PICAXE PRIMER

■ BY RON HACKETT

SHARPENING YOUR TOOLS OF CREATIVITY

INTRODUCING THE NEW PICAXE M2-CLASS MICROCONTROLLERS

Back in July when I was just beginning to think about a topic for this month's Primer, Revolution Education released three new M2-class processors: the 08M2, 14M2, and 20M2. That settled the issue for me; I immediately ordered a bunch of them, and impatiently waited for the package to arrive at my doorstep. If you prefer to not wait as long as I did, all the M2 processors are now available here in the USA from Peter Anderson (www.phanderson.com/picaxe/index.html). In case you have been unable to locate a source for the AXE401 Shield Base, I should mention that Professor Anderson also carries the kit version of the AXE401 on his site.

BRIEF OVERVIEW OF SELECTED FEATURES OF THE M2-CLASS PROCESSORS

We don't have near enough space in this installment of the Primer to discuss all the significant new features of the M2 processors, so I'm going to limit our discussion to just a few of the most significant improvements to the older M-class devices. Naturally, the three new M2 processors have much in common with their older sibling — the PICAXE-18M2 — but they also add a few new tricks of their own, including a built-in internal temperature sensor and two powerful new commands (*rfin* and *rfout*) that will greatly simplify working with inexpensive RF wireless transmitters and receivers. (I'm sure we'll be exploring that capability before long!)

Figure 1 presents a summary of what I consider to be the most significant new features of the M2 processors.

To begin with, all M2 devices are able to operate with a supply voltage as low as 1.8V which means that battery-powered projects now only require a two-cell alkaline battery pack. In addition, the range of the internal clock frequency has been greatly expanded; all M2 processors can operate at nine different frequencies, from 31 kHz to 32 MHz.

Since power consumption is directly related to clock frequency, the new lower frequencies can be used to significantly extend the life of battery-powered projects. Of course, the maximum operating frequency of 32 MHz is four times faster than the 8 MHz maximum of the older M-class processors, which greatly increases the range of possible M2 projects.

There are also several significant improvements in the memory capacity of the M2 processors. First, the program memory has been increased to 2,028 bytes which is eight times the capacity of the older M-class processors. Also, the 256-byte data (EEPROM) memory is now completely separate, so using it doesn't decrease the amount of available program memory. This means that all the M2 processors — including the tiny 08M2 — can run programs containing as many as 1,800 lines of BASIC code, which again greatly increases the range of possible M2 projects.

The number of general-purpose (GP) variables has been doubled from 14 to 28 (i.e., b0...b27) which also makes more complex projects possible. Finally, the amount of memory space available for "storage variables" has also been greatly

Feature	08M2	14M2	18M2	20M2
Voltage Range	1.8V - 5.5V	1.8V - 5.5V	1.8V - 5.5V	1.8V - 5.5V
Min. Internal Freq	31kHz	31kHz	31kHz	31kHz
Max. Internal Freq	32MHz	32MHz	32MHz	32MHz
Program Memory	2048 bytes	2048 bytes	2048 bytes	2048 bytes
General-Purpose Variables (b0Éb27)	28 bytes	28 bytes	28 bytes	28 bytes
Storage Variables	100 bytes	484 bytes	228 bytes	484 bytes
Total Variables	128 bytes	512 bytes	256 bytes	512 bytes
Pwmout channels	1	4	2	4

