99'ER ON LINE

EDMONTON 99'ER COMPUTER USERS' SOCIETY

P. O. BOX 11983
EDMONTON ALBERTA
CANADA T5J 3L1

99'ER ON LINE... is the newsletter of the Edmonton 99'er Computer User's Society published ten times a year. Unless otherwise stated, all articles may be republished in other Newsletters provided that source and author are identified. We will in turn credit authors quoted in 99'er ON LINE.

NEWSLETTER CORRESPONDENCE: Editor: Wayne Gruenewald, 131 Garland Cres. Sherwood Park, Alberta, Canada. T8A 2R1 (403) 467-9034.

OFFICERS: Vice President Jamie Moore; 2nd Vice President Andrew Webster; Treasurer Dennis Miller; Secretary Rene Aloisio.

OFFICERS AT LARGE: Paul Helwig Newsletter Library; Gordon Bradlee Book Library; Win Appelt Disk Library; Ron Hohmann Module & Cassette Library; Suren Fustukian Sysop, T.I. Bulletin Board 433-2848,300/1200Baud.

DISCLAIMER: Information published in this Newsletter is created by and for amateurs, therefore, we cannot guarantee the accuracy or use of presented information.

REGULAR MEETINGS: of the Edmonton 99'er Computer User's Society are held on the second Tuesday of each month in room 849 of the General Service Building of the University of Alberta from: 7:00 till 10:00PM, and are open to all members in good standing. Non-members may attend their first meeting free of charge.

ADVERTISEMENTS: Commercial space is available in this Newsletter at the following rates: Full page $20.00, Half page $15.00, 1/4 page $10.00. Discuss your needs with Wayne, at (403) 467-9034, or the next meeting, alternatively send "photo ready" copies to him. Members may advertise their personal computer related items for free but are asked to limit their ads to about 50 words. Mail your ads., to the Editors address or hand it to him at the General Meeting; Newsletter deadline 3'rd Monday of the month.

MEMBERSHIP FEES: Family; 12 months $20.00, 6 months $15.00. Students: 12 months $15.00, 6 months $10.00. New member initiation; $20.00.

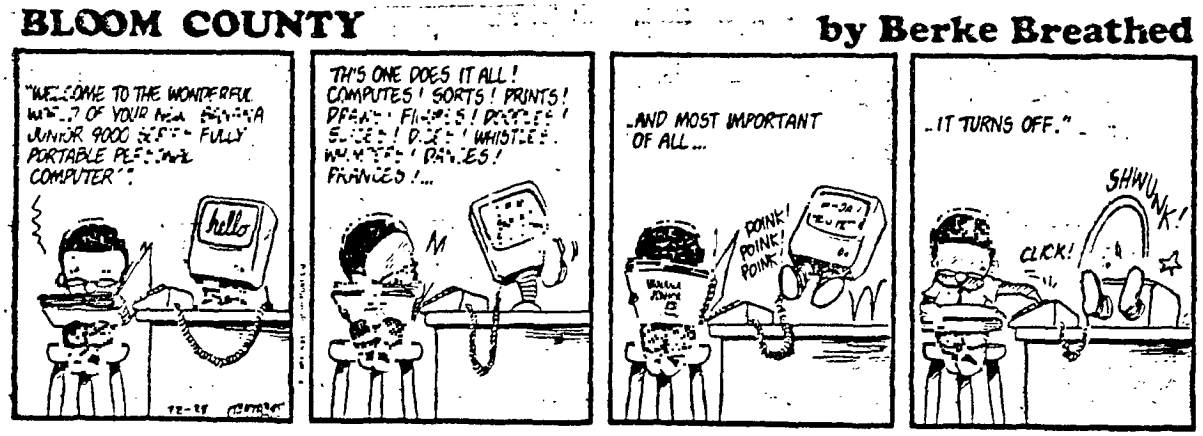
¦----------(News Letter)----------¦

NEXT MEETING.

