At Long Last, Longs

www.melabs.com

As many of you know, past versions of the PICBASIC PRO™ Compiler have included 3 different types of variables, bits, bytes and words. These variable types are unsigned, meaning you can not do things like test for negative numbers in an If statement. While there are certainly many ways to work with these variable sizes to use larger and even negative numbers, it is easier to have a larger, signed variable type to begin with.

So with this in mind, the latest versions of PICBASIC PRO™ add a new signed, long variable type. To accomplish this, we actually created a new executable version of the compiler, PBPL.EXE, along with the accompanying new library and header files. PBP.EXE and PBPW.EXE still only work with bits, bytes and words. To use the new long type, you would use PBPL.EXE as the executable. You can select this in the IDE you are using. See the melabs web site for more information on how to do this for the different IDEs.

These new long variables are only available for the PIC18 parts. The larger instruction set of the PIC18 parts allow long variables to be handled more efficiently than in the 12-bit or 14-bit core parts. There is also usually more RAM available on the PIC18 devices. Other than for use with the 12-bit and 14-bit core parts, why might you want to use the non-long versions of the compiler for the PIC18 parts? We'll get to that soon enough.

Most of the examples in this article show numbers in the decimal format (base 10), the way we normally see them in life. More information on different numerical representations can be found in the section of our web site on number systems: http://melabs.com/resources/articles/index.htm.

Before we go into the details of this new signed, long variable type, let's get little more perspective by taking a look at the previously available types.

Bits

Bit variables are the smallest of the variable types. They are only a single bit in memory. An I/O pin is an example of a single bit. One or more bits may be used as flags in a program. A single bit can represent only 2 states or values. These values can be interpreted as high or low, true or false, on or off, or in numeric form, 1 or 0. So if you count using a bit variable starting at 0, the next number would be 1. If you add one more, the new value would be 0 again. There is no other choice. There is also no way to interpret a sign from a single bit. It's value is always unsigned.

Bytes

Eight bits are grouped together to form a byte of memory. While these 8 bits can still be accessed individually, they can also all be treated as one unit by using the whole byte. The byte

is the standard memory unit of an 8-bit microcontroller. A byte can represent a range of values from 0 to 255. If you try to go beyond the upper limit of 255 by adding 1 or incrementing the variable, it will roll back around to 0. Likewise, if you subtract 1 from a byte that contains 0, it will scoot back around to 255. Bytes, in PICBASIC PRO, are unsigned. However, there are ways to treat bytes as if they are signed. More on that in a moment.

Words

As just mentioned, bytes are the native size for an 8-bit microcontroller. However, other variable sizes can be created using multiple bytes together as a unit. Word variables are created by the compiler by combining 2 bytes together. These bytes are sequential in the variable memory space and allow numbers larger than 255 to be represented. When a word variable is used in a PICBASIC PRO program, the compiler automatically does what is necessary to update both bytes, treating them as a single unit from the users perspective. A word variable's value can range from 0 to 65535. The 2 bytes that make up a word variable can still be accessed individually. Each of the 16 bits that are in the word variable can be accessed individually, as well. As in bytes above, word variables are also usually treated as unsigned. More on that shortly.

And Finally, Longs

It's always nice to have more. So with the latest versions of the PICBASIC PRO Compiler there is now a signed, long variable type. Each long variable is made up of 4 sequential bytes in memory - that's 32 bits. The numeric values that can be represented in a long variable range from -2147483648 to 2147483647, or in some cases, from 0 to 4294967295. What can we do with these giant numbers? Well lots of handy things. But before we get into some of them, let's get to signs.

Signs

As stated above, the compiler usually treats bits, bytes and words as unsigned, and longs as signed. When we looked at bits, we saw that there really wasn't any way to have a sign for a bit in any case. It is either 0 or 1 and that's it.

But for bytes and words, it is possible to treat them as if they were signed variables in some cases. One important factor to note when considering signs is that the actual sign bit itself is represented by the top bit of the variable. For a byte variable, this would be bit 7. When that bit is high (1), the value can be considered negative. So when treating a variable as signed, its largest value is now only half of what it would be when treated as an unsigned variable. In the case of an 8-bit, byte variable, since the sign is taking up the top bit, there are only 7 bits left to represent the rest of the data (-128 to 127).