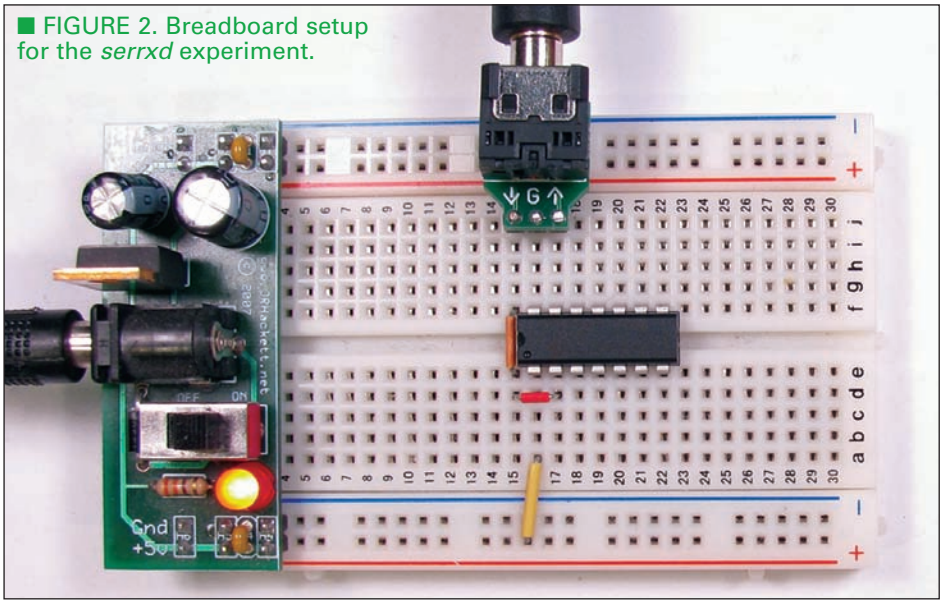
■ FIGURE 1. Selected features of the PICAXE M2 processors.

increased. (The M-Class processors supported 48 storage variables.)

This may not seem like such an important improvement at first. (How often have you needed more than 48 storage variables?) However, the M2 processors also now support the X2-class feature of "indirect addressing" of the storage variable memory. In case you're not familiar with indirect addressing, we're going to take a close look at it shortly because we can accomplish some really powerful tasks with it.

If you're interested in microcontroller-based robotics, you will definitely want to check out the 14M2 and 20M2 processors which both include a total of four PWMOUT channels; two of them are entirely independent, and the other two share the "servo" timer. As a result, either processor can control two independent DC motors and several servomotors at the same time (or three independent DC motors). That's an amazing amount of processing power for a small robot controller.

That's about all the space we have this month to introduce the most significant new features of the M2 processors. If you're interested in a more complete overview, RevEd has released a comprehensive M2 datasheet (PICAXE-M2 Product Briefing, available at **www.rev-ed.co.uk/docs/picaxem2.pdf**). At this point, we're going to turn our attention to a project that's been on my mind for some time, and the 14M2 is just the processor for the job.



■ FIGURE 2. Breadboard setup for the *serrxd* experiment.

IMPROVING OUR SERIAL LCD PROJECT

Back in June '09, we began our serial LCD project, and in August of that year we developed a major software driver for it. As you know, we used a PICAXE-14M processor which we pushed to its memory and speed limits. Even so, we had to make some compromises, especially in the requirement of sending data to the LCD in fixed-length packets. This was necessary because the 14M's *serin* command is "blocking" which means that whenever the 14M executes a *serin* command, the program stops running until the specified number of bytes have been received. (That's why we needed to know how many bytes were going to be transmitted each time.)

One of the many software improvements in the M2-class processors is the inclusion of a *timeout* option for the *serin* and *serrxd* commands. The timeout option enables us to specify how long (in milliseconds) an M2 program should wait at a *serin* or *serrxd* command before "giving up" and moving on if no data is received. (We aren't going to discuss it this month, but you may be interested in knowing that the M2 *irin* command now also has a timeout option.)

As long as we provide more than enough variables in which to store the incoming serial data, the new timeout option will enable our program to move on to processing the data, no matter how many bytes are actually received. That's exactly the capability we need to improve our original serial LCD project.

With that in mind, as soon as I received my new M2 processors in the mail, I removed the 14M from one of my serial LCDs, replaced it with a 14M2, and began modifying the software. I spent more than two days working on the program, and was getting nowhere. I just couldn't get it to work at all, and the more I tried, the more frustrated I became. (Sound familiar?)

