

A mystery solved – how the PBUFF printer buffer project works

Don McKenzie

In the March 1987 issue, we published Don's 256K printer buffer as a Star Project. It has been quite a popular project, which Don markets as a short form kit, and the article proved of interest to a great many readers. However, a number of readers were interested in Don's circuit as no memory decoding or bank switching is used, seemingly "breaking the rules". Don tells how it's done.

SOME TIME after publication of the PBUFF Printer Buffer project, a reader, L. McKeon, wrote to AEM's Editor, Roger Harrison, suggesting that details on precisely how the circuit worked. Several other readers also wrote to the magazine at different times. Roger passed Mr McKeon's letter on to me with the suggestion that I may wish to reply in a format suitable for publication as an article. But before doing that, let me bring you briefly up to date with PBUFF.

I have made a number of changes to the physical design of the project since it was published. The double-sided board is now supplied with plated-through holes. The circuit is virtually the same except for 74LS04 and EPROM pinout. Existing input and output signals, that is, 12 in line at 2.3 inch centres have been retained for compatibility with existing optional add-on boards. These two connectors are now known as J1 and J2. A new connector has been added. This is set out in a 26 pin standard header pattern and is J3. A dual-row male unprotected header can be soldered into this position. A flat ribbon cable is used for board inter-connections. An eight inch length of cable must have a 26 pin female IDC type crimp header crimped onto each end. Boards now simply jumper together with this cable. Provision has been made on the board for an 8K static RAM (6264). However, current versions of software do not use it. This is for future development.

Mr McKeon, from Nundah in Queensland, wrote: "I'd like to see published a 'How it works' piece on Don McKenzie's PBUFF printer buffer from the March (1987) issue of AEM. He uses a 16K ROM and 256K RAM on a bog standard Z80A, but I can't see any components used for memory decoding or bank switching."

A little knowledge

It's surprising what can be done with an elementary micro system and a little bit of knowledge. As an example, let's pose a question for readers well-acquainted with the workings of the Z80 CPU. Apart from a minor aspect, this doesn't have a great deal to do with PBUFF, but it does demonstrate the type of thinking required to manipulate a CPU in a mode not detailed in the manuals:

How many port addresses can be accessed by the Z80?

I would think that 95% of all knowledgeable answers would be 256, since that's the number the reference manuals give. The 256 ports are decoded using the low order address bus (A0-A7) validated with the port select signal (IORQ). This combination gives the chip select signal for any port mapped device. The CPU also issues a read or write signal to inform the port device as to the direction of data flow on the data bus. The port device

responds by transmitting or receiving data during the chip select condition.

Let's explore the OUT (C),A instruction. The IN equivalent operates in the same manner. The contents of register C are placed on the bottom half of the address bus (A0 through A7) to select the I/O device at one of 256 possible ports. The contents of Register B are placed on the top half (A8 through A15) of the address bus at this time. Then the byte contained in Register A is placed on the data bus and written into the selected peripheral device. The BC Register pair can be loaded with any value from 0000 Hex to FFFFH. The contents of the BC Register pair appears on the full address bus during the OUT instruction. These sixteen address lines can be hardware decoded to 65,536 discrete I/O ports. The software only needs to set up the BC Register pair prior to the OUT (C),A instruction to select one of these 64K ports. This is 256 times the ports as stated in the manuals. The only restriction with the expanded port configuration is the need for preservation of the standard hardware ports of an existing system.

All port instructions may need revamping. For instance, with the OUT (C),A instruction, the B Register must be set to zero before true (half decoded) ports 0 to FFH are selected. Looking at the IN A,(n) instruction, the contents of the A Register appear on the top half of the address bus. (A8 through A15) This will result in random selection of the 64K ports dependent on the contents of the A Register. These types of instructions will have to be removed and re-written with the dual Register pair types of code. Not very practical for your garden variety computer that probably has BASIC in ROM and can't be changed readily, but if you are building a system from the ground up, and need 346 ports then this bit of knowledge may assist you greatly. If you don't need 346 ports on your computer then erase your memory banks on this thesis for further data.

Copyright

Let me fill you in on what has taken place during PBUFF development. In July 86, I was approached by the Production Manager of Ectron Pty Ltd in Melbourne. They had an existing ▶

For readers interested in the PBUFF project who haven't got the March 1987 issue, back copies (or photostats if copies have run out) are obtainable for \$4.00 post paid from AEM Back Issues, PO Box 507, Wahroonga 2076 NSW.

