Getting started: Setting up the DYNAMO cross-development system:

Step 1: Copy the DYNAMO folder onto your Macintosh. Then rename the folder to indicate the project.

Step 2: Create a ProDOS disk. It should be initialized at a 4:1 interleave. ProDOS should be copied onto it (get the ProDOS file from a bootable ProDOS 8 disk, or get the P8 file from the System folder of a GS/OS disk and rename it to ProDOS on the floppy). The BUILDAPP.SYSTEM and BUILDAPP.TEXT files (found in the folder prodos.stuff) should be copied to the ProDOS disk via Apple File Exchange or copied over a server. When copied to the ProDOS disk, the filetype/auxtype of BUILDAPP.SYSTEM should be $FF/$0000, and the filetype/auxtype of BUILDAPP.TEXT should be $04/$0000.

Step 3: Start MPW by clicking on any of the source files in the project directory. Do a full-build of SAMPLE. When prompted, insert the ProDOS disk.

Step 4: Insert the ProDOS disk into the Apple II (or IIGS). Boot it. (Or double click on BUILDAPP.SYSTEM from the finder.)

You should now be running the DYNAMO sample application. This sample is simply an exerciser of the DYNAMO runtime routines. It tests the routines, and also the source code demonstrates using the runtime and macro libraries.

Now, for more detail:

The BUILDAPP.SYSTEM program allows the developer to build a multiple-segment application into one program file, and to launch it simply on the //e. Segments are automatically relocated to appropriate banks of memory, and then the application is launched starting at any address in any bank chosen.

BUILDAPP.SYSTEM uses a text file as a script for building the application. The default name for this text file is BUILDAPP.TEXT. This can be changed by modifying the pathname in the program BUILDAPP.SYSTEM. The pathname starts at the sixth byte of the file, memory location $2006 if BLOADed. The pathname is a pascal string. (This is the standard ProDOS way of giving the launched application a pathname.)

To use the BUILDAPP.SYSTEM program, you will need a disk with ProDOS, the BUILDAPP.SYSTEM program, a build script file called BUILDAPP.TEXT, and all the segments of your application on one disk. This disk is now bootable. When the disk is booted, BUILDAPP.SYSTEM will construct a single executable block of code starting at address $2000. Then it will prompt you as to whether or not you want to save this image. During development, you will often choose not to save it, since it takes just that much longer to get the application

executing. Then you can test your latest addition or change to your application. To make a bootable disk without `BUILDAPP.SYSTEM`, you just save the application image to disk when prompted, and then copy the application to a disk with

just ProDOS on it. If the application name ends with .SYSTEM, and it has a filetype of $FF, you have a bootable application.

You may wish to develop on one machine, which writes the segments to the ProDOS disk, and then just boot the disk on another machine. This saves a lot of time booting into your development environment. There is also the advantage of always being able to look at the most current source while executing the program. You don't need two machines to develop, but it will definitely speed up the add/change/test development cycle every programmer goes through.

How the application is build depends on the BUILDAPP.TEXT script file. Here is the explanation of this file:

The first line of the build script is the pathname that will be placed at $2006 of the application. You may have no need for this, but if you are developing a system application, launchers may place a pathname there, and you may want a default.

Lines 2-N are for the segments to be loaded. The first field is the mode byte. Bits 0-2 of the mode byte indicate what bank of memory the segment should be transferred to at execution time.

Bit 0 indicates ROM or language card. Off for ROM, on for language card.

Bit 1 indicates which language card (if bit 0 is on). Off for primary language card, on for auxiliary language card.

Bit 2 indicates 1st or 2nd $D000 bank of language card. Which language card is still indicated by bit 0 and 1. Off selects the secondary $D000 bank ($C083), and on selects the primary $D000 bank ($C08B).

Bit 3 indicates primary or auxiliary 48k. Off for primary, on for auxiliary.

Bit 4 indicates that the auxiliary stack pointer should NOT be initialized. If you load a block of code or data into the auxiliary language card, the auxiliary stack pointer is initialized to $FF unless you indicate otherwise by setting this bit on.

Bit 7 of the mode byte being on indicates the end of the segment list. Bits 0-3 still indicate a bank of memory, and the following field indicates the launch address within that bank.

The second field is the load address. This is the address the segment will be moved to at execution time.

The third field is the name of the segment itself. This can be a full or partial pathname.

The next two lines indicate whether you want hex or decimal information indicating starting address and total application size.

The final line indicates the application filetype, auxtype, and filename. The application builder

will ask if you want the application saved. After the file is saved (or not), the application is launched for testing.