When I couldn't stand it any more, I decided to take a break from it for a day or two, which is a strategy that has helped me in the past. When I returned to it, I simplified my approach, temporarily eliminating the LCD and using the terminal window for output. My goal was to develop a simple *serrxd* routine that would be able to accept a variable number of bytes as input. My breadboard setup for this experiment is shown in **Figure 2**. The programming adapter in the photo is the Prog-03 adapter. It's designed specifically for the new M2 processors. As you can see, it minimizes the number of necessary connections (because it also directly connects to the ground rail).

The program we'll be using (*SerrxdFastSimple.bas*) is included in the download files on the *N&V* website, along with the other two programs we'll be using this month. We'll use the following abridged version of the program to discuss the issues I encountered.

```
' ***** SerrxdFastSimple.bas)
*****

#com 6
#picaxe 14M2
#no_data
#terminal 9600
```

```
setfreq m8

do

b0="*":b1="*":b2="*":b3="*"  ' [1]
          sertxd ("> ")
               ' [2]
          serrxd b0
        ' [3]
          serrxd [1000],b1,b2,b3
        ' [4]
          reconnect
        ' [5]
          sertxd (b0,b1,b2,b3,cr,lf)   ' [6]
          wait 1
loop
```

In line 1, we're initializing the first four GP variables to "*" (ASCII char 42). Don't forget, it's okay to put multiple statements on the same line if you separate them with a colon. Line 2 simply displays a prompt in the terminal window. Line 3 is a traditional blocking *serrxd* command. Its purpose it to make the compiler wait at this point until a serial character is received, and then move on to the non-blocking *serrxd* command in line 4. This command includes a timeout option that instructs the compiler to "give up" after a timeout of 500 mS (at 8 MHz) if no additional serial input is received.

The *reconnect* command in line 5 requires a bit of explanation. We have often sent data from a PICAXE program to the terminal window for display, but I don't think we have done the reverse, i.e., sent data from the terminal window to a PICAXE program. It's easy to do; you just type the data into the *Output Buffer* area at the bottom of the terminal window and click the *Send* button. However, there is one important little complication.

Whenever a PICAXE program is running, the processor repetitively scans the *serin* line to see if the Programming Editor (or AxPad) wants to initiate a new program download. The scanning of the *serin* line would disrupt any serial input we want to send to the running program. To avoid this problem, the PICAXE compiler automatically issues a *disconnect* command in the background (see the *disconnect* documentation in Section 2 of the PICAXE manual). Since the processor is now disconnected from the programming software, it's no longer possible to download a new program without first carrying out a *hard-reset* procedure (i.e., disconnecting the power to the processor, initiating the download, and then quickly restoring the power).

A simpler alternative is to include a *reconnect* command immediately after the *serrxd* commands so that program updates can again be downloaded to the processor without necessitating a hard-reset. Finally, line 6 simply transmits the four characters that are currently stored in the first four GP variables to the terminal window.

So, how should this program behave? Since the four GP variables are each initialized to "*" each time through the loop, entering each of the following strings in the *Output Buffer* and clicking *Send* should result in the indicated output, right?

```
Entering "a" should result in a
display of "a***"
Entering "as" should result in
a display of "as**"
Entering "asd" should result in
a display of "asd*"
Entering "asdf" should result
in a display of "asdf"
```

I imagine you can guess the outcome — it doesn't work! What's a frustrated programmer to do?

THE PICAXE FORUM TO THE RESCUE (AGAIN)

I don't know about you, but I hate to admit when I can't solve a problem. However, I was beginning to think that there might be an issue with the new *serrxd* command, so I put my pride in my back pocket and posted a cry for help on the PICAXE Forum (**www.picaxeforum.co.uk/forum.php**). In about an hour, I received a reply from "Technical" (one member of RevEd's technical support team) confirming that there is, in fact, a bug in the current version of the Programming Editor software (v5.4.0, as of August '11). Ironically, this bug doesn't involve the new *timeout* option of the *serrxd* command; it affects the older blocking version, and results in the command failing to wait for a serial input. The program immediately moves on to the next command even in the absence of any serial input!

