Program size, Possible Optimisations and Command Timing

1.0 Introduction

PICAXE microcontroller users often seek information about the timing of various BASIC program commands within the PICAXE system.

The PICAXE microcontrollers do not operate with machine code or assembler, but instead, use a BASIC interpreter which is permanently installed in the PIC chip. Due to the methods use to store the users BASIC program within the PICAXE chip, the time to perform a given command is somewhat variable and for the same PICAXE clock (resonator) speed can vary between the various PICAXE chips.

The following information is a collection of data based upon interpretation between PICAXE manuals and the relevant Microchip PIC datasheets, measurements undertaken by the author (Westaust55) and information previously provided by forum members (Technical, hippy and BeanieBots).

2.0 PICAXE program, data and variable storage

All PICAXE chips incorporate a Bootstrap program and BASIC interpreter, collectively referred to as the firmware, which are permanently held in Flash memory in the PIC chip. The Bootstrap program is used to load the PICAXE users BASIC program from the Programming Editor into the PICAXE chip.

The “M” series PICAXE chips only have sufficient flash memory (2048 bytes) for the firmware and utilise the limited EEPROM memory for both the users BASIC program and the EEPROM data storage area as a shared resource. The program is stored from the top of EEPROM memory (location 255) downwards and the EEPROM data is stored in locates form the bottom (location 0) upwards. The “M” series PICAXE chips have a limited 128 bytes of RAM which must be shared between internal requirements, program variables (b0 to b13) and general RAM accessed with PEEK and POKE commands.

For the larger PICAXE chips, there is sufficient flash memory to enable additional BASIC program commands and store the users BASIC program. This allows the user the full use of the 256 bytes of EEPROM memory purely for the users data storage. Details for some of these larger chips is provided below.

The 18X PICAXE chip contains 7168 bytes of flash memory which is allocated as 2048 bytes for the users program and 5120 bytes for the firmware. There are 256 bytes of EEPROM memory and 368 bytes of RAM which is distributed for internal requirements, program variables (b0 to b13) and general RAM accessed with PEEK and POKE commands.

The 18M2 PICAXE chip contains a total of 4096 bytes of flash memory which is allocated as 2048 bytes for the users program and 2048 bytes for the firmware. There is also 384 bytes of RAM available for internal requirements, program variables (b0 to b27) and general RAM accessed with PEEK and POKE commands.

For the 28X1 and 40X1 PICAXE chips the flash memory totals 8096 bytes of which half (4096 bytes) is used for the users BASIC program. This leaves 4096 for the firmware which is twice the firmware space compared to the “M” series PICAXE chips therefore enabling the inclusion of additional commands. There is a total of 368 bytes of RAM available for internal requirements, program variables (b0 to b27), scratchpad (128 bytes) and general RAM accessed with PEEK and POKE commands.

For the 28X2 and 40X2 PICAXE chips the flash memory totals 32 kBytes of which half (16 kBytes) is used for the users BASIC program as four slots each comprising 4 kBytes. There is then a further 16 kBytes available for the firmware which is four times that provided by the X1 parts and consequently added commands and greater flexibility of the IO is available. These X2 parts also have more RAM available with a total of 1536 bytes which allows the provision of a larger number of program variables (b0 to b55)   and a larger scratchpad memory area (1024 bytes) and general RAM accessed with PEEK and POKE commands.

3.0 PICAXE Program Format and Storage

Within the Programming Editor and the users computer, the users BASIC program is stored as a simple text file with a .bas file extension/type that can be opened and edited in various other text editor programs, such as MS Notepad.

When the BASIC program is downloaded to a PICAXE microcontroller, the Programming Editor creates a tokenised version of the users BASIC program which is more compact and therefore requires less program space within the PICAXE chip. This is essential when you consider that the "M" series PICAXE chips only have a total of 256 bytes of program space.

Instead of downloading BASIC program command keywords such as READ, PAUSE, SEROUT, i2cSLAVE, etc, as text equivalents which would require many bytes of program space for each command, to reduced space as is common with interpreted BASIC each command/keyword is changed to a unique token. Furthermore, math operators such as +, - , / and * are also converted to unique tokens.