The PBUFF short form kit can be obtained for \$39 plus \$3.00 postage from Don McKenzie, 29 Ellesmere Crescent, Tullamarine 3043 Vic. If you need further information on just send Don a stamped (37¢) self-addressed post office preferred size envelope (235 mm by 120 mm) and he will forward information on.



64K printer buffer, and wished to develop a 256K version. I signed an agreement with them to supply software to enable them to easily convert their 64K buffer into a 256K buffer. I not only supplied my own 256K read/write routines, but integrated them into ECTRON's existing 64K software. The terms and conditions of this contract were met in July 86. This new buffer is known as the "Ectron Model 2256 Data Spooler". As a result of this agreement, ECTRON now have sole and unrestricted copyright on the software that I supplied to them.

One reservation I had regarding this agreement was the fact that I still had a lot more development work that I wished to carry out on the buffer, and I wanted to keep marketing the unit in kit form. Ectron's main interest is in the business market, and mine more aimed at the hobbyist. As agreed to by both parties, the following clause was added to the contract: "Ectron agrees, however, that Don can still continue to supply his original software in EPROM's as part of his Printer Buffer Kit, which is aimed at the computer hobbyist market." The other condition that I imposed on ECTRON was that they could only use my supplied software in a universal type (stand alone) buffer. This leaves me clear to produce a slot or card unit suitable for a specific computer, i.e.: IBM PC or Apple type installations. So although I don't hold Copyright on my own 256K read/write routines, I am still able to use them for myself in kit form, and slot (computer specific) form only.

Thus, I am not able to supply to other third party enterprises source code listings or additional software information. I can't let the cat out of the bag. I now have the interests of Ectron to protect. The Ectron buffer is a professional unit in a superb metal case, aimed at the business market, ready to plug in and go, but it costs just a little more than building your own PBUFF. Please note - my PBUFF EPROM will not function in the Ectron spooler. The EPROM is not mapped in the same territory, and the Ectron's front panel controls operate in a vastly different manner.

The situation with software copyright will change slightly with my new software for the Rev.F PBUFF board. This has provision for 8K ROM and 8K static RAM on board. This will enable the user to download a machine language program from a disk file into the buffer and re-program the buffer to suit individual requirements. The read/write routines have been re-written to a new formula. However I will not release source code listings on this information. I do see the need for entry points documented for ROM calls to enable programmers access to PBUFF.

Considering the copyright restrictions, I will now attempt to explain "how it works"

Here's how it's done

The main concept with PBUFF is to dramatically reduce the hardware required to produce a microprocessor controlled printer buffer with a large capacity, and give it a few bells and whistles without additional hardware. No address or data buffers are used or required, providing the CPU is not overloaded. All CPU signals must be electrically as short as possible for reliable operation.

The only port-addressed device is the 8255 PPI (programmable peripheral interface). This is the interface to the computer input and printer output bus. The Z80A's IORQ line is connected to the chip select of the 8255. Any time an I/O instruction is executed the 8255 responds. Any port from zero to 255 is the 8255's port decode. The power-up routines program the 8255 to mode 1, Port A input, Port B output, and Port C handshaking for A and B. Capacitor C5 hanging on the chip select line to ground produces enough of a delay to satisfy the timing restrictions of Z80A to 8255 interfacing.

The printer output strobe is produced by inverting CPU signal A13. One gate of the 74LS04 is used for this. Any time the program sets A13 high, a printer strobe is output. The setting of A13 is actually done with a BC register pair port instruction, not a memory address. This technique of high order address

bus port selection has been explained earlier in this article. During a power-up or reset of the CPU, all address and data bus signals go to a high impedance state. The high to A13 appears as a printer output strobe every time the buffer is reset. This is where resistor R10 is useful. Without R10 installed some printers will hang up. They detect the output printer strobe during reset, read the data and issue an acknowledge (ACK) to the buffer. The buffer doesn't pay much attention to this ACK because it has just had a reset and wants to do its initialization routines. Result, the printer goes to sleep. Not all printers, just some, like Star Geminis. With R10 installed, during a reset, instead of A13 tri-stating, it is held low by R10 to ground and this spurious strobe isn't produced.