Technical reassured me that the bug would be corrected in the next update of the Programming Editor software. Fortunately, Technical also provided a simple work-around that solves the problem until the software update is released. All I had to do was to replace line 3 in the above program with the following code:

```
bugfix:
     serrxd [60000,bugfix], b0
```

In order to fully understand why this work-around is effective, you may want to read the *serrxd* documentation in Section 2 of the PICAXE manual, but I'll briefly explain how it works. The *timeout* of 60000 forces the *serrxd* command to wait about 30 seconds (at 8 MHz) for the first serial character to arrive. If a character fails to arrive in that amount of time, the second optional parameter (*bugfix*) instructs the compiler to jump to the *bugfix* address. Since this address immediately precedes the *serrxd* command, the effect is to keep the compiler from advancing beyond this point in the program until the first character has been received.

If you modify the program to include Technical's work-around, you will see that it behaves exactly as we would predict. By the time you are reading this, version 5.4.1 of the Programming Editor will probably have been released, so you can just install it on your PC and test the program without the "bug fix" — it should work correctly. If not, let me know and we'll figure it out.

In addition to the *serrxd* issue we just discussed, I had two other minor problems as I was experimenting this month. First, in some of the programs I tested the terminal window failed to display the first line of text that was sent to it. Also, even when the first line did show up in the terminal window, there was frequently a single

"garbage" character preceding what I wanted to display.

Since I had already reached my maximum frustration-tolerance level, I didn't hesitate to again ask for help on the PICAXE Forum. Within 20 minutes, I received a reply from "Hippy" (another member of RevEd's technical support team) which explained that a newly downloaded PICAXE program starts running before the terminal window has time to open, so if the program sends data immediately, it will not display. The solution is simple: just put a delay at the beginning of the program.

A couple of hours later, Technical responded to my garbage character issue, asking me which download cable I was using. At the time, I was using a reconfigured FTDI cable with one of my new AxMate-MS programming adapters which — like the Prog-03 in **Figure 2** — is specifically designed for the new M2 processors. (I just can't stop making programming adapters!)

Technical's second reply directed my attention to the schematic diagram for the AXE027 PICAXE USB cable (**www.rev-ed.co.uk/docs/axe027.pdf**). In the schematic, I saw a 10K pull-down resistor on the serin line which I know isn't included in the FTDI cable. In fact, that cable includes a 10K pull-up resistor on the serin line. I wasn't sure what to do about that, but the first thing I tried (just adding a 10K pull-down) worked — the garbage character disappeared. Two problems and two solutions within a couple of hours, and it all transpired on a Saturday — that's outstanding technical support!

Ultimately, I decided to permanently install the pull-down resistor on the bottom of my AxMate-MS adapter (see **Figure 3**). If you are using any FTDI-based USB adapter and getting the extra garbage character, you may also want to consider installing the 10K pull-down resistor.

Once my *serrxd* routine was functioning correctly, I was ready to tackle the more complicated part of my serial LCD update project. One of the goals I had for the update was to be able to change what's displayed on both lines of the 16x2 LCD with one serial string. Since there could be a few LCD commands mixed in with the 32 characters that the LCD can display, I was determined to be able to receive and process strings as large as 40 characters.

As a result, the 28 GP variables that are now available on the M2 processors wouldn't be enough to hold all the incoming data in my grandiose little scheme, so I decided to use the storage variables instead. As I mentioned earlier, the 14M2 contains 484 bytes of storage variables, so I was sure it could handle the largest LCD I could possibly find. In addition, the storage area in all M2 processors can be accessed using what's referred to as *indirect* addressing which is much faster that the usual approach (i.e., *direct* addressing), so that's the next topic we need to discuss.