Program labels are not loaded as text strings into the PICAXE but are converted to an address and stored in a compressed format. Likewise constants defined using SYMBOL statements are saved directly into the program at the relevant locations as compressed values where the number of bits relates to the number of bits needs to hold the value along with an overhead.

For computers using interpreted BASIC, the tokens typically use a full byte (8-bits) and some extended BASIC languages may use two byte tokens. With, the reduced instructions and need to maximise use of the available program memory space, the PICAXE system uses further compression based on the use of 4 and 5 bit tokens within the PICAXE system.

Program data held in EEPROM and RAM is all based on byte boundaries thus data with values in the range 0 to 255 will always occupy 1 byte (8-bits) for each byte value stored (or two bytes for a word variable/value).

3.1 PICAXE Number Compression and Representation

As mentioned above, numbers such as constants are not stored as 8-bit values in the program space but in a compressed format whereby the more frequently used smaller values occupy less space. The space requirements within the PICAXE program area for numbers are defined as follows:

Number Value	**Bits Req'd for Number**	**Number Size Bit Combination**	**Total Overhead Bits Required**	**Total Bits Used**
0 to 1	1	%00	3	4
2 to 15	4	%01	3	7
16 to 255	8	%10	3	11
256 to 65535	16	%11	3	19

The overhead part requiring the extra 3 bits comes from a need for a bit to flag that it is a number (and not a variable) and two more bits to indicate the "size" of the number.

Because of the way that the PICAXE program is compressed on a bit boundary, all numbers are shuffled up with minimal leading zeros. As such they can virtually be though of as left justified. The following table depicts how numbers are stored within the PICAXE programs

Number Value	Bits Required in Program	Number Size bits	Number Representation in Program Space			
			←msbit			lsbit →
0	1	%00	0	—	—	—
1	1	%00	1	—	—	—
2	4	%01	0010	—	—	—
3	4	%01	0011	—	—	—
4	4	%01	0100	—	—	—
5	4	%01	0101	—	—	—
6	4	%01	0110	—	—	—
7	4	%01	0111	—	—	—
8	4	%01	1000	—	—	—
9	4	%01	1001	—	—	—
10	4	%01	1010	—	—	—
11	4	%01	1011	—	—	—
12	4	%01	1100	—	—	—
13	4	%01	1101	—	—	—
14	4	%01	1110	—	—	—
15	4	%01	1111	—	—	—
16	8	%10	0001	0000	—	—
17	8	%10	0001	0001	—	—
18	8	%10	0001	0010	—	—
19	8	%10	0001	0011	—	—
20	8	%10	0001	0100	—	—
21	8	%10	0001	0101	—	—
22	8	%10	0001	0110	—	—
23	8	%10	0001	0111	—	—
24	8	%10	0001	1000	—	—
25	8	%10	0001	1001	—	—
26	8	%10	0001	1010	—	—
27	8	%10	0001	1011	—	—
28	8	%10	0001	1100	—	—
29	8	%10	0001	1101	—	—
30	8	%10	0001	1110	—	—
31	8	%10	0001	1111	—	—
32	8	%10	0010	0000	—	—
64	8	%10	0100	0000	—	—
128	8	%10	1000	0000	—	—
256	16	%11	0000	0001	0000	0000
511	16	%11	0000	0001	1111	1111
512	16	%11	0000	0010	0000	0000
1024	16	%11	0000	0100	0000	0000
2048	16	%11	0000	1000	0000	0000
4096	16	%11	0001	0000	0000	0000
8192	16	%11	0010	0000	0000	0000
16384	16	%11	0100	0000	0000	0000
32768	16	%11	1000	0000	0000	0000
65535	16	%11	1111	1111	1111	1111

One thing that can be seen is that the larger a number, the more bits are needed to hold in a token.

Thus a PAUSE 260 will require 8 bit more program space than a PAUSE 255 command so it is clear that careful use of numbers can improve/reduce program size.