So how do we treat an unsigned byte or word variable as signed? In some cases, there are special modifiers used by some of the PICBASIC PRO instructions to do just this. For example, lets look at a byte variable we will call B, for lack of better inspiration. We can create it like this:

```
B       Var     Byte
```

Now we can set its value to anything we like, within its range:

```
B = 0
B = 100
B = 255
```

Let's say we want to display B's value on an LCD. We can write:

```
Lcdout Dec B
```

Dec is a modifier that tells the compiler to output the value of B as ASCII decimal characters that we can read on the display. So if B contained the value 255, we would see a 2, a 5 and another 5 on the display.

There is another modifier we can use to pretend B is a signed number:

```
Lcdout Sdec B
```

Sdec basically does the same thing as Dec in that it displays ASCII decimal characters. But it takes one additional step: it looks at the top bit of B and if it is high, it displays B as a negative number, including the minus sign. So if B still contained 255, what we would see on the display would be a minus sign (-) followed by a 1. Wait a minute, a -1? How did we get there?

Well how much detail do we really want to get into? How many of you are still awake? The compiler uses 2s-complement math for addition and subtraction. Let's say you want to subtract 10 from 20:

```
B = 20
B = B - 10
```

You would get 10, of course. But what if you subtracted 20 from 10 instead:

```
B = 10
B = B - 20
```

Now what would you get? You wouldn't get an error because in most cases there is no way to show an error on a microcontroller. It does 2s-complement math so in our unsigned byte variable we end up with 246. But if we display it on our LCD using the Sdec modifier we would see -10, just like magic.

To get more information on 2s-complement math it is probably best to just punch that search term into your favorite search engine. We have lots more to cover here.

We have seen how 2s-complement subtraction can allow us to create, more or less, a negative value. It works with addition the same way. However, we cannot use these negative values in multiplication or division with our unsigned byte and word variables. We cannot even say something as simple as:

```
If B < 0 Then iamnegative
```

This equation will never be true because the compiler knows that bytes and words (and bits, too) are unsigned and can never be less than 0.

There are workarounds for this such as:

```
If B.7 = 1 Then iamnegative
```

This tests the top bit of the byte, which is by convention the sign, and if it is high, it knows the value is negative.

But it sure might be handy to have a real signed variable sometimes. So the new long variable that has been introduced was made signed instead of unsigned. This allows us to easily say:

```
L       Var     Long
L =  -10
If L < 0 Then iamnegative
```

And it will really work! Yay! Even multiplies and divides will work as signed when used with long variables.

There are a few cases where long variables will be treated as unsigned. One is with our Dec modifier above. It will only work as expected when the value of the long is positive. To get the full range of plus and minus values, use Sdec. Also, Pause and other PICBASIC PRO commands will treat longs as unsigned. This allows Pause to give a really, really long delay. The PICBASIC PRO manual gives the ranges of values used for the various commands.

No Free Lunch

There are a couple of drawbacks to longs. They take up more memory and they take longer to operate. It is obvious that they take up more memory (4 bytes instead of 1 or 2). But what you may not realize is that there are temporary, intermediate variables that sometimes need to be used by the compiler. If you write an expression, for example, temporary variables may need to be created to store intermediate results in this expression while the compiler is working through it.

```
L = L + (W * 2)
```

In the equation above, the compiler will need to create a temporary variable to store the intermediate result of W * 2 before it can be added to L.

Temporary variables need to be the largest size available so that there is enough room to store any potential result. For the non-long version of the compiler (PBPW.EXE), temporaries are word-sized. For the long version of the compiler (PBPL.EXE), temporaries need to be long sized.

There is also the speed issue. It takes a fair bit longer to do a long multiply (or any math or other operation for that matter) than it does to do a word multiply. There are just more bytes that need to be handled.