Pin 12 of the 8255 is the input BUSY to the computer. This signal is inverted by the 74LS04 to produce the ACK signal. This means that while the 8255 is holding the ACK active (low) it sends the BUSY signal to the computer. The ACK is also connected to R8 and the data LED. The LED lights up when the BUSY is high and ACK is low, and switches off when BUSY is low and ACK is high. This LED is used for the ROM diagnostic routines to aid kit builders, and also indicates data entry into the buffer. When the buffer is full, the LED lights up brightly to indicate a full BUSY condition. The value of R10 is a compromise between lighting the LED up brightly enough to see data activity, and not loading the ACK line to the computer excessively.

The reset of the 8255 is a positive-going pulse, which is inverse to that for the CPU. One gate of the 74LS04 is used to generate this pulse from the CPU reset.

EPROM pinout differs slightly between the Rev. A to E boards, and the Rev.F board. The early version uses the A11 pin of the EPROM as a one-of-two software selection method by switching from lower to upper half of the EPROM. The later version uses A12 by either connecting it to the true A12 of the CPU giving a full 8K ROM if needed, or pulling it to +5 volts to select the upper bank.

The EPROMs basically have two chip enables. Ignoring the top four pins in a 28-pin socket and counting the pin numbers as you would in a 2732 package, pin 18 and 20 are the chip selects. Pin 18 connects to the MREQ signal of the CPU. Any memory referenced instruction will select the EPROM, but no data output will take place until you also chip select pin 20. Pin 20 is connected to A15 of the CPU. Any time A15 is low and memory is selected the EPROM is enabled. This gives an effective address for the EPROM from zero to 7FFFH. A13 is already dedicated to the output strobe, so a maximum size of 8K of EPROM can be used. The existing versions of software make use of around 400 hex bytes. I use a larger EPROM than needed only because the larger ones are now more readily available and cheaper.

The dynamic RAMs are enabled when A14 and A15 are high. This gives a starting address of C000H. A15 high disables the EPROM. The addressing and refreshing of 64K Drams in this configuration is not new. Several articles have been written on the subject, even Elektor Electronics U.K. April 86, had an article on this subject. The Z80A takes care of refreshing of the DRAMs in the normal refresh register mode. The data in the refresh counter is sent on the lower portion of the address bus (A0-A6) along with a refresh control signal while the CPU is decoding and executing the fetched instruction. This mode is totally transparent to the programmer and does not slow down the CPU operation. This caters for DRAMs with 7-bit refresh, which to my knowledge covers all 4164s except Texas Instruments types.

The TI DRAMs and all 41256s require an 8-bit refresh. To do this, data bit 7 in the refresh register must be toggled to correspond with the 8-bit format. The program must constantly service this refresh requirement. This involves calling a refresh routine approximately every 256 instructions. This could be done with hardware, but involves a lot more real estate and conflicts with my concept of a simple cheap buffer. ▽

The machine language code to write to a 64K RAM location looks something like this:

Assume C register contains the byte to be written.

Assume HL register pair points to the storage memory location.

```
LD    A,R      ;Get refresh address.
EX    AF,AF'   ;Save refresh.
LD    B,H      ;Save MSB current address for
                later restore.
LD    A,H      ;Put MSB into A reg.
LD    H,0C0H   ;Select MSB of Dram enable circuit.
LD    R,A      ;Move MSB current address
                to refresh reg.
LD    (HL),C   ;Write byte.
EX    AF,AF'   ;Restore old refresh address.
LD    H,B      ;Restore MSB current address.
```

The R register does an auto-increment every instruction fetch. To keep track of this register, a save and restore is performed, as every read or write destroys the contents of the R reg. The R register must be preserved for correct DRAM refreshing and bit 7 toggling.

Now what really took place? The LD (HL),C wrote the byte, but where? The byte jumped into the memory matrix that was addressed with the MSB of the address pointed to by the R register, and the LSB pointed to by the L register. The chip select of the DRAMs was enabled by the C0H contents of the H register. You could have loaded the H register with FFH to do the same thing. As long as A15 and A14 are high, DRAM is enabled and ROM is disabled. But if you did load the H register with an FFH, A13 would go high and a strobe would be output to the printer, so settle for a C0H.

So the DRAMs aren't really part of a typical memory map, it's software deception. The true map of the DRAMs as far as the CPU addressing is concerned is C000H to C0FFH. Only 256 addresses. H register set to C0H and L register set to LSB select each byte. The MSB of the DRAMs is dependent on the contents of the R register during a read or write procedure.