Likewise the program space for the overhead in using a loop as:

```
FOR b0 = 1 to 16
NEXT b0
```

requires 4 more bits of program space than a loop using

```
FOR b0 = 0 to 15
NEXT b0
```

even though both will go through the loop structure the same number of times

As another example, the program snippet:

```
HIGH 7
PAUSE 100
LOW 7
```

requires more 6 more bits of program space than using:

```
HIGH 1
PAUSE 100
LOW 1
```

So it would seemingly be prudent to allocate the lower two input and output pins on earlier PICAXE parts and the lower two IO pins on the later X2 and M2 parts**.**

3.2 PICAXE Program Compression

As discussed above, the tokens as used for BASIC program commands and math operators are also compressed. With the compressed 4, 5 or 6 bit tokens as used for commands and math operators, variables, labels, etc, the PICAXE systems shifts the tokens up so that there is no unused bits and the program file is based upon on 1-bit boundaries rather than byte boundaries.

The program space requirements needed for these program parts are detailed below (based upon the publically and readily available information):

Program Part	**Bits for Token**	**Comment**
Commands (LET, IF, HIGH, etc)	5 or 6	5 bits for M series & early chips 6 bits for **X1** and later chips
Labels (after THEN, GOTO, GOSUB)	11 / 14 / 15	11 bits required for 256 byte addr. 14 bits required for 2048 byte addr. 15 bits required for 4096 byte addr.
Math operators (=, -, +, *, **, /, //, etc)	4 or 5	4 bits for pre- X1 parts (15 functions) 5 bits for **X1**/**X2** parts (30/32 funct'n)
Variables (bit0 - bit7, b0 – b13, w0 – w6)	6 to 8	6 bits for M series & early chips 7 bits for **X1** parts 8 bits for **X2** parts
Bit "flags" (indicate when something does/doesn't follow – eg comma)	1	

Some program commands apparently just happen to also use additional bits but no specific details have been published/posted.

From Clive Seager as quoted on the forum by forum member **Beaniebots** with respect to the X1 parts:

> **"**Programs will be longer. This is because each command is now 6 bits long (was 5) to allow support of all the extra commands and similarly the variables are also a bit longer. This is why we say ~600 lines for 28X and ~1000 (not 1200!) for the 28X1. However there is also some better compression on some commands to compensate**"**.

From looking at early versions of the PICAXE manuals, there were only Math related 15 commands including the equals (=) so that these could be contained within 4 bits. Newer PICAXE chips support more twice the number of commands which requires a minimum of 5 bits. To cater for this the extra bit is placed at the front/left of the token and for the existing tokens the extra bit has a value of "0". For the newer X1/X2 commands in many cases the msb is a "1" but for the Unary math functions the msb may be a "0" or a "1" and discrimination from other tokens is achieve with other bits/flags.

These math operators are followed by a bit flag where "0" indicate the last operation and "1" indicates a further following operation.

3.3 Variable Tokens

As mentioned in section 3.2, the number of bits used to represent a variable has increased as developments with the PICAXE chip range have occurred. This has been necessary to provide "space" for the additional variables. The following table based upon current PICAXE manual data shows how the number of variables has increased across the PICAXE range over time.

PICAXE type	**Variable Type**		
	Bit	**Byte**	**Word**
"straight", A, X and M parts	bit0 to bit15	b0 to b13	w0 to w6
X1 parts	bit0 to bit31	b0 to b27	w0 to w6
M2 Parts	bit0 to bit31	b0 to b27	w0 to w13
X2 Parts	bit0 to bit31	b0 to b55	w0 to w27

As the number of variables has increased/doubled so the number of bits to represent a variable has also increased by one bit for each doubling of the quantity of variables.

Although a given number of bits can represent a specific number of values, for example 8 bits can hold a value from 0 to 255, there is also a need to incorporate some information to indicate whether the program token represents a bit, byte or word variable. This is part of the reason why there are 14, 28 and 56 byte variables rather than the possibly expected quantities of 16, 32 or 64 respectively.

My understanding is that for the X2 parts, the variable tokens all occupy 8 bits of program space and can be defined by the bit patterns given in the following tables. Note that these are my interpretation and may be incorrect – provided only for concept and I take no responsibility for errors or omissions.