We have done some things to address some of the speed and size issues. One of the things we did was to differentiate the operation of the # modifier from the Dec and Sdec modifiers in the Debug, Hserout, Hserout2, Lcdout and Serout instuctions. If you use the # modifier for ASCII decimal output in any of these instructions, it will only work with a maximum of 16 bits of data. This means it will work faster and generate less code:

```
W      Var     Word
Lcdout #W
```

Using the Dec or Sdec modifiers with any of the above instructions (with the exception of Serout which doesn't support modifiers other than #) will allow the display of the full 32 bits of data:

```
L      Var     Long
Lcdout Sdec L
```

Serout2, however, treats the # and Dec modifiers the same, working with the full 32-bits of data. No speed gains to be had there, I am afraid.

So the bottom line is if you don't have to have longs or signed variables, it can be quicker and more compact to stick with the word version of the PICBASIC PRO Compiler. But if you want to get bigger and go below 0, longs are here.

Now that we have a handle on the sizes and the signs, lets take a quick look at how to create each type and what some of the variable modifiers do, before we move on to the really fun stuff.

Creating Variables

Creating each of the variable types is pretty easy:

```
T      Var     Bit
B      Var     Byte
W      Var     Word
L      Var     Long
```

You should, of course, use variable names that are more telling of what that variable does in a program. Some programmers even like to encode the size as part of the variable name:

```
wBatteryLevel       Var     Word
```

The small w in the label above would indicate to the programmer that this is a word-sized variable. It does not tell the compiler anything. It is only of use to someone reading the program.

While we're here, I can point out that you can also tell the compiler where to put a variable in the memory map. It normally works this out for itself, but you can make a suggestion if you have a preference:

```
B       Var     Byte $70        ' Put byte variable B at location hexadecimal 70 in RAM
W       Var     Word Bank1      ' Put word variable W somewhere in RAM bank 1
```

The compiler will try to comply with the requests and if it cannot, it will issue a warning.

Arrays

Arrays are simply a series of memory locations that can be accessed sequentially using a single index. A Var statement at the beginning of the program allocates the number of locations specified between the brackets in RAM. The programmer is then responsible for making sure not to try to read or write data to locations outside of the allocated space. The compiler will not check to make sure any data is within the boundaries of the array. It will happily read and write past the end of an array, if requested.

The Var statement below will create an array of 10 long elements, indexed from 0 to 9. This will use 40 bytes in RAM since each long location uses 4 bytes.

```
lArray          Var     Long[10]
```

Aliases

Aliases are a way to give a variable or register, or part of a variable or register another name.

Let's say that you have a long variable and for some reason you need to be able to get to the third byte of it (bytes are counted starting with 0: Byte0, Byte1, Byte2, Byte3). There are several ways to do this. One of the ways is to create an alias to it:

```
L       Var     Long
bL3     Var     L.Byte2
```

The name bL3 can now be used to look at or set the third byte of the long variable L without altering any of Ls other 3 bytes. Keep in mind we could have named it anything, not necessarily bL3. We can set that byte to 0 as follows:

```
bL3 = 0
```

Of course, this same thing could be accomplished without actually creating an alias:

```
L.Byte2 = 0
```

We can also create aliases to words and bits:

```
wLH  Var    L.Word1
tLSign Var  L.Bit31
```

The bit example would allow direct access to the sign bit of L.

You can even create an alias to a single element of an array:

```
lArray      Var   Long[10]
lElement2   Var   lArray[2]
```

The section on aliases in the PBP manual has a table that shows the different modifiers that can be used to create aliases or otherwise used in a program.

What Now?

Now that we have signed longs, what do we do with them? First there is the signed part. Going negative can be really useful. As mentioned earlier, signed numbers can be more easily tested for less than 0. Two signed numbers can actually be compared to each other and yield the expected result. Signed multiplies and divides can be executed on longs. Keep in mind that while unsigned numbers can be treated as signed using 2s complement, as discussed above, this only works with addition, subtraction and a few operators like Sdec. Two unsigned numbers cannot be compared to each other as if they were signed.