The next meeting will be March 13th, At 7:00PM. General Services Building U of A Campus room 849.

¦----------(News Letter)----------¦

BLOOM COUNTY **by Berke Breathed**



JANUARY MEETING

By: Rene Aloisio

Jamie opened the meeting at 8:15 p.m. with a discussion of the MAGICFM program. He discussed uploading and downloading with this program. Two TI994A computers were hooked directly together to demonstrate various features of the program. He also demonstrated a new program call Mass Transfer.

Win showed us some of the new features of TI-ARTIST PLUS.

Supercomputer could store 2,000 Bibles

ED STRUZIK
Journal Staff Writer

Edmonton

An Edmonton company today unveils a parallel computer so powerful it has the capacity of storing 2,000 Bibles.

In fact, with the ability to harness 1,044 off-the-shelf commercial processors, including 4.2 billion bytes of memory, Myrias Research Corp.'s computer is the most powerful of its kind in the world.

"Industry experts now agree that parallel computers harnessing the power of hundreds of thousands of 'off-the-shelf' microprocessors to work on a single computer problem will be the high-performance computing standard for the 21st century," says Myrias president Peter Gregory. "Myrias has made it a reality in 1990."

Unlike other massive parallel systems designed to use custom processors that can operate on only one to four bits of information at a time, the Myrias system is based on full 32-bit, off-the-shelf processors.

This makes it not only cheaper, but more broadly applicable. It also allows for a higher level of compatibility with industry standards.

For those who speak the computer language, the system executes 3,670 MIPS (million instructions per second) and 630 MFlops (million floating point operations per second).

In comparison, large mainframe computers typically execute less than 100 MIPS.

Gregory expects the supercomputer will be of great value to Canadian scientists and technologists. Potential uses include weather prediction and climate modelling, designing new industry materials, designing drugs, medical imaging and diagnostics.

He also sees it being used in improving techniques in resource exploration as well as in sea and space exploration.

The 1,044 processor system backs up the company's claim that programs which can scale up in complexity by using larger data sets or by increasing the level of detail will result in speed improvements on larger systems.

Starting at minicomputer prices, and having the ability to expand to virtually any practical size means that style of computing will be available to a much broader range of users than previous supercomputers.

Gregory claims that the genius of the Myrias system lies in the company's unique software that helps users express their problems in a simple and quickly-grasped manner, and then efficiently manages the multiple processors, memories, and other internal workings of the system automatically.

The ballpark price for the new supercomputer will be about $10 million.

Small supercomputers made by Myrias have received high marks in tests at the Alberta Research Council and the University of Alberta.

"You've got computer virus. There's a lot of it going around these days."

Assembly Language Won't Byte

By: Peter Lottrup

In the first part of this series, we saw how to use the first three of the seven directives of the Line by Line Assembler included with the Mini Memory Command Cartridge and how to structure an Assembly Language program line. Now we will start to write small programs and in that way learn our first words of Assembly Language vocabulary. We will also learn how to END and execute a program once it has been typed in and how to save it on tape. Finally, we will learn a new assembler directive: STR.

Displaying a Message

First we will write a small program that will display the words TEST PROGRAM in the center of the screen. Before we begin to do this though, we must remember a little about the display screen's structure. As we know from BASIC, the screen is divided into 24 rows and 32 columns, giving it 24 x 32 squares, or 768 positions. The top left corner is the first position, and the bottom right is position 768.