3.3.1 X2 Bit Variables

P.E. Variable Name	Program Space Token		P.E. Variable Name	Program Space Token
Bit0	1000 0000		Bit16	1001 0000
Bit1	1000 0001		Bit17	1001 0001
Bit2	1000 0010		Bit18	1001 0010
Bit3	1000 0011		Bit19	1001 0011
Bit4	1000 0100		Bit20	1001 0100
Bit5	1000 0101		Bit21	1001 0101
Bit6	1000 0110		Bit22	1001 0110
Bit7	1000 0111		Bit23	1001 0111
Bit8	1000 1000		Bit24	1001 1000
Bit9	1000 1001		Bit25	1001 1001
Bit10	1000 1010		Bit26	1001 1010
Bit11	1000 1011		Bit27	1001 1011
Bit12	1000 1100		Bit28	1001 1100
Bit13	1000 1101		Bit29	1001 1101
Bit14	1000 1110		Bit30	1001 1110
Bit15	1000 1111		Bit31	1001 1111

3.3.2 X2 Byte Variables

P.E. Variable Name	**Program Space Token**		**P.E. Variable Name**	**Program Space Token**
B0	0010 0010		B28	1010 0010
B1	0010 0011		B29	1010 0011
B2	0010 0100		B30	1010 0100
B3	0010 0101		B31	1010 0101
B4	0010 0110		B32	1010 0110
B5	0010 0111		B33	1010 0111
B6	0010 1000		B34	1010 1000
B7	0010 1001		B35	1010 1001
B8	0010 1010		B36	1010 1010
B9	0010 1011		B37	1010 1011
B10	0010 1100		B38	1010 1100
B11	0010 1101		B39	1010 1101
B12	0010 1110		B40	1010 1110
B13	0010 1111		B41	1010 1111
B14	0110 0010		B42	1110 0010
B15	0110 0011		B43	1110 0011
B16	0110 0100		B44	1110 0100
B17	0110 0101		B45	1110 0101
B18	0110 0110		B46	1110 0110
B19	0110 0111		B47	1110 0111
B20	0110 1000		B48	1110 1000
B21	0110 1001		B49	1110 1001
B22	0110 1010		B50	1110 1010
B23	0110 1011		B51	1110 1011
B24	0110 1100		B52	1110 1100
B25	0110 1101		B53	1110 1101
B26	0110 1110		B54	1110 1110
B27	0110 1111		B55	1110 1111

3.3.3 X2 Word Variables

P.E. Variable Name	Program Space Token		P.E. Variable Name	Program Space Token
W0	0011 0010		W14	1011 0010
W1	0011 0100		W15	1011 0100
W2	0011 0110		W16	1011 0110
W3	0011 1000		W17	1011 1000
W4	0011 1010		W18	1011 1010
W5	0011 1100		W19	1011 1100
W6	0011 1110		W20	1011 1110
W7	0111 0010		W21	1111 0010
W8	0111 0100		W22	1111 0100
W9	0111 0110		W23	1111 0110
W10	0111 1000		W24	1111 1000
W11	0111 1010		W25	1111 1010
W12	0111 1100		W26	1111 1100
W13	0111 1110		W27	1111 1110

3.3.4 Other X2 Variables

There are a number of other variables and while the following list may still be incomplete it provides the program space token for some of those additional variables.

P.E. Variable Name	Program Space Token		P.E. Variable Name	Program Space Token
pinsA	1010 0000		s_w0	0011 0001
pinsB	0010 0000		s_w1	0011 0011
pinsC	0110 0000		timer3	0011 0111
pinsD	1110 0000		compvalue	0011 1001
dirsA	1010 0001		hserptr	0011 1011
dirsB	0010 0001		hi2clast	0011 1101
dirsC	0110 0001		timer	0011 1111
dirsD	1110 0001		ptr	1011 1101
outpinsA	1111 0101		adcsetup	1011 1111
outpinsB	1111 0001		flags	1111 1001
outpinsC	1111 0011		bptr	1111 1011
outpinsD	1111 0111			

3.3.5 X2 Variable Summary

For other PICAXE parts, the program tokens may differ but the concept is the same as provided in the above tables.

From the above table, it an be seen that bits 4 and 5 in each token are unique to the variable type. So a program token with the format;

- %xx00 xxxx or %xx01 xxxx represents a bit variable,
- %xx10 xxxx represents a byte variable, and
- %xx11 xxxx represents a word variable.

Note also that:

- Both byte and word tokens commence with %0010 as the lower nybble
- Word and byte tokens are aligned for the lower nybble.

For the latter point, if we consider say w5 which uses the same memory locations as bytes b11:b10 The lower nybble for w5 is 1100 which is the same as the lower nybble token for b10.