But you do have to pay attention when you are moving negative numbers around. You really want to make sure you stay within the long variable type to preserve the sign of a negative number. Here's a short (bad) example:

```
B     Var   Byte
L     Var   Long

L = -1
B = L
L = B
```

That is example is pretty short and probably pretty useless as well. But it does demonstrate a trap you can easily fall into while moving a number from one variable to another. In this short program, L starts off as -1. What does it end up as? Well, it ends up as 255. Since B, as a byte, is unsigned, L will end up as a positive number. As we saw far above, 255 is the 2s complement form of -1. Any transfer from an unsigned bit, byte or word into a long will end up by definition

as a positive number. If you want to maintain the sign, stick with moving longs to longs.

Another good thing about signed longs is they are long, i.e. they can hold really large numbers. While you can do a lot with words, if you are averaging many of them or trying to talk to external devices like 24-bit A/D converters, words don't make it easy. While these things can be done, and have been done, with words, having a number that can be larger than 2 billion can have its place. Longs are as easy to use as any other variable type and may be used any place bytes or words are used. Just remember, longs do take up more storage space and do take longer to operate on. So don't replace all your variables with longs just to do it. Use the appropriate variable size for the job.

But I Just Have to Have Floating Point

Really? Because that can really slow down and bloat up a program. If you just have to have it, then there are ways to do it. We cover one on our web site here: http://melabs.com/resources/fp.htm.

But if you just want to be able to handle numbers with say 1, 2 or 3 digits after a decimal point, it is pretty easy using longs and possibly even words. It's all a matter of how you look at numbers.

Let's say we're working on a counter for a piggy bank. We would like to be able to keep track of the dollars and cents in the piggy bank as we add money to it. Now we could use one variable for the dollars and another for cents, but instead, let's just use one variable that includes them both. So now we're back to needing floating point. Not really. We just need to modify our thinking about numbers a bit.

What if we just use one variable to track all our riches in cents? We don't need floating point for that. What we actually want is fixed point. If we want to think about it in terms of dollars, we just assume that we have a decimal point that is always 2 positions to the left in our variable. So if we have 1017 cents, we can also think of it as 10.17 dollars. You can look at other numbers, volts, for example, the same way: 5.046 volts can be looked at, and calculated as, 5046 millivolts. To make this work, we just need to be a little tricky with how we accumulate and display our values.

Keeping track of how much money we add to our piggy bank is pretty easy. We just add in any new amounts in cents. If we add 1 dollar, we just add it in as 100 cents (or dollars * 100). When we want to see the amount, here's what the dollars and cents might look like on an LCD:

$10.17

I have 10 dollars and 17 cents. I could go to a movie, if I didn't want popcorn.

To display this dollars and cents format from our single cents variable we could say:

```
Cents           Var     Long
```

```
Cents = 1017
Lcdout "$", Dec (Cents / 100), ".", Dec2 (Cents // 100)
```

This sure looks like we displayed a floating point number, but the code was pretty quick and compact. Let's break down the Lcdout statement to see what is happening. First we displayed a $ sign by putting it in quotes. Next we want to display the number of dollars. We know this is the number of cents divided by 100. We use Dec to turn the number into ASCII decimal numbers that will show on the LCD. Then we put out the period for the decimal point. Finally we took the remainder of the Cents to put out 2 characters to the right of the decimal point. We used a variation of Dec to do this: Dec2. The 2 tells Dec to always put out 2 numbers as this is what we expect to see after the decimal point.

Now we probably really didn't need longs in my piggy bank example above. We could have used words because I am not likely to accumulate more than $650 before I go on a spending spree. A word variable can hold up to 65535 cents, or $655.35, before rolling back to 0. But you might have saved up more money than this and need longs. How many cents can you save in a long? How about 2147483647 cents, or $21474836.47. I think you can order the popcorn.

There is a caveat here. This particular code will only work with positive values. But that is OK since my piggy bank doesn't work like a real bank - my balance can't go negative. But there are certainly times you do want to display a negative floating point look-alike number.

How about we average 3 numbers that each have 3 decimal places and can be negative as well?

```
L1      Var     Long
L2      Var     Long
L3      Var     Long
Average Var     Long

L1 = 123456         ' This is 123.456 in fixed point form
L2 = 234567         ' This is 234.567 in fixed point form
L3 = -456789        ' This is -456.789 in fixed point form

Average = (L1 + L2 + L3) / 3
Lcdout Sdec (Average / 1000), ".", Dec3 ((Abs Average) // 1000))
```

Getting the average is pretty easy. It is just the three numbers added together and the result divided by 3. There is nothing special we have to do at this point about whether the numbers are plus or minus as the compiler takes care of this with the signed long variables.