To display the message, you have to tell the computer several things. You must first tell it in which screen position you want the message to begin. This number can be in decimal or in hexadecimal (base sixteen) form; if it in the later, it must include the > sign fist to indicate this. You will also have to tell the computer how long the message will be, exactly how many characters, including spaces. Finally, you will have to inform the computer what the address (or label) of the text is, so it will know where to find what it has to display. Your listing will look like this:

```
7D00 045B    LI R0,392
7D04 0354    LI R1,ST
7D08 0244    LI R1,12
7D0C A100    BLWP@>6028
7D10 0007 NN JMP NN
7D12 6100 ST TEXT'TEST PROGRAM'
```

(The first column of numbers is one memory location address; the second column is the present contents of that address, which at this stage could be any hexadecimal number.) Follow the instructions in part 1 of this series to enter the Line by Line Assembler and then begin entering these program lines.

Explanation of the Program

The first line means load register number zero with the immediately following decimal value of 392. The instruction LI (Load Immediate) simply means "place the following value in ..." R0 specifies a certain register, which is defined in the Editor/Assembler manual as a memory word that serves a specific purpose. For the time being, just think of it as a box in the computer's memory where you place a value that you want the computer to remember. Seeing it in this way, you should understand from the first line of the program that we are telling the computer to load the value of 392 (the decimal position of the center of the screen) into "box" (register) number zero. We found this number in the following manner: The message had to appear in row 12, halfway down the screen. As each row has 32 characters, the value of the first position in that twelfth row was found by multiplying 12 x 32. Then, we added 8 characters to end up at the place where we wanted the message to begin.

The next line is quite simslar to the one we have just seen: We are telling the computer to load into register number one the label of the place containing the text to be displayed. In reality, the computer converts the label, ST, into a number (or memory address) for its use. In this way, the computer knows where to find the text to display.

The third line is similar to the other two: We are telling the computer to load the value 12 into register number two. This will tell the computer how long the message is going to be.

The fourth line is the one that actually causes the message to be displayed on the screen. In this case, we are trying to write a message which contained in the computer's memory (CPU RAM) to the screen (VDP RAM). There is an Assembly Language routine which does this for us: the VMBW, or VDP Multiple Byte Write routine. As the name tells us, This routine takes bytes from CPU memory and writes them into specified locations in the VDP memory (on the screen). If we were using the Editor/Assembler, we could use the mnemonic VMBW in the following line:

```
BLWP @VMBW
```

BLWP is similar to GOSUB in BASIC. It tells the computer to branch and execute the routine (VMBW) we are specifying. We can't write the line like this when using the Line by Line Assembler unless we add some extra lines to the program. This is because this assembler does not recognize what "VMBW" stands for. But from the Mini Memory manual (page 35), we know that the VMBW routine is at hexadecimal address >6028. So in the Line by Line Assembler, if we write:

```
BLWP @>6028
```

the line will be correct. Then, using the information we have already placed into the computers memory, the message will be displayed on the screen. The VMBW routine beginning at address >6028 expects to find certain values in certain registers: In register 0, it expects to find the fist location to use in its display. In register 1, it expects to find the address of the message to display (in this case, a label which it interprets as the address >7D12). In register 2, it expects to find the length of the string to display.

(continued)

If we just stop the program there, however, the message flashes on and off before we can even see it. So the next line creates an endless loop, causing the computer to jump (JMP) to that same line over and over. Note that the JMP instruction in Assembly Language is the same as the GOTO statement in BASIC; neither is subject to any condition.

Finally, the only thing we need to include in our program is the text we want to display using the TEXT directive, which we discussed in part one. This text must be labeled with the same label we used to refer to it in the second line of the program: ST.

Running the Program

When you have finished typing in the lines as they appear in the program listing, you are ready to end. Type a space, type END, and press ENTER. The following message will be displayed:

0000 UNRESOLVED REFERENCES

This means that all the labels mentioned in the program have been used in some way or other, that is, that you have not mentioned a label in an operand field which did not exist in a label field. For instance, imagine that we had not included the line with the TEXT directive and the ST label. Then, the second line of the program would be mentioning a label which did not exist in the program. If you ended the program at that point, the message displayed would be:

0001 UNRESOLVED REFERENCES

Do not end the program if all the references have not been resolved because it will not run correctly. If you press any key except ENTER, you will return to the memory location where you left off. Then you can use the SYM directive (discussed below) to find the unresolved label or labels. If you do get the message:

0000 UNRESOLVED REFERENCES

meaning that all labels are correct, then press ENTER twice and you will return to the Mini Memory main menu.

