try and display an error message if a calculation goes wrong. This can be done by displayed the letters 'Err' as a shorthand for error. This will be called when the resultant number is larger than the four-digit maximum or when there's a divide-by-zero scenario.

```
0xA => Equals
0xB => CLR
0xC => Multiply
0xD => Divide
0xE => Subtract
0xF => Add
0x10 => E
0x11 => r
0x12 => Nothing
```

The above maps out the new cases to be added to the function `displayNumber()`, where `0xA` now represents equals. In the code, the case switch will perform some operation when this is seen, as with all the other values.

The code

There are significant changes to the code in order to convert it to a calculator. We need to move the digit display and rotate function into the Timer0 ISR. We will also need to completely change what's happening in the main code in order to control the behaviour of the project. We will be using a state machine (see Fig.3.). Space considerations mean it is not feasible to look at every change made to the code, but we will look at some of the key features.

```
void TMR0_ISR(void) {
    INTCONbits.TMR0IF = 0;
    TMR0 = timer0ReloadVal;

displayDigit(currentdigit, digits[currentdigit]);

    if(currentdigit >= 3) {
        currentdigit = 0;
    } else {
        currentdigit++;
    }

    if(TMR0_InterruptHandler) {
        TMR0_InterruptHandler();
    }
}
```

Instead of incrementing the seconds, the Timer0 ISR now calls the function `displayDigit()`. The variable `currentdigit` is incremented in the `if` statement that follows, making sure it circulates from 0 to 3 and restarts at 0 again.

```
clrDisplay();
calcstate = 1;
```

Starting off with the main code, we're going to first use the `clrDisplay()` function to turn off digits 0-2 and place a waiting 0 on digit 3. This will be our initial starting point. Since we will be using a state machine, we need to start it off by setting it to the first position using the `calcstate` variable.

```
while (1) {
    KeypadVal = ADC_GetConversion(0);
    disVal = getKeyPress(KeypadVal);
```

The `while` loop starts with an ADC capture using the MCC `ADC_GetConversion()` function. The value is

then stored in KeypadVal. The next function called is getKeyPress(), which takes the value in KeypadVal and tries to evaluate which button has been pressed. These two function calls are outside the main switch, meaning they will always be called. getKeyPress() will convert any key press to a specific value (all except one key, which is the CLR key). When the CLR key is pressed, a soft reset occurs using the assembly soft reset command asm ("reset");. A software (or soft) reset is one where the code jumps to instruction zero. (A hard reset, is where the power is cycled on and off. A soft reset will often start everything from fresh, but it may not always work, especially if poorly designed code writes over parts of memory that it shouldn't, thereby corrupting the memory space.)

```
switch(calcstate) {
case 1:
    if(mathSign > 0) {
        calcstate = 2;
        break;
    }
    if(KeypadVal < maxADC) {
pushToDisplay(disVal);
    }
    break;
```

Starting with Case 1 in the switch statement, we want to display the numbers captured and converted in the previous functions. The value to be displayed is stored in the variable disVal. First, we check to see if the value mathSign has been assigned a value. mathSign is initialised as zero in the functions.c file. When a mathematical operator key has been pressed, mathSign will be assigned a value based on the key pressed. At this point, the next state will be selected. Before that, we check the KeypadVal is less than maxADC. This verifies a valid number key has been pressed. maxADC represents the maximum ADC input for a valid key press. Then the value is pushed to the display using the pushToDisplay() function. This function will not be discussed here, it simply left shifts any current digits and stores the new value in the right-most digit. It will only allow four values, maximum. Any numbers pressed after that will be ignored.

```
case 2:
    getDisplayNumber();
    clrDisplay();
    __delay_ms(100);
    calcstate = 3;
    break;
```