All numeric fields can be decimal or hex.

The below is a sample build script that was used to make the application builder itself.

There is only one segment for this application. The runtime was linked into the main code module, generating only one binary code segment.

```
BUILDAPP.TEXT             ;The default script filename.
0,$800,BUILD.BIN          ;The buildapp application.
$80,$800                  ;Starting address.
$                         ;Display starting address in hex.
$                         ;Display application size in hex.
$FF,0,BUILDAPP.SYSTEM     ;Filetype, auxtype, and filename.
```

The `BUILDAPP.TEXT` file can be up to 2045 bytes long.

Bank switching will still have to occur within the application, as necessary. There is no getting around that on a //e. `BUILDAPP.SYSTEM` is to simplify the loading and starting of an application, by making it a single, launchable file.

Below are the macros included in the runtime, and their basic usage:

OPERAND USAGE:

If the operand represents an integer variable or string, then you do not put a # in front of it. Variables and strings are always either a ldx literal, or a ldy literal, so the # is assumed. When the macro is expanded, the # is put there automatically.

If the operand represents a value, then you will need to put a # or an * in front of the operand. If you put a # in front of the operand, then it is a literal. If you put an * in front, then it uses the value at that address. This is similar to the C unary operator *.

These macros are interfaces for the runtime routines associated with them. The runtime routines handle up to 128 integer variables, and up to 256 strings. The runtime also has multi-dimension array support. The integer functions are simple add,sub,mul,div, and others. These others include mass-initialization, min, max, print decimal, etc. The string functions are most of what is available in AppleSoft, in various forms. The array functions are used to define a base pointer to a row of dta within the array, and then to index into and out of that row.

The principle of the runtime routines is that the xreg holds a destination variable number (for ints: 0-254, for strings: 0-255). All runtimes preserve the xreg, therefore, you can do multiple operations to a single variable without having to reload the xreg. The values that are used on the xreg variable (the source data), is one of 3 forms for integers:

1. 1-byte value

2. 2-byte value
3. 2-byte integer variable.

1-byte values are placed in the acc. 2-byte values are placed in the acc,y (acc=lo, y=hi). 2-byte integer variables have the variable number placed in the yreg. (The yreg is not preserved by the runtime routines.) Once the source data is loaded (in acc, acc-y, or y), the proper call to the runtime routines is made. The 'proper' routine is based on the type of data the source is. (If the source is a variable, and we are adding, the macro will call the addvar routine.)

Strings are also referenced by number. There are 3 tables for strings:

1. String length table.
2. Max string length table.
3. Pointer table.

So, each string takes up four bytes, plus however long the max string length is. Having the pointer allows the program to point into memory that was never loaded or initialized. This can save time loading the application from disk. The string routines will never overwrite the buffer space allocated for them. The string will be truncated. So, you can append strings without worry about clobbering memory.

Some code using the macros and runtime might look like:

```
              _restore textForBill   ;Point at literal text.
              _readstr billStr       ;Read text into string.
              _prstr                 ;Print the string.
              _hexpad  #'0'          ;Set hex padding to a 0.
              _rtcout  #'$'          ;Print a $.
              _varcpy  bill,val      ;Set bill to val.
              _addl    ,#5           ;Add 5 to bill.
              _mulvar  ,factor       ;Multiply bill by factor.
              _vhexout               ;Print bill in hex.
              rts
textForBill   dc.b     'Bill''s value in hex is:  ',0
```

We could replace the _restore, _readstr, _prstr, with:

```
textForBill   _write   'Bill''s value in hex is:  ',0
```

We could still read the text in the _write macro from elsewhere by:

```
              _restore textForBill+3 ;Point at literal text.
              _readstr billStr       ;Read text into string.
```

The +3 skips the jsr to the write routine (generated by `_write`). Both the `_write` and `_readstr` routines stop at a 0. (The `_readstr` ending character can be changed to whatever you want with the `_readend` macro, but the default stop character is a 0.)

Below is a list of the macros, and how they are used:

`_rtreset`

This macro initializes everything necessary in the runtime. This allows a resume after the user presses a reset.