From the information provided in this section, it is obvious that using smaller numeric values for the data will save space, however there is no space savings gained with respect to the actual program tokens for the variable names.

As mentioned earlier, the information pertaining to the numeric values in section 3.1 and the X2 series program variable tokens in section 3.3 are based upon my investigations over a couple of hours using nothing more than a PICAXE chip and the Rev Ed Programming Editor (V5.3.4) software without any “skimming devices” or other electronic tools.

3.4 GOSUB Command Overheads

When a BASIC program uses a GOSUB command to perform the code within the specified subroutine, the PICAXE must have a means to identify the point in the program from which the branch to the GOSUB occurred to enable the program execution to ultimately RETURN to that point.

If the PICAXE firmware was to store the actual address of the address token after a GOSUB command, for the “M” series parts this would require saving 11 bits onto the “return” stack. For a four deep stack as available for the “M” PICAXE parts, this would require 44 bits which needs a total of 6 bytes. When one considers that the “M” series PICAXE parts only have 128 bytes of RAM registers of which 48 are made available for the PICAXE user, the 6 bytes is a lot compared to the remaining RAM available for all of the PICAXE internal functions.

Even for the X1 and X2 parts which allow an eight deep stack, that would equate to 15 bytes of RAM (8 addresses each of 15 bits) for the return stack from a total of 368 bytes of RAM, after 128 bytes is allocated for the scratchpad and a further 95 bytes made available to the users program.

To reduce the stack size requirements, every GOSUB is assigned a number (0..15 for “M” parts and 0..255 for larger PICAXE chips). It is this number which is pushed to the stack. That number only requires four bits for the “M” series PICAXE chips so the stack is reduced to just two bytes of RAM in total. For the larger parts with a stack depth of eight that requires just eight bytes of RAM in total. That gives a good saving in scare RAM memory resources.

When a RETURN is executed at the completion of the subroutine, it pops the subroutine’s 4-bit or 8-bit number back off the stack which is the equivalent to “jump to the nth GOTO address in a look-up table to determine the return address. The Return address lookup table is held in the program memory of which there is far more than there is RAM. The address in the look-up table takes the program execution back to the command immediately after the calling GOSUB.

4.0 A Brief Look at Program Optimisation

On the “X” parts (eg 18X), selecting 256 GOSUB’s as opposed to 16 GOSUB’s in the Programming Editor (View/Options) will use more program space so always select the 16 GOSUBs options unless you really do need more GOSUB’s.

The LOOKUP and LOOKDOWN commands are particularly memory ‘hungry’ as each 8-bit value (ie values are 16 to 255) requires 11 bits for the value and one more bit for the separator bit flag resulting in 12 bits per 8-bit value.

For example the program line:

```
LOOKUP b5, (100, 150, 200), b6
```

Based on the tables above, and considering that the bit usage given below may not be perfectly accurate it does give a reasonable idea, the anticipated total required program space bits are as follows:

Line Part	Details	'M' series Bits Used
Command	LOOKUP	5
Variable	B5	6
Separator/More Flag	,	1
Bracket	(	0
Value	100	11
Separator/More Flag	,	1
Value	150	11
Separator/More Flag	,	1
Value	200	11
Bracket	)	0
Separator/More Flag	,	1
Variable	B6	6
Total Program Space require =		54 bits

Performing a Syntax check in the P.E. for both an 08M and 40X1 indicates 10 bytes used. If we deduct the 3 bytes for an empty program we have 7 bytes which equates to a maximum of 7 x 8 bits = 56 bits so the above calculation is roughly on target.
If one of the values in changed to greater than 255 then the P.E. syntax check shows one extra byte required which is as expected.
Adding one extra value less than 256 and performing a P.E> Syntax check shows 12 bytes used and as we have introduced 11 bits for the extra value plus 1 bit for the "More flag" the values still appear to be relatively in order.

