Data Memory Usage in Basic 2.4

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

1.0 Introduction

The dual memory (Control and Data) of the Wang 2200 series of computers enhances the speed capabilities of the system. The architecture of the 2200 system precludes the usage of Control memory for storage of temporary data. The Control memory is too awkward to fetch data for use in atomization, character comparisons and math constants.

Control memory is used primarily for instructions, and for some limited message storage. Data memory contains all the pointers and tables required for word comparisons, both for atomization and normal atom enhancers, as well as containing the math constants required to produce LOGS, conversions and Trigonometric functions.

2.0 Basic Overview

In previous documents, we established that Data memory can range in size from 4096 bytes to 512k bytes. Since any address over 65536 requires us to use bank selection techniques, Wang has formed a hardware solution that allows us to access the first 8192 bytes of memory from any bank. Any address below hex 2000 will automatically revert to Bank 0. It is in this area that three most important areas lay, Constant storage, Partition Status/Control and the Universal Global Partition.

In general, the following breakdown may be established:

1: Constant Storage	0000 - 08FF
2: Partition Status	0900 - 0BFF
3: Universal Partition	0C00 - 1FFF

Partition status and Control is described in depth in 'Common Partition Control', and therefore will jot be discussed here.

3.0 Constant Storage

The appendix of this document contains the listing of Data Memory for Basic 2.4. However, since all previous Data Memory areas are similar, the reader should have no problem reverting back to other versions.

3.1 Checksums

There are two levels of checksums in the Wang 2200 computer. The first is in Control memory, at each 4k word boundary, while the second one is for data memory. This checksum, in theory, is to catch any double bit errors that may have occurred. Remember that single bit errors would have caused a hardware vector (PEDM). Double bit errors are not detected in this manner. However, when either a load operation is performed, or the RESET key is depressed, a checksum is performed in Data Memory.

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

Since this checksum is loaded at the same time as the file, we cannot dynamically recalculate a checksum if Data Memory has been modified. In essence, once this Constant storage area has been loaded, no modification of Data Memory covered by the checksum can occur.

The location of the checksum is not the same for all versions of Basic. The first two bytes of Data Memory, at location 0000, point to the location of the actual checksum. But the checksum used by Wang is actually thirty-two (32) bits wide, where one 16 bit word signifies the exclusive oring of all bits, while the other is a shifted result. Subtracting two from the contents of location 0000 points us to the actual checksum location.

This location is the last location in data memory that Wang considers 'sacred', and any modification by us prior to this pointed to value will result in an VEDM error. When changing Data memory, we will use the program PATCHER to recalculate the new Data Memory Checksums.

When first experimenting with data memory through the use of our utilities, DEBUG, etc, we can prevent the checking of Checksums in Data memory by clearing the pointer to the checksums. Setting location 0000 to 0000 causes Wang to ignore what we are doing, allowing us to manipulate bytes without fear of aborting out.

The current version of Basic 2.4 locates has location 0000 pointing to 08E2. The actual checksum locations are therefore 08E0, 08E1, 08E2 and 08E3.

The calculation of the checksum is not difficult. Remember that location 0000, the pointer to the checksum, is also included in the checksum. Initially, two 16 bit registers are cleared, called Cksum1 and Cksum2, then the formula is:

```
For N = 0 to (Contents of 0000 - 2) step 2

     Cksum1 = Cksum1 XOR (16 bits location N)
     Cksum2 = (Cksum2 ADDC Cksum2 ADDC Overflow) ADDC
                  (16 bits location N)

Next N
```

This forms the correct XORed data, to be placed at the location pointed to by 0000 - 2. Now we must include this result in the final 16 bit checksum.

```
Cksum2 = (Cksum2 ADDC Cksum2 ADDC Overflow) ADDC
             Cksum1
```

Cksum2 is now inserted in the memory location pointed to by location 0000.

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

3.2 Atomization

Starting at location 0002 in Data Memory, (DM), are two byte pointers to atomization lists used during Pass 0 of Basic 2.x. In general, I have broken down the lists as follows:

0002	ALIST1	Left hand side ATOM list
0004	ALIST2	Immediate Mode only ATOM list
0006	ALIST3	Complex Math Atoms
0008	0000	End of first list
000A	ALIST4	Trig function chain
000C	ALIST5	Right hand side Numerics
000E	0000	End of second list

When a potential word is attempted to be atomized, These lists are used to point to lists containing the verbage for the potential atom, along with the atom itself. As an example, take list 'ALIST3. Currently pointing to 07BD in Data memory, we find the following:

```
07BD 0D          ALIST3    FSB       $0D
07BE 04          ABS(      FSB       COS(-2-.
07BF 41425328              FST       'ABS('
07C3 C1                    FSB       $C1
07C4 04          COS(      FSB       EXP(-2-.
```

The first byte in the chain list always tells us how many entries there are in this list. ALIST3 is therefore $0D, or 13 entries in length. The process continues now, atom by atom attempting to make a match. The first atom to be tested is ABS(. The first byte of each atom entry describes the length of the atom, in the case of ABS(, that length is 4. Wang will attempt to match the word in Data memory with ABS(.