`_hibitchrs`

This macro is used to turn on the hi-bit for characters that are sent to rtcout.

```
_lowbitchrs
```
This macro is used to turn off the hi-bit for characters that are sent to rtcout.

```
_regchrs
```
This macro is used to make sure that characters sent to rtcout are used as-is. There will be no modification of the hi-bit.

```
_write    &op1,&op2,&op3,...
```
This macro prints a cstring. You can use as many operands as MPW allows. You can continue onto the next line with the line continuation character \. After all the operands are processed, the macro places a cstring 0 terminator character at the end of the string. This is done automatically. Placing one there yourself will define an extra one, which will end up being a BRK instruction in your code.

```
_writecr
```
This macro prints a carriage return.

```
_wrcstr   op
```
This macro prints a c string pointed to by the operand.

```
_rtcout   op
```
This macro prints a character. This character is either already in the acc (no operand), or what is described by the operand. The operand can either be an absolute or a value in memory.

```
_signed
```
This macro sets signed mode. Printing decimal numbers is affected by this.

```
_unsigned
```
This macro sets unsigned mode. Printing decimal numbers is affected by this.

```
_chngsgn
```
This macro does a two's compliment on the variable.

```
_decoutl  op
```
This macro prints a 1-byte decimal value. This value is either already in the acc (no operand), or what is described by the operand. The operand can either be an absolute or a value in memory.

```
_vdecout  op
```

This macro prints a 2-byte decimal value. This value is stored in a variable. The variable number is either already in the xreg (no operand), or is determined by the operand.

```
_decout    op
```
This macro prints a 2-byte decimal value. This value is either already in the acc,y (no operand), or what is described by the operand. The operand can either be an absolute or a value in memory.

```
_hexpad    op
```
This macro sets pad mode for hex. The value is either already in the acc (no operand), or what is described by the operand. The operand can either be an absolute or a value in memory. Printing hex numbers is affected by this.

```
_hexnopad
```
This macro sets no pad mode for hex. Printing hex numbers is affected by this.

```
_hexoutl   op
```
This macro prints a 1-byte hex value. This value is either already in the acc (no operand), or what is described by the operand. The operand can either be an absolute or a value in memory. Since it is only printing a 1-byte value, the maximum amount of padding is one character.

```
_vhexout   op
```
This macro prints a 2-byte hex value. This value is stored in a variable. The variable number is either already in the xreg (no operand), or is determined by the operand. The maximum amount of padding is three characters.

```
_hexout    op
```
This macro prints a 2-byte hex value. This value is either already in the acc,y (no operand), or what is described by the operand. The operand can either be an absolute or a value in memory. The maximum amount of padding is three characters.

```
_addvar    op1,op2
```
This macro adds a variable to the destination variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the source variable number is assumed to be in the yreg.

```
_addl      op1,op2
```
This macro adds a 1-byte value to the destination variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in the acc.

```
_add       op1,op2
```

This macro adds a 2-byte value to the destination variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in acc,y.

```
_subvar   op1,op2
```
This macro subtracts a variable from the destination variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the source variable number is assumed to be in the yreg.

```
_subl     op1,op2
```
This macro subtracts a 1-byte value from the destination variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in the acc.

```
_sub      op1,op2
```
This macro subtracts a 2-byte value from the destination variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in acc,y.

```
_mulvar   op1,op2
```
This macro multiplies the destination variable by a variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the source variable number is assumed to be in the yreg.

```
_mull     op1,op2
```
This macro multiplies the destination variable by a 1-byte value. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in the acc.

```
_mul      op1,op2
```
This macro multiplies the destination variable by a 2-byte value. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in acc,y.

```
_divvar   op1,op2
```
This macro divides the destination variable by a variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the source variable number is assumed to be in the yreg. The remainder from the divide is in the acc,y.

```
_divl     op1,op2
```
This macro divides the destination variable by a 1-byte value. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in the acc. The remainder from the divide is in the acc,y.

```
_div       op1,op2
```

This macro divides the destination variable by a 2-byte value. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in acc,y. The remainder from the divide is in the acc,y.

```
_var       op
```
This macro sets the current variable. The current variable is defined by a number in the xreg. All runtime functions preserve the xreg, so multiple operations can be done to the same variable without having to reload the xreg with the variable number.

```
_set0      op
```
This macro sets a variable to 0. If there is no op1, then the destination variable number is assumed to be in the xreg.

```
_varcpy    op1,op2
```
This macro sets a variable to another variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the source variable number is assumed to be in the yreg.

```
_setl           op1,op2
```
This macro sets a variable to a 1-byte value. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in the acc.

```
_set       op1,op2
```
This macro sets a variable to a 2-byte value. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the value is assumed to be in acc,y.

```
_setvars   op1,op2,op3,op4,...
```
This macro sets multiple variables to constant values. The macro is followed by as many operands as MPW allows. The line continuation character \ can be used. The odd operands are the variables to be set, and the even operands are the values they should be set to. Operand1 gets the value of operand 2, and so on. The macro automatically terminates the list with a 255 byte. This is why 255 is not allowed as a variable number.

```
_minswap   op1,op2
```
This macro swaps the two variables if the xreg variable is bigger than the yreg variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the source variable number is assumed to be in the yreg.

```
_maxswap   op1,op2
```
This macro swaps the two variables if the xreg variable is smaller than the yreg variable. If there is no op1, then the destination variable number is assumed to be in the xreg. If there is no op2, then the source variable number is assumed to be in the yreg.

```
_vsgncmp  op1,op2
```
This macro does a signed compare of two variables. The equal status is true if the variables are equal. If the xreg variable is greater or equal, then the carry is set. If the xreg variable