By comparison, using the following program snippet to achieve the same thing:

```
EEPROM 0,(100, 150, 200)
READ b5, b6
```

Line Part	Details	'M'Series Program Bits Used	Larger PICAXE Program Bits Used
PE Directive	EEPROM	0 (PE directive only)	0 (PE directive only)
Value	0	0 (Part of PE directive)	0 (separate EEPROM area)
Separator/More Flag	,	0 (Part of PE directive)	0 (separate EEPROM area)
Bracket	(	0 (Part of PE directive0	0 (Part of PE directive
Value	100	8 (EEPROM always 8 bits)	0 (separate EEPROM area)
Value	150	8	0
Value	200	8	0
Bracket	)	0 (Part of PE directive	0 (Part of PE directive)
Command	READ	5	6 (based on past forum post)
Variable	b5	6	7 for X1**?** and 8 for X2
Separator/More Flag	,	1	1
Variable	b6	6	6
Total Program Space require =		42 bits	20 bits (X1) or 21 bits (X2)

So seemingly with the example above, using the EEPROM and READ commands saved 22% with the "M" series chip and a whopping 66% for the larger (X, X1 or X2) PICAXE chip.

Using the P.E. Syntax check indicates that for the 08M a total of 10 bytes are used (= 7 after allowing for 3 bytes for the empty program) and for the 40X1 a total of 6 bytes are used (= 3 bytes after allowing for 3 bytes for the empty program).

Note that while the earlier PICAXE chips used 3 bytes for an 'END' marker in an empty program, for the **X2** parts, this has been reduced to 2 bytes (as identified with the PE syntax checker).

While the 3 byte space requirement falls within the anticipated space for the 40X1 version (as 20 bits = 2.5 bytes) the requirement for 7 bytes for the 08M version is roughly 1 bytes greater than anticipated (since 42 bits = 5.25 bytes which would report as 6 bytes). Discrepancies aside however, the program space requirements are generally of the right order of magnitude.

Another interesting comparison, albeit only available for the X1 and X2 parts, is between the NOT and INV commands. Consider the following examples:

```
        b0 = NOT b3          ; available for all PICAXE chips according to recent manuals
                               but not mentioned in earlier (eg V5.2) versions of the manuals.
and
        b1 = INV b3          ; this command only exits for X1 and X2 parts.
```

Both commands achieve the same end result in that they bit-wise invert the individual bits from the byte value. However the NOT command uses three more bytes of program space than the INV command. This suggests that the NOT command may be some form of "maco" since it needs 24 bits (3 bytes) more program space.

In summary, while the use of EEPROM memory, particularly for the larger PICAXE parts can save program space, there is still an advantage that the LOOKUP/LOOKDOWN commands has over the EEPROM command in that variables can be used to return data, and data can have a value greater than 255. LOOKUP allows the effective creation of an array of bit/byte/word variables.

By contrast, using EEPROM memory with the READ command has the advantage that all data uses exactly 8-bits, and at run-time the Firmware doesn't have to step through the LOOKUP list to find the data needed and finally jump to the end of the LOOKUP command to continue with the next program statement. As a consequence, using EEPROM memory and the READ command is much quicker. There's also the advantage as stated that the EEPROM data is often held separately from program code, which also has the advantage of saving program memory space.

Some optimisations are not immediately obvious as has been highlighted on the PICAXE forum.

```
        LET b0 = b0 + b1 – this uses ~3 bytes with a 40X1 chip and 4 bytes with a X2 chip.
        LET b0 = b1 + b0 – this uses ~5 bytes with a 40X1 chip and 6 bytes with a X2 chip.
```

If you try the above, keep in mind that the P.E adds an "invisible" END command (3 bytes on pre-X2 parts and 2 bytes on X2 parts) to any program, so you will see 6 and 8 bytes respectively for the two above program lines with, for example, an X1 chip.

Not only does the first example save some program space, but as can be seen from the timing examples in section 5.0, in the case where the variable for the result is the first variable after the "=" results in approximately a one-third reduction in the time to perform the calculation.

The left to right math equation computation means that incorporating a previous result into the same variable as part of a more complex calculation may require the initial variable be appended at the end akin to the second case above. However it does indicate that one should consider how to undertake the math equations where possible.

As the BASIC program tokens are not byte-sized, and because the Programming Editor only reports the size of program in bytes it is not clear exactly how much space is used when adding, changing or deleting a command. A two byte long program may have had 9 bits to 16 bits used, adding another command may or may not show up as an increase in size. Program optimisation is therefore something of a “Black Art” which can be a little hit and miss, and not helped by the fact that you are trying to save bits, but only get told how many bytes or part bytes you've used. A one bit increase can appear as a one byte increase and a seven bit reduction may not show at all.