If not successful, Wang decrements the count of atoms in that list, and if non-zero, proceeds with the next atom in the list. If successful, the next byte, in the case of ABS(, C1, is used as an atom. Note that if this value is 00, no atom is present, and this may be just a 'reserved word'.

No fancy hashing techniques are employed here. It is just plain brute force comparisons that resolve a word to an atom. However, since this process of converting words to atom is only done once during resolution phase, we dramatically reduce the required memory requirements for storage.

Note that is quite easy to change an word to an atom if we reduce the size of the word, but difficult if we try to increase the size. However, we can balance out by decreasing one while increasing another.

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

3.3 ATOM processing

After the verb is processed to a token or atom, the process of reverse expansion is needed to print out the verbage during list sequences. Furthermore, if we have a left hand side atom, we must somehow vector to a routine in Control Memory to process that atom.

Location 0100 of memory is the start of the vector table for atoms, and significant data is stored there. For purposes of discussion, the first four entries are reproduced below:

```
0100 071F FDB  LIST
0102 44C7
0104 24B7 FDB  TYPE2!CLEAR
0106 1C00
0108 24B2 FDB  TYPE2!RUN
010A 1C1D
010C 24BE FDB  TYPE2!RENUMBER
010E 480F
```

Let us suppose that while processing, the atom 80 is encountered on the left hand side. The program in control memory performs the following calculations:

```
((Atom - 80)*4)+0100
```

This gives us the Base address in the vector table. In this case, we arrive at 0100 as our result. The four bytes at this location are pertinent to the atom LIST, therefore, atom 80 is the LIST atom. The first two bytes, 071F, point to a location in Data memory that contains the length and verbage LIST. The next two bytes, 44C7, is the vector in Control Memory to goto. Location 44C7 is thus the start of the routine to analyze the LIST atom.

It is the function of that routine to further analyze any more atoms on the line, or find whatever arguements it requires.

The next atom in our example is the CLEAR atom. Note that the verbage TYPE2 has been ored into the verbage identifier for CLEAR. The upper nibble of the first 16 bits always tells Basic what exact type of atom this is. We mask this data out to find where in Data memory the actual verbage CLEAR is located. CLEAR's verbage is located at:

```
Location for verbage = Hex(0FFF) AND 24B7,
                                 Or  04B7
```

Therefore, we would expect to find at location 04B7, the length of the verbage CLEAR, 5, followed by the word CLEAR. The vector for CLEAR is 1C00.

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

The functions defined by the high order nibble are as follows:

TYPE 0	*Stand alone, functions processed by routine*
TYPE 2	*If in first position, must be Immediate Mode only*
TYPE 4	*Right hand side, require numeric argument and).*
TYPE 6	*Peripheral Modifiers, ie, DISK,P,G TEMP*
TYPE 8	*Functions of another function, ARC*
TYPE A	*Numeric Functions*
TYPE F	*Cannot be stand alone, modifiers to another atom.*

The vectors in the second 16 bit word may on occasion be zero. This signifies that it cannot be an executable atom. (TEMP)

If an atom has a vector on the left hand side, we can easily intercept that vector to add a function. All we have to do is change the initial vector to one that we wish to go to. We then can test our function, and if not present, return to the original function. If it is our function, we would process the atom according to our whims.

As an example of this, we modified the routine #ID to search for the atom CLOCK, which we implemented. We had modified BACKSPACE atom, being one of the two spares, to be CLOCK.

The normal vector for # was 1684. We could change this to our routine, and check for the atom CLOCK. This would allow us to have #CLOCK as a function. However, we let it process to the point where it found the ID verb, and changed the vector at that point to point to our routine. We then checked for CLOCK, and either process our atom, or goto the correct #ID routine.

3.4 List Processing

Not all of the functions within Wang Basic can be compressed into the Atom format. For this reason, Wang will vector to a routine when it processes the main token, which in turn, will attempt to find the next verbs that match. These verbs may or may not be atoms. Examples of this type of search is $GIO, #ID, MATSEARCH, etc.

The routine in main Control memory would look something like this:

```
1000 LPI   PARESE$LIST
1001 JSR   SEARCHLIST
1002 JMP   IF
1003 JMP   GIO
1004 JMP   TRAN
      .
      .
1010 JMP   BADLIST
```

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

Note that the address of the list is placed in the PHPL registers, and a call to the SEARCHLIST routine is performed. SEARCHLIST will then use the PHPL pair as a POINTER to a list of POINTERS, which in turn point to the length and Verbage of the following datum.