is smaller, then the carry is clear. If there is no op1, then the variable number is assumed to be in the xreg. If there is no op2, then the variable number is assumed to be in the yreg.

```
_vcmp     op1,op2
```
This macro does an unsigned compare of two variables. The equal status is true if the variables are equal. If the xreg variable is greater or equal, then the carry is set. If the xreg variable is smaller, then the carry is clear. If there is no op1, then the variable number is assumed to be in the xreg. If there is no op2, then the variable number is assumed to be in the yreg.

```
_sgncmp   op1,op2
```
This macro works the same as _vsgncmp, except that it compares a variable to a constant.

```
_cmp op1,op2
```
This macro works the same as _vcmp, except that it compares a variable to a constant.

```
_rndseed  op
```
This macro is used to seed the random number generator. If there is no op1, then the random seed is assumed to be in the acc,y.

```
_random   op
```
This macro is used to return a random number from 0 to op - 1. If there is no op1, then the random number limit is assumed to be in the acc,y.

```
_strval   op
```
This macro takes the value of a string and returns it in the acc,y. If there is no op1, then the string number is assumed to be in the xreg.

```
_midstrval     op1,op2
```
This macro takes the value of op1 string starting at op2 character and returns it in the acc,y. If there is no op1, then the string number is assumed to be in the xreg. If there is no op2, then the character number is assumed to be in the yreg.

```
_prstr    op
```
This macro prints the entire string. If there is no op1, then the string number is assumed to be in the xreg.

```
_prleftstr     op1,op2
```
This macro prints op1 string starting at the first character for op2 characters. If there is no op1, then the string number is assumed to be in the xreg. If there is no op2, then the number of characters is assumed to be in the acc.

```
_prmidstr op1,op2,op3
```

This macro prints op1 string starting at the op2 character for op3 characters. If there is no op1, then the string number is assumed to be in the xreg. If there is no op2, then the character number is assumed to be in the yreg. If there is no op3, then the number of characters is assumed to be in the acc.

```
_leftstrcpy    op1,op2,op3
```

This macro copies op3 characters from op2 string to op1 string. If there is no op1, then the destination string number is assumed to be in the xreg. If there is no op2, then the source string number is assumed to be in the yreg. If there is no op3, then the number of characters is assumed to be in the acc.

```
_strcpy   op1,op2
```

This macro copies op2 string to op1 string. If there is no op1, then the destination string number is assumed to be in the xreg. If there is no op2, then the source string number is assumed to be in the yreg.

```
_midstrcpy op1,op2,op3,op4
```

This macro copies op4 characters, starting at op3 character from op2 string to op1 string. If there is no op1, then the destination string number is assumed to be in the xreg. If there is no op2, then the source string number is assumed to be in the yreg. If there is no op3, then the character number is assumed to be in the acc. If there is no op4, then all characters to the end of the source string will be copied to the destination string. The op4 case is the only case where the assumed value is a particular value (#255), instead of what is in a register. This is the case because there are only three registers.

```
_leftstrcat    op1,op2,op3
```

This macro concatenates op3 characters of op2 string onto op1 string. If there is no op1, then the destination string number is assumed to be in the xreg. If there is no op2, then the source string number is assumed to be in the yreg. If there is no op3, then the number of characters is assumed to be in the acc.

```
_strcat   op1,op2
```

This macro concatenates op2 string onto op1 string. If there is no op1, then the destination string number is assumed to be in the xreg. If there is no op2, then the source string number is assumed to be in the yreg.

```
_midstrcat     op1,op2,op3,op4
```