DIRECT ADDRESSING OF VARIABLES

As we saw earlier in **Figure 1**, the M2 processors have 28 GP variables and many more storage variables. The values of these variables are stored in an area of RAM that's known as the *byte scratchpad*. The M2 GP variables *b0* through *b27* are assigned to RAM locations 0 through 27 in the byte scratchpad, and the storage variable area begins at RAM location 28 and continues consecutively for however many storage variables the processor contains.

For example, the 14M2 contains a total of 512 bytes of RAM in the byte scratchpad (28 bytes for its GP variables and 484 bytes for storage variables). Since the GP variables occupy locations 0 through 27, the storage variable area in the 14M2 extends from location 28 to location 511.

The standard type of addressing that we use when we work with the GP variables is referred to as *direct* addressing because we directly read or write the value that is stored at any given location in the byte scratchpad. For example, the value of the first GP variable (*b0*) resides at RAM location 0, so when we write b0 = 5, we're directly storing the value 5 at location 0 in the byte scratchpad. If we define the name *myVar* for the *b0* variable (*symbol myVar = b0*), then we can also write *myVar = 5* to directly store the value 5 at location 0 in the byte scratchpad.

PICAXE BASIC includes two commands (*peek* and *poke*) that provide a second method of directly addressing the processor's variables. For example, assuming we have already written (*symbol myVar = b0*), *poke myVar, 3* accomplishes the exact same thing as *myVar = 3*; the value 3 is stored in memory location 0. Of course, you probably would never use the *poke* command for that purpose, because it's much simpler to write *myVar = 3*. However, *peek* and *poke* can also be used to directly access the storage variables as well, and the simpler method can't because there are no names assigned

■ FIGURE 3. Pull-down resistor added to the AxMate-MS.

to the storage variables, and no way of assigning names to them.

For example, even if you write *symbol myStoreVar = 28*, you can't write *myStorVar = 7* because you have defined a constant, not a variable. However, you can write poke *myStoreVar, 7* when you want to store the value 7 at the (constant) location 28.

The most important point to remember about the storage variables is that they can only be used to store and retrieve data; we can't assign convenient names to them or use them in any sort of calculations. For those purposes, we need the GP variables. In spite of these limitations, the storage variables are a very powerful feature of the M2 processors. To fully appreciate the power of the storage variables, we need to understand the concept of indirect addressing.

INDIRECT ADDRESSING OF VARIABLES

Indirect addressing is more complicated (and much more powerful) than *direct* addressing. Rather than directly addressing the memory location we want to access, indirect addressing uses the concept of a *pointer* variable which "points to" the desired address. PICAXE BASIC includes the following four built-in special function variables that we can use to implement indirect addressing (I'll explain each one shortly):

bptr – (pronounced *bee pointer*); the byte scratchpad pointer.

@bptr – (pronounced *at bee pointer*); the byte scratchpad value pointed to by bptr.

@bptrinc – (pronounced *at bee pointer inc*); the byte scratchpad value pointed to by bptr (post increment).

@bptrdec – (pronounced *at bee pointer dec*); the byte scratchpad value pointed to by bptr (post decrement).

The byte scratchpad pointer (*bptr*) is used to point to the location we want to access, and the other three variables automatically contain the value that's stored at the location pointed to by *bptr*. I know that sounds confusing, so let's take a look at a simple program to clarify all this. (the line numbers are included for the following discussion):

```
' **StorageVarDemo.bas **
#com 6
#picaxe 14M2
#no_data
#terminal 9600

setfreq m8

bptr = 28               '[1]

for b0 = 65 to 90       '[2]
    @bptrinc = b0       '[3]
next b0                 '[4]

@bptr = 0               '[5]

bptr = 28               '[6]

do until @bptr = 0      '[7]
    sertxd (@bptrinc)   '[8]
loop                    '[9]
```

In line 1, *bptr* is initialized to 28 (the first storage variable location). In lines 2-4, we're executing a *for...next* loop 26 times. The start and end values for the loop (65 and 90) correspond to the ASCII values for "A" and "Z," so the first time line 3 is executed, the ASCII character "A" is stored at the location pointed to by *bptr*, and then *bptr* is automatically incremented in preparation for the next iteration of the loop. (This automatic incrementing of *bptr* is what makes indirect addressing so powerful.)