If the verbage in Data memory does not match the program, the stack, which contained the return address from the JSR, is popped, and incremented, then restored onto the stack. If a match is found, a return is executed, which return control back to the argument list, resulting in a vector to the correct routine.

In our example, location 1001 is the JSR, so 1002 gets pushed to the stack. If we had the word GIO following the $ symbol, the routine at first attempts to match it with IF. Since IF does not match, the stack is popped, incremented to 1003, and pushed back to the stack. The next attempt matches with GIO, so a Return is executed, returning us back to the location pointed to by the top of the stack, 1003. 1003 is a JMP to the GIO routine ....

If no match has been found, the system would run through the list and find a terminating pointer word of 0000. This would cause a return, normally to a JMP BADLIST, to produce an error message.

Again, we can intercept this JMP to one of our locations, and possibly process our own atoms. It is very difficult, without reassembling the entire @@ file to expand this chain list. It is easier to modify the JMP, and process somewhere else.

The Parse lists are conveniently located in Data Memory. The following Parse List locations contain the POINTER to the LIST of POINTERS, and though the actual addresses of the LIST of POINTERS may change from version to version, these addresses always have remained constant:

00F0	PARSE$LIST	$ atom list
00F2	PARSEMATLIST	MAT atom list
00F4	PARSEMAT1LIST	MAT (Matrix Math) parse list
00F6	PARSESELECT	SELECT atom list
00F8	PARSEPRINT	PRINT sublist (PRINT AT)
00FA	PARSEMATH	Boolean math AND,OR ...
00FC	PARSEVELD	File type modifiers (BA,BT,DC)
00FE	PARSE#LIST	# atom lists

The program ATOMLIST will display the contents of these lists when run.

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

3.5 Message Storage

Various often used messages are used throughout Basic 2.x are stored in Data memory. Below is a list of messages:

0012	ENDMSG	Displays 'END PROGRAM'
001E	FREEMSG	Displays 'FREE SPACE ='
002A	TXFRMSG	Trace mode, displays 'TRANSFER TO'
0036	ERRMSG	Dual function. Displays ERR when used by the error routine, such as ERR 34. SECTOR is used during VERIFY error
0042	MUXEMSG	Displays 'Error xx loading Terminal Controller' if no @MXE0 file exists.
08C4	RSTMSG	Displays this on line one when the RESET key is depressed.
08E4	PASS:1	Password used for verification of $INIT statement. Note that this is just outside of the checksummed area.

Earlier versions of Basic contained the Catalogue messages in the front of Data Memory. Basic 2.4 added a significant amount of verbage for SCREEN, PASSWORD, DATE, TIME and DISCONNECT, forcing Wang to move the catalogue messages to Control Memory.

3.6 Constants

Quite a few math constants are located in Data memory. The area between 0300 and 04A5 is referred to as the Constant storage area. I have not studied the area enough to be absolutely sure as to the contents. However, the following small areas are defined:

0300	Constant for PI	3.141592653590
0308	Natural Log 10	2.30258509299404
031C	Radians to Grads	R x 63.66197723675
0326	Radians to Degrees	R x 57.29577951308..
0330	Degrees to Grads	D x 1.111111111111..
0344	Grads to Grads	G x 1.000000000000..
034E	Grads to Radians	G x .0157079632679489

The other numbers have not been studied, but are, I suppose, part of the equations for Trig functions, as well as Square roots.

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation

3.7 Default Values

Several default values are retained in Data memory. I cannot fathom why they wasted the space here, but Console, Tape and some other defaults are listed here:

04A6	0001	Default Console Input device
04A8	0413	Default PLOT device
04AA	0000	Default TAPE device
04AC	0310	Default Disk Device
04AE	0005	Default Console Output Device
04B0	50	Default console width (80 characters)
0010	9602	Constant used during Disk wait for Ready delays.
08DE	4D24	System type, 4D = M for MVP, while 24 is revision 2.4

3.8 Random Numbers

The random number generator used by Wang is used in two places. One is internal to the partition control area, the other is a 'global' register.

If executing the RND(0) statement, the Random number generators' seed is stored in your partition control block. If a RND(0) has never been issued following a RESET condition, the systems random number seed is used. This seed is stored at four locations immediately after the password. These locations are 08EC through 08EF.

When Control Memory is first loaded, the seed is initialized to the same exact constant, 00002001, as when you execute a RND(0) function within your own partition. This seed is manipulated during partition switching time. Since this seed is always changing, it is more of a 'Random' number that that of an internal partition Random number.

Copyright @ 1983 by Computer Concepts Corporation, Shawnee Mission, Ks
No part of this document may be reproduced without the expressed written permission of Computer Concepts Corporation