This macro concatenates op4 characters starting at op3 character from op2 string onto op1 string. If there is no op1, then the destination string number is assumed to be in the xreg. If there is no op2, then the source string number is assumed to be in the yreg. If there is no op3, then the character number is assumed to be in the acc. If there is no op4, then all characters to the end of the source string will be concatenated to the destination string. The op4 case is the only case where the assumed value is a particular value (#255), instead of what is in a register. This is the

case because there are only three registers.

```
_litstr   op1,op2,op3,...
```

This macro works very much like _write, except that the characters are not printed. They are copied into the designated string. If there is no designated string, then it is assumed that the xreg already holds the string number.

```
_strchr   op1,op2
```

This macro returns the op2th character of op1 string. If there is no op1, then the destination string number is assumed to be in the xreg. If there is no op2, then the character number is assumed to be in the acc.

```
_strloc   op1
```

This macro returns the physical location of op1 string in memory. The string location is returned in acc,y.If there is no op1, then the destination string number is assumed to be in the xreg.

```
_cstr     op1,op2,op3,...
```

This macro is used to generate a cstring using the same syntax as other macros. It works very much like _write and _litstr in the way it handles parameters. The only thing that this macro does, however, is define data.

```
_restore  op
```

This macro sets the read data pointer. If there is no op, then the address for reading data is assumed to be in the acc,y.

```
_readint  op
```

This macro reads an int from the current data pointer and advances the pointer by two bytes. If there is no op1, then the destination variable number is assumed to be in the xreg.

```
_readstr  op
```

This macro reads string data into the designated string until the end-of-string character is encountered. The data pointer is then set to point after this end-of-string character. If there is no op1, then the destination string number is assumed to be in the xreg.

```
_readend  op
```

This macro is used to set the end-of-string character for `_readstr`. If there is no op1, then the `_readstr` ending character is assumed to be in the acc.

The array handling is simple pointer-based storage and retrieval. There is also some logic for multiple-dimensioned arrays. This system can handle arrays up to four dimensions deep. With a little work, this can be expanded to any number. The system allows for easy expansion to more dimensions. It works by having the application first give a description of the array. This description consists of the location of the array, the size of the elements in the array (byte or word), and what size the dimensions are. There are actually four pointers into the array area.

The first one is always the base address. When the base address is first established, all four pointers are set to the base address. If you then wish to index into the array, you tell the runtime at what point in the array you wish to access for

each level of subscript. If a subscript is omitted at the beginning of the index list, then it is assumed that the pointer for that level is already what you want. If a subscript is omitted after the first subscript, then that subscript is assumed to be 0.

For example:

We have a four dimensional array of words, called things, equated to address $4000. We would declare the array with the `_array` macro as follows:

```
_array    #things,w,#3,#4,#5,#6
```

This line means "use the memory at #things as an array of words that is 3 by 4 by 5 by 6 in size." The array handling routines now use this information when getting and putting data to the array. The routines only keep track of one array. If you wish to switch between two or more arrays, you will have to reissue `_array` commands. The `_array` command does not allocate or protect any memory. It just gets the relevant information for the array. Thus, you can switch arrays as often as you want. Of course, there is some time overhead in switching, but it is rather minimal.

If we wished to get the element `things(1,3,4,5)`, and place it in the variable `thing`, we would do the following:

```
_index    #1,#3,#4
_getw     thing,#5
```

The `_index` macro builds pointers for the array into each level of subscript. The `_getw` macro indexes (using #5) from the lowest level pointer, and gets that word and stores it in the variable `thing`. If we then wish to get the element `things(1,3,4,4)`, we would:

```
_getw     thing,#4
```

We don't have to recalculate any pointers. They are still intact from the above _index macro. If we wish to change any of the subscripts, other than the last one, we will have to issue another `_index` macro. Lets say that we want the element `things(1,2,0,5)`. Knowing that the previous `_index` macro got the element `things(1,3,4,5)`, we can minimize the pointer math by having the `_index` macro redo only the data that is no longer valid. We would do an `_index` macro as follows:

```
_index    ,#2
_getw     thing,#5
```

The first subscript is the same. Therefore, the pointer for the first subscript is correct. The pointer for the second subscript is generated from the first (just as the pointer for the first is generated from the base address of the array). So, we don't have to mention the first subscript again. Not doing so saves code. Once any subscript pointer is recalculated, all the lower level pointers are set to that value. This means that there is an assumption that all the remaining subscripts are zero. If this is indeed the case, they don't have to be mentioned either. In this example, the third subscript is zero. So, it doesn't have to be mentioned either. This method of separate pointers for different levels of the array can be very efficient. It also can be confusing. If it is not clear whether you can leave subscripts out, don't. It will still work with all of the subscripts, although it will be more code than necessary.