Think of the 256 locations pointed to by L register as cells rather than bytes of information, and think of the variable parameter (0-255) that the R register outputs, as a bank select. This gives 256 banks of 256 cells (8 bits wide), a total of 65,536 bytes of data. The cell addresses are at C000H to C0FFH, and the bank select address is loaded into the R register prior to any cell read or write.

What we are left with is a lump of DRAM that is not much good for anything except storing great chunks of data. You can't run a machine language program in it. It's a complete wrap around buffer. When the storage pointer gets to FFFFH, its next location is zero, and round'n'round it goes.

To access the full 64K of these types of dynamic RAM, The input address lines are multiplexed. The bank select is equivalent to a row address strobe, (RAS*) and the cell select is the column address strobe (CAS*). RAS* and CAS* timing is generated using the 74LS00 as follows (the * following a signal name means it is active low):

RAS* generation: CPU signals M1* and MREQ* are low during every CPU instruction fetch cycle. This keeps the output of the flip flop (E9 gate a, and E9 gate c) and thus RAS*, high. After this fetch cycle, the contents of the R register are placed on A0-A7, and the CPU starts an execution cycle with MREQ* going high, followed by M1*. Only when MREQ* goes low again will the flip flop toggle to activate the RAS* lines, which will remain low until the next CPU fetch cycle.

CAS* generation: This signal will be active during every CPU read or write cycle with A14 high (Dram enable) and A15 high. (EPROM disable) The WR* line is generated from the RD* signal using one gate of the 74LS04.

8K to 64KRAM operation: With the 8K to 64K version of software (V2.2), a single 4164 can be used for an 8K buffer. The power up routines check memory size, and select the appropriate

read/write drivers. An 8K buffer for instance will get a byte from your computer, and place it in bank 0, cells 0 to 7, as a single chip is only one bit wide (65,536 by 1). When retrieving this byte, it must gather up information from cells 0 to 7. A 64K buffer writes the full eight bits (one byte) at a time. Bank 0 is selected and cell 0, then cell 1, up to cell 255. When the first cell array is full, bank 1 is selected, and so on. The software checks for buffer full condition so that no overflow onto existing data can occur.

256KRAM operation: The full eight DRAMs (41256) must be installed for Version 3.2 PBUFF. We now introduce a page selection mode. Pin 1 of the CPU is connected to pin 1 of the Drams. This is CPU A11, and the additional address pin of the DRAMs. I could have used any CPU address line that wasn't dedicated for device decoding, but pin 1 to pin 1 has an unbelievable ring to it. This decodes the cells into CPU addresses C000H to C0FFH, and C800H to C8FFH. Every address pin you add to this type of DRAM expands it's total memory capacity by four times. The software sets up pages 0, 1, 2, and 3. The first byte written is to page 0, bank 0, cell 0, then to P1, B0, C0. The full 4 pages are filled before the next cell is selected. Cell addresses at C000H are used to select pages 0 and 2, and cell addresses at C800H for pages 1 and 3. Discrete page selection between 0-2, and 1-3 is done with internal software. This gives the four states for MSB, and LSB page selection. And this is where I must fall short of the complete account of the 256K technique due to the copyright laws covering this ground as outlined earlier in this article.

CPU signals

Only a small explanation on a few CPU signals is all that remains. The crystal, Q1, R1, R2, C1, C6, and one gate of the 74LS04 are used to generate a reliable clock usable by the Z80A. The Z80's like as near as possible to a 0 to +5 volt swing. My original circuit had a couple of gates of a 74LS04, a few resistors and capacitors, and the crystal. It worked OK until an SGS brand 74LS04 was used. Altering the value of the resistors in the circuit overcame this problem, but I felt a more stable clock circuit was the order of the day.

The Test (NMI*) and reset pins of the CPU are held high via R9 and R3. Pressing the switches grounds these points and the program counter is set to a new value. The reset jumps to address zero, (same as power-up), and the Test jumps to 66Hex, which outputs the test message. Capacitor C2 is used to debounce the reset circuit.

As the method of accessing the DRAMs rules out running any software in this region, the only bytes that can be used for programming are the 208 bits of the CPU registers. The stack pointer is only used for arithmetic functions, not in the normal mode, so CALLs or RETURNs can't be done in the usual fashion. I don't have the luxury of using a full byte as a flag. One bit generally accounts for a flag. The software has been expanded about as far as it can go without additional hardware. As mentioned before, the Rev.F board has provision for an 8K static RAM. This will open up new uses for PBUFF, and add many new features. 