5.0 Command Timing

Any attempt to rely on timing for a particular PICAXE command is fraught with difficulties.

For example the simple empty program loop:

```
FOR b0 = 0 TO 65525
NEXT b0
```

Was measure to take approx 65 seconds using an 08M but approx 75 seconds using a 40X1 with both PICAXE chips running at 4 MHz using the internal resonator.

The above timing difference may be attributed in part to resonator variance/tolerance and in part to the placement of the tokens crossing byte boundaries and requiring additional time to “extract” the tokens and execute the program. A smaller part of the time difference could even be attributed to the 08M program running in EEPROM memory while the 40X1 program is stored in flash memory which is marginally slower to access.

Not withstanding the variances that will occur between the various PICAXE chips and the position of the program tokens relative to byte boundaries, the following information is provided as a rough guide to program command execution times. The data is provided here for completeness of providing as much detail about the PICAXE program space and time as can readily be found or determined.

The timing data for the 28X chip is based on past data posted by BeanieBots on the PICAXE forum using a second 28X to record the time intervals. The data for the old 08 chips was posted by hippy on the PICAXE forum. The remaining timing data was measure by the author (Westaust55) using a PICAXE 40X1 (firmware VB.1)

The test result below were obtained using two PICAXE chips running at the default resonator speed. The first PICAXE was set up to measure the duration using the PULSIN command and output the timing results to an LCD in units of 10 usec.

The second PICAXE had the following program:

```
Main:
   High 1
   'command under test
   Low 1
   Goto Main
```

Most of the commands were tested with little variation noticed between their timings so I will only mention the different ones. I also added a line before and after the code to see if program position had much effect. There is about a 2% variation depending on code position.
The numbers stated are those as read from the LCD so it is up to you to interpret them.

Note that the test results in the table below are adjusted to correct for the fact that the PULSIN command registers in 10 microsecond increments with a 4 MHz clock speed.

PICAXE Chip ==>	**08**	**08M**	**18X**	**18M2**	**28X**	**28X1**	**28X2**	**28X2**
Firmware Ver	4.3	9.2	8.8	2.A	7.1	A.2	B.1	B.1
Clock Speed	4 MHz	4 MHz	4 MHz	4 MHz	4 MHz	4 MHz	**4** MHz	**8** MHz
Source	Hippy	WA55	WA55	WA55	Beaniebots	WA55	WA55	WA55
Command								
Duration	(usec)	(usec)	(usec)	(usec)	(usec)	(usec)	(usec)	(usec)
PAUSE 1		1,250	1,220	1,290	1,450	1,240	2,320	1,160
PAUSE 15		15,320	14,780	15,360		14,770	3,0310	15,140
PAUSE 255		256,110	246,830	255,820		245,920	509,220	254,400
PAUSE 256		257,140	247,900	257,120		246,970	511,290	255,460
GOTO	266	280	330	810		390	470	230
GOSUB	330	900	1,260	3,150	1,310	1,490	1,730	860
RETURN	510							
HIGH 0	230	250	250	310	520	280	390	190
HIGH 4	255	290	270	340		320	440	220
LOW 0	226	250	250	310	520	280	380	190
LOW 4	254	290	270	340		320	440	220
IF b0=1 THEN		550	600	710	840	690	810	400
IF b0=255 THEN		620	680	780	920	750	910	450
IF b0 = b1 THEN		520	680	1,230	880	800	960	480
IF w0 = w1 THEN		540	700	1,260	900	820	990	490
for the **28X** other IF…THEN tests/comparisons similar test results were recorded (as advise by **BB**)								
b0 = 0	329	360	350	450		420	510	250
b0 = 2	355	380	390	470		450	540	260
b0 = 16	397	430	430	510		510	580	290
w0 = $FFFF	480	540	530	640		610	720	360
b0 = 0 + 0	629	670	680	810		800	970	480
b0 = 0 - 0	616	660	670	800		790	950	470
b0 = 0 * 1	777	830	840	970		960	1,120	560
w0 = $FFFF * $FFFF	1,156	1,250	1,290	1,450		1,420	1,620	810
w0=w0 * w0 (with w0 = 0)	645	690	680	840		810	970	480
w0=w0*w0 (with w0 = $FFFF)	650	690	690	850		810	970	480
b1 = b1 + b2		500	500	650	1,000	630	770	380
b1 = b2 + b1		760	770	960		950	1,140	570
PEEK 0, b0		360	360	540	630	490	580	290
PEEK 255, b0		440	430	610	680	570	650	320
POKE $50, b0		440	440	590	700	540	630	310
RANDOM b0		270	250	350	500	350	410	210
READ 0, b0		500	550	520		500	580	290
READADC 0, b0 (pin1 for 08M)		500	500	690	680	950	1,050	520
READADC10 0, w0 (pin1 for 08M)		520	520	700	680	960	1,070	530
TOGGLE 0	227	250	250	360		290	380	190
TOGGLE 2		290	270	420		330	440	210
TOGGLE 4	254	290	270	420	520	330	440	220
WRITE 0, b0		4,290	5,390	3,210	5,090	4,240	3,550	3,090
SEROUT 0, N2400, (“”)		5,210	5,150	5,380	5,370	5,190	5,180	4,890
SEROUT 0, N2400, (“AAA”)		15,160	15,010	15,610	15,100	15,030	14,790	14,300
REM **or** #REM / #ENDREM		0	0	0	250	0	0	0
DEBUG		373,700	372,280	127,450	327,200	150,010	183,980	183,810