Now let's look at how to RUN our program using EASY BUG. (Next time, we will learn how to SAVE the program by name so that you will be able to run it directly, like the LINES demonstration program.) Return to the title screen by pressing QUIT, and choose the EASY BUG option. Press any key to skip the menu, and when the question mark appears, type:

?E7D00

and press ENTER. This means execute the Assembly Language program which starts at memory location number 7D00 (which is where we began to write our program). You will see that the message we wanted to show is displayed immediately on the center of the screen. Note that the only way to regain control of the computer in this program is by turning off the power, and even if we do this, the program is still kept in Mini Memory.

If you now want to save the program on cassette, select EASY BUG, skip the title screen and type:

?S

S means save the contents in memory to tape. The computer will then ask:

FROM?

With this prompt, the computer wants to know FROM what memory location you want to begin recording. To be sure that you get everything, it is best to type 7000, as it says in the manual. It is not necessary to include the > sign. The computer will then ask:

TO?

With this prompt, the computer wants to know the last memory address to save. Here type 7FFF, again to be sure you get everything. This means that you are storing the entire contents of the Mini Memory cartridge on tape. When you have typed 7FFF and pressed ENTER, the usual procedure to save on tape will follow.

To load the contents later on, select EASY BUG and then choose the L command. The computer will then respond with the standard sequence of instructions for loading from cassette.

When you have finished loading the program and the question mar appears, type E (execute Assembly Language program), followed by the hexadecimal address where the program begins, in this case >7D00.

(continued)

The SYM (Symbol Table) Directive

As we have seen, it is very possible to get the message that there are one or more unresolved references when you END a program. If this happens, it is often very difficult to find those labels, especially when the program is long. In these cases, use the SYM assembler directive. It is simple to use: Type a space to skip the label field, and type SYM. Then press ENTER. All resolved and unresolved references will be displayed, up to a maximum of 32 unresolved references. Then you can see exactly where the problem lies.

Try this out by typing the program to display text once again, except for the line where you include the text. When you end the program, you will get the message:

```
0001 UNRESOLVED REFERENCES
```

as we have already seen. Press any key except ENTER to return to the assembler, and then type a space and SYM. Press ENTER, and the following will be displayed:

```
RESOLVED REFERENCES
NN-7D10
UNRESOLVED REFERENCES (WORD)
ST-7D06
```

RESOLVED REFERENCES are all the labels which have been gien a value, or defined: UNRESOLVED REFERENCES (WORD) are all the labels which are referenced in any instructions except JUMP instructions. (in our case, the ST label is displayed, together with its memory location, indicating that our problem lies there.) A third possibility is:

```
UNRESOLVED REFERENCE (JUMP)
```

This is any label which has been referenced in a JUMP instruction but has not been given a value or defined. For this situation, add this line:

```
7D12 xxxx    JMP CV
```

and type SYM again. It will show you the resolved labels, the unresolved word label, and the unresolved jump label (CV-7D12). In this way, it is easy to find out which labels are unreferenced in the program. Using the AORG directive, it is easy to move back to any line and correct it, or add anything missing.

Conclusion

In this part of the series we have learned many new things: how to type in, run and save short programs, assembler instructions, and the assembler directives SYM and END.

Now that you are starting to understand the structure of Assembly Language programs, we are ready to look at a whole list of important Assembly Language instructions so that you can start typing out your own programs. In the next part of this series, we will also start saving programs by name, so we can run them directly.

You should now be able to experiment a bit by writing your own programs, based on what you have learned in this article and prior knowledge. You will see that the dark cloud hanging over Assembly Language is beginning to diappear.