The second time through the loop, "B" is stored at location 29, and *bptr* is again automatically incremented. When the loop has finished executing, the characters "A" through "Z" have been sequentially stored at locations 28 through 53. At this point, *bptr* = 54 because it started at 28 and was automatically incremented 26 times. In line 5, we're storing the value zero (which is a non-printing ASCII character) at location 54. We're going to use the zero as an "end of string" marker, as we're about to see.

In line 6, we need to reset *bptr* back to its initial value of 28 in preparation for sending the received characters to the terminal window. Here's where our end of string marker comes into play. In lines 7-9, we execute a *do until...* loop that sends each stored character to the terminal window and automatically increments *bptr* in preparation for the next iteration of the loop. When we reach the zero, the loop immediately terminates, so the zero is not sent to the terminal window.

The program we just examined is also included in this month's downloads. If you don't yet have them, it would be a good idea to download all three programs at this point, and spend some time experimenting with *StorageVarDemo.bas* so that you have a good understanding of indirect addressing. When you're ready, we'll move on to our next program.

USING INDIRECT ADDRESSING TO SPEED UP RECEPTION OF SERIAL DATA

This program (*SerrxdFast14M2.bas*) combines the benefits of the *serrxd* timeout option with the speed of indirect addressing to efficiently process large, variable-length serial data strings. Before you run *SerrxdFast14M2.bas*, there are a couple of points I want to mention, so you may want to print out a copy for reference. First, there's no *reconnect* statement in the program; just a *disconnect* statement near the beginning. *Reconnect* can't be used in this program because it loops back around to the first *serrxd* statement (which — as you remember — includes an automatic *disconnect*) so quickly that the program is hardly ever scanning for a new download. Therefore, the only way to initiate a new download in situations like this is to carry out the *hard-reset* procedure mentioned earlier (i.e., disconnect the power to the processor, initiate the download, and then quickly restore the power).

SerrxdFast14M2.bas uses the

same bugfix work-around that was included in *SerrxdFastSimple.bas*, except now we're storing the first data byte in the storage area rather than in a GP variable. The second *serrxd* statement also uses the storage area to hold as many as 49 additional bytes of incoming data. The statement may look a little strange due to its unusual length, but this approach is much faster than trying to accomplish the same thing in a loop.

I assume there is some preset maximum length for a single line in the Programming Editor, but just for fun, I ran the program with twice as many *@bptrinc* parameters in this line and it worked perfectly. Of course, this suggests that we could use the same approach to fill a 20x4 LCD with one serial command, but that's a project for another time!

When you have read through the program and understand how it should work, download *SerrxdFast14M2.bas* to your 14M2 processor and try it out. If you have any problems with it, email me (Ron@JRHackett.net) and I'll try to help.

UPDATING OUR SERIAL LCD "CUSTOM CHARACTER" DRIVER

The *SerrxdFastSimple.bas* demonstrates all the programming techniques we need to update the *LCD16x2-CustCharDriver.bas* program (June '09) so that it can handle variable-length serial input without missing a character. I'm happy to say that I was able to do that without encountering any additional problems along the way. Unfortunately, there isn't enough space to discuss the updated software this month.

However, I also don't want you to have to wait another two months for the denouement of this month's project, so I'll post the updated software, along with an explanation of its features on my website (see **www.jrhackett.net/LCD16.shtml**).

See you next time ... **NV**