The increased speed through use of pins 0 and 1 versus use of pins 2 and up (due to token size differences) can be clearly seen, as can the token-sizing effects can also when it comes to variable assignment .

The 'cost' of mathematical operations is quite interesting. Adding +0 adds 300 usec and the -0 adds 287 usec. Looking at the effect of math calculations really highlights how hard it is to predict timing because it can (but not always) be greatly affected by what numbers are being used.

As can also be seen from the tabulated data, the newer M2 and X2 parts (due to greater complexity, flexibility and other firmware matters to keep things ticking along including the need to process more complicated command tokens, check more things like interrupts, and perform the mapping of port data to actual I/O pins), takes on average approximately 30% longer at the same clock speed to perform the same command as the “M” “X” or “X1” series parts.

- -

Some timing comparisons made using an oscilloscope by forum member MartinM57 are as follows:

(see forum thread: http://www.picaxeforum.co.uk/showthread.php?t=13262)

20X2 Program loop:

```
Setfreq m64
DO
   Toggle B.0
   Toggle B.0
   ; or other command
   ; as per table below
LOOP
```

08M Program loop:

```
Setfreq m8
DO
   Toggle 1
   Toggle 1
LOOP
```

PICAXE Chip ==>	**08M**	**20X2**
Firmware Ver	n/a	n/a
Clock Speed	8 MHz	64 MHz
Source	MartinM57	MartinM57
Command		
Duration	(usec)	(usec)
Toggle B.0 (twice)	125.8	31.78
High B.0 : Low B.0		31.79
pinB.0 = 0 : pinB.0=1		34.65

- -

Some further speed test data previously provided by PICAXE forum member **BeanieBots** is as follows using a now superseded PICAXE 18 back in 2003:

(see: http://www.picaxeforum.co.uk/showthread.php?t=618)

Loop1:
  pins=pins^$FF
  Goto Loop1
	f=557.2 Hz
	t = 1,795 usec per loop

Loop1:
  High 1
  Low 1
  Goto Loop1
	f=1.143 kHz
	t= 875 usec per loop

Loop1:
  For w0=0 to 65535
    High 1
    Low 1
  Next w0
  Goto Loop1
	f=617.4Hz
	t= 1,620 usec per FOR…NEXT loop

Loop1:
  For w0=0 to 65535
    High 1
    b3 = b4
    Low 1
  Next w0
  Goto Loop1
	f=464.0 Hz
	t= 2,155 usec per FOR…NEXT loop

Loop1:
  For w0=0 to 65535
    High 1
    Readadc10 0, w2
    Low 1
  Next w0
  Goto Loop1
	f=478.3 Hz
	t= 2,091 usec per FOR…NEXT loop

These tests were done at an ambient of 18C.
Touching the PIC with a finger gave a rapid 2% rise in frequency which slowly fell after removing the finger so I assume the change is temperature related rather than capacitive