In Case 2, we want to grab the number on the display. This is currently stored in a 4-byte array called digits[]. The getDisplayNumber() function takes the separate numbers in the digits[] array and converts them into a single number. For example, 7, 8 and 9 would be combined to get 789. The display is then cleared using clrDisplay() – again, a small wait and the state machine moves onto the next state.

```
case 3:
    if(disVal == 0xA) {
        calcstate = 4;
        break;
    }
    if(KeypadVal < maxADC) {

pushToDisplay(disVal);
    }
    break;
```

In Case 3, we want to capture the second number and display it on the LED display. This is similar to Case 1, except we're looking specifically for the equals key to be pressed (which is represented by 0xA as mentioned earlier).

```
case 4:
    getDisplayNumber();
    clrDisplay();
    mathAnswer = performMath();
    convertDisplayNum(mathAnswer);
    calcstate = 5;
    break;
```

Case 4, we want to capture the second number on the display. We clear the display using the clrDisplay() function again. Next up we have a function caller performMath(), which takes the first number entered, the captured math operator and the second number entered and evaluate the answer, which is then stored in the variable mathAnswer. This number must now be converted into a format that can be displayed on the LED display. convertDisplayNum() is the function that converts the number into the 4-byte array digits[]. To finish, we move onto the next state using the calcstate variable again.

```
case 5:
    // Do nothing here
    break;
```

This is an important state in the state machine. Here we enter a state, from which we will not easily exit. At this point, the result will be displayed on the LED display. The only way to get out of this is to press the CLR key, which will reset the PIC and the calculator.

```
default:
    calcstate = 1;
    break;
}
}
```

It's not always necessary to add the default case in a switch statement, but it is good practice. If for some weird reason the variable calcstate contains a value other than 1-5, then the default case will reset this variable back to 1, resetting the process again.

There's a few interesting points to see in the software. One of the key points from above is keeping the ISR as lightweight and as fast as possible. Although adding more functionality to the ISR can be good providing the risks and delays are understood. It's also interesting to note the changes to the hardware were minimal, while the software changes were rather extensive. There's very little code that could be re-used.

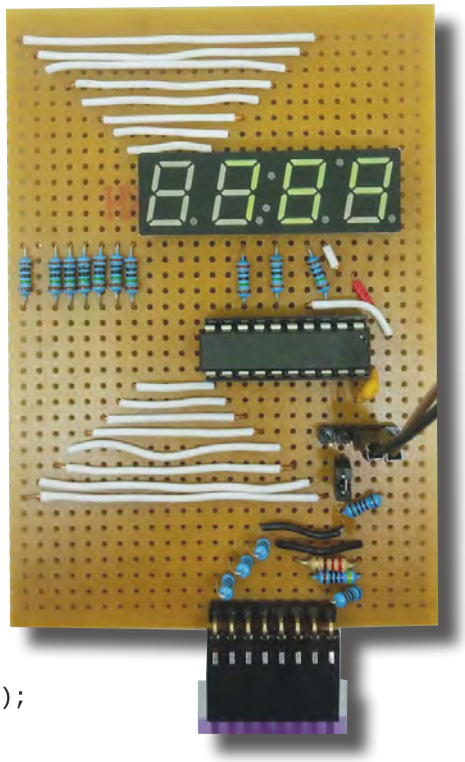

Fig.4. Fully working calculator showing 789 as the result

Last but not least, the display is noticeably brighter. Having the display and rotation in the ISR ensures that each digit is given equal power-on time.

Next month

I'm taking a small sabbatical for my greatest adventure yet – the birth of my newborn twins: Chris and Ethan. While I'm away, the original and highly esteemed *PIC 'n Mix* columnist Mike Hibbett will be making a short return to fill in for me. He has some exciting projects in store for you. All I'll say is that it has something to do with 'FFT'. I look forward to seeing him back in action and I will see you all again upon my return.

Not all of Mike's technology tinkering and discussions make it to print.

You can follow the rest of it on Twitter at *@MikePOKeeffe*,

on the *EPE Chat Zone* or *EEWeb's* forums as *'mikepokeeffe'*

and from his blog at *mikepokeeffe.blogspot.com*