The Lcdout statement for displaying a signed fixed point value looks similar to the unsigned version above. There are really only a couple of differences. For the part of the number before the decimal point, we use Sdec instead of Dec so that when the number is negative, it will be interpreted properly, as well as display a minus sign. The part after the decimal point is a little trickier. We can't just use Sdec again because we can't have a minus sign potentially popping up

in the middle of a number. But we can't use Dec for a number that might be negative. So the solution is to use the Dec but make sure the number can never be negative by using the Abs function. Abs gives the absolute value of whatever it operates on. So if Average is negative, Abs of Average is flipped around to positive. If Average was already positive, Abs has no effect.

If a Frog Had Wings

We looked at If statements a little bit, above. But there are probably some more details that might be nice to look at. The main thing to remember is that the If statements in the long version of the compiler know that long variables are signed and word, byte and bit variables are unsigned, i.e. always positive. Constants are signed also. Some examples are probably in order.

```
B1    Var    Byte
B2    Var    Byte
W1    Var    Word
W2    Var    Word
L1    Var    Long
L2    Var    Long

B1 = 10
B2 = 20
If B1 < B2 Then Goto Smaller
```

This example is pretty straightforward. The compiler does the expected comparison. B1 and B2 always contain positive numbers. In fact, the comparisons with each variable type being the same do pretty much what is expected in a simple comparison:

```
W1 = 100
W2 = 1000
If W1 = W2 Then Goto Same

L1 = -20
L2 = 20
If L1 > L2 Then Goto Bigger
```

What we really want to look at is what happens when we start mixing variable types and even constants.

```
B2 = 20
W2 = 1000
If W2 > B2 Then Goto Loop
```

B2 and W2 are both unsigned variables so we never have to worry about them going negative. But there is a size difference. Since B2 can never be larger than 255, once W2 exceeds 255, the If statement shown will always be true. I know this is fairly obvious. Even more obvious:

```
B2 = 20
If B2 < 1000 Then Goto Forever
```

Once again, B2 can never be larger than 255 so this equation will always be true. We can easily see this. But what if the variable is named MyVar? If we have declared MyVar as a byte, and we really can't tell much from that name, we will run into the true forever case. If it is a word, it probably can make sense. Just something to think about.

Let's look at signed comparisons.

```
B1 = 10
If B1 = -10 Then Goto Never
```

The If above can never be true. Since the compiler knows that B1 is unsigned, therefore always positive, the result of comparing B1 to any negative number will always be false.

```
B2 = 20
L1 = -20
If B2 > L1 Then Goto Bigger
```

This example shows a comparison of an unsigned and a signed value. The compiler will handle this properly, that is, following its rules. It will always treat the unsigned value as positive and the signed value and, well, signed. So you can do this kind of thing. Just be sure to not expect B2 to go negative.

What Else Can We Do?

There are some specific commands that can benefit from a larger variable size. While all the commands can now accept longs as any of their parameters, longs can be helpful in commands such as Count, Pause, Pulsin, Pulsout, Rctime, Shiftin and Shiftout. In the Count command, for example, a long variable can store a count far larger than the previous limit of 65535 using a word variable. Shiftin and Shiftout can now move 32 bits with a single variable or constant. We already mentioned that Pauses can get really long, using longs. Of course, loop commands like For..Next, Repeat..Until and While..Wend can also use a long variable. Then there are the Lookdown2 and Lookup2 commands that can now search for and return very large numbers.

Be sure to look at each instruction in the PICBASIC PRO Compiler reference manual to see the specific effects and limits when using different variable sizes.

Time To Say So Long

Hopefully, we have given you some better insight into the use of signed long variables. They can help simplify a program when needed, but they also have their costs. So be sure to use them where applicable, and where they are not necessary, just continue to use the non-long version of the compiler.