There is another form of the _index macro. It is the _vindex macro. This is used when the index value is stored in a variable. One unfortunate syntax limitation is that you can't mix the two types on one line. You still can mix the two types, however, when indexing into an array. For example:

```
bob = things(color, 3, type, 5)
```

would be done like:

```
_vindex   color
_index    ,#3
_vindex   ,,type
_getw     bob,#5
```

The following macros are used to define and access arrays.

```
_array    loc,elesize,op1,op2,op3,op4
```

This macro prepares the array handling routines to work on the defined array. The operand loc is either the address of the array (when preceded by a # sign), or a 2-byte location where the address to the array is stored (when preceded by an *). The operand elesize determines the size of the elements in the array. The elements can be either bytes or words. The operand elesize should be either a b or a w. Operands op1 through op4 describe the dimensions of the array. These must be constants. Operands op2 through op4 are optional. The array may only be linear, in which case, op2 through op4 would have no function. The operand op1 is actually not used. It is just for commenting purposes. The array routines do not do any range checking, and all that op1 is good for is to determine if the first subscript is out of range. You must have an op1, however, as a placeholder. This is to encourage it being used, as it is useful when reading the source code.

```
_index    op1,op2,op3
```

This macro is used to calculate a pointer into the array. There are actually four pointers. The first three are used as partial results from previous calculations, so that all subscripts don't have to be involved every time there is a calculation to determine the specific element of the array. When the pointer for op1 is calculated, the resulting offset is added to the base array pointer (defined with the _array macro). This is then stored as the 2nd, 3rd, and 4th level pointers. If there are no more operands, then the effective pointer into the array will be this value. If op2 and op3 were zero, then this value would be correct. The assumption in this case is that any undeclared subscripts beyond the first one declared are zero. So, if your array is only two-dimensional, you would then never declare an op2 or op3, and they would therefore never affect the pointer into the array. An interesting aside to this is that one array can actually be considered to have a variable number of dimensions. Let's say that you have an array that is 10x10x10. This array could also be viewed as a 10x100 array. Both have the same number of elements. Using the _index macro differently, you could access the array either way. For example:

```
_index    #5
_getw     value,#55
```

would work as well as

```
_index    #5,#5
```

```
      _getw     value,#5
```
This flexibility can be very useful. (It can also be confusing.) If you have a case where you know that the first subscript is the same as the last time you used the `_index` (or `_vindex`) macro, you can choose not to restate that subscript. You might even have a

situation where the second subscript is the same also. In these cases, you can choose to leave those operand fields empty. This will produce less code than if you mentioned them. (There is an overhead of 7 bytes per index with the `_index` macro, and an overhead of 5 bytes with the `_vindex` macro.)
The syntax for this would look like: `_index ,,#5`
The commas are necessary to indicate that subscripts 1 and 2 are already defined, and that you are referring to subscript 3. If the commas were not there, the `_index` macro would assume that the `#5` is the first subscript, and that subscripts 2 and 3 are zero, or not used at all.

```
_vindex   op1,op2,op3
```
This macro works just like `_index`, except that the operands are variables, instead of constants or values at a particular memory location. This is useful when you want to use variable subscripts.

```
_getb     op1,op2
```
This macro is used to get a 1-byte value from the array. The operand `op2` is the right-most subscript in the array. This value can be either a constant, or a value in memory. The operand `op1` is the variable to store the array value into. The low byte of the variable will be set to the value, and the hi-byte will be set to zero.

```
_vgetb    op1,op2
```
This macro works just like `_getb`, except that the operand `op2` is a variable.

```
_getw     op1,op2
```
This macro works just like `_getb`, except that the value from the array is a word, instead of a byte.

```
_vgetw    op1,op2
```
This macro works just like `_getw`, except that the operand `op2` is a variable.

```
_putb     op1,op2
```
This macro places the low byte of the variable `op1` into the array at the point described by the right-most subscript `op2`. The operand `op2` is either a constant, or a location in memory where the subscript is stored.

```
_vputb    op1,op2
```
This macro works just like `_getb`, except that the operand `op2` is a variable.

```
_putw     op1,op2
```
This macro places the variable's value into the array at the point described by the right-most

subscript `op2`. The operand `op2` is either a constant, or a location in memory where the subscript is stored.

```
_vputw    op1,op2
```

This macro works just like `_getw`, except that the operand `op2` is a variable.