SHARE “Assembler BootCamp”

Assembler BootCamp Starter Kit

Synopsis:

This document describes several useful tools for Assembler Language beginners:

1. ASSIST/I, a PC-based program that edits, assembles, and simulates execution of simple Assembler Language programs, with extensive debugging facilities.
2. Two host-system macro sets:
   a. Macros that provide the functions of the “X” macros supported by ASSIST/I (and more!).
   b. Basic I/O macros that print data items and registers, dump storage contents, read 80-byte record images, print lines, and do simple conversions.
   c. The source code and documentation for ASSIST, a mainframe-based student-oriented assembler/interpreter.

Disclaimer

Sample Programs

The programs described in this document are available:

- A CD-ROM containing a large amount of useful material, in the **ABC** directory. That directory contains the following folders and files:
  1. ASReview, which contains these folders:
     - ASSIST Source for Review
       - Doc Files
       - Sample Assignments
       - Source Files
       - Test Programs
       - X-Macros
  2. ASReview.zip
     - A "zip" file containing the materials in the "ASReview" folder.
  3. BOOTASST
     - CAS.EXE and demo programs
  4. BootMacs
     - The "Simple I/O Macros" (These macros, unlike the PSUMACS, do not require additional run-time routines for link-time inclusion in an executable program.)
  5. Presentations
     - The presentation-slides handout from the Boot Camp sessions, in PDF format. (Note: your CD-ROM may have materials from a previous Boot Camp.)
  6. Problem Sheets
     - The "WHATNOW" exercise.
     - The "DEMENTIA" exercise.
  7. PSUMACS
     - The Penn State University X-Macros, for use in an IBM mainframe environment.
  8. PSUPROGS
     - The Penn State University X-Macros runtime support routines, for use in an IBM mainframe environment.
  9. Starter Kit
     - This document, in PDF format.
- All materials are available at http://www.kcats.org/assist/ and http://www.kcats.org/share/.

Acknowledgments

- John Ganci's meticulous proofreading was very helpful.
- Michael Stack provided advice, code, and documentation.
- John Ehrman was the editorial hack.

(Revised 17 Feb 2010, Formatted 29 Jun 2010, 1636.)

Contents

Figures

ASSIST/I

ASSIST (or ASSIST/360) was written at the Pennsylvania State University by John Mashey. It executed on host mainframe systems, and supported a large set of "X" macros; these are described at "ASSIST Input/Output and Debugging Instructions/Macros" on page 42. Later, a PC-based version of ASSIST called ASSIST/I was written at Northern Illinois University. This is the program described in this document.

ASSIST/I has a number of limitations you should be aware of:

- It is based on an early System/360 instruction set, and therefore does not support many instructions that may currently be available on your host system.
- Sometimes the listing may show base-register resolutions different from what's in the assembled code. Check the storage displays at the beginning of execution to be safe; the code is probably OK.
- You will find that there are tradeoffs between working with ASSIST/I on your PC, or between the PC and the host system. The text assumes you are using the PC only.

Editorial Comments

1. The editor program provided with ASSIST/I is flexible and can do many useful things, but is quite awkward to use. Programs and data for input to ASSIST/I can be prepared on host systems; tab characters are OK. Save your host files as variable-length (V-format) records, and then download them to your chosen workstation directory. Also, you can use any PC-based line editor to create and modify files.
2. The programming styles used in the sample programs is not what most people would consider "correct". Rather than using conditional branch masks like `B'1011'`, most programmers would prefer to use an extended branch mnemonic like BNM ("Branch if Not Minus"). Similarly, manually counting lengths of character strings and offsets into other fields (as in many examples) is poor practice; it is much better to use length-attibute notation (`L'`) or an equated length.
3. Not all the sample programs on the diskette in the `\BootAsst\` directory will assemble without diagnostics on host systems using the High Level Assembler (but the diagnostics are typically low-level warnings).
4. The assembled code from ASSIST/I starts at address zero, which is not an available area of main storage on host systems.

A Brief Introduction to ASSIST/I

This chapter provides a very brief introduction to some of the basic functions of ASSIST/I and to the text editor that is an integral part of this package. “Interactive Debugging With ASSIST/I” on page 8 expands upon this material to provide enough information for you to work effectively with ASSIST/I. Chapter “ASSIST/I User's Guide” on page 17 is a quick reference for this software.

Several notational conventions are used. These are

1. User responses to ASSIST/I are shown in capital letters.
2. The symbol <cr> is used to indicate that the Return key is to be depressed.
3. Editor commands require use of the Control key. The symbol ^ indicates that the Control (or Ctrl) key is to be held down while the letter immediately following this symbol is typed. For example, ^K may be thought of as a single keystroke that requires two keys to be depressed, much as the Shift key on a standard typewriter is used with a letter key to produce uppercase. The instruction ^K is pronounced “control K”, and ^QX is pronounced “control Q X”.
4. The General Purpose Registers are simply called “registers”, and specific registers are typically referred to as “Rn”, as in R2 for register 2.
5. Addresses, register contents, and memory contents are shown in hexadecimal.

Getting Started with ASSIST/I

ASSIST/I may be invoked by typing

```
CAS<cr>
```

The screen shown in Figure 1 will appear:

```
                          IIIIIIII
                             II
          A S S I S T  /     II
                             II
                          IIIIIIII

          E   ===>   Edit a program
          R   ===>   Run a program
          F   ===>   Execute a program Final Run
          P   ===>   Print a program or listing
          A   ===>   Alter ASSIST/I options
          Q   ===>   Quit

          Enter desired function: ___

ASSIST/I  Version 2.03                    Serial #:  AIM000000
Copyright (C) 1984                               BDM  Software
```

Figure 1. ASSIST/I Main Screen

This is the main ASSIST/I menu; its presence indicates that the system is ready for use. Note the options offered on this menu; only the E and R options are discussed in this chapter.

We'll show how to create a simple program, save it on disk, assemble the program, and then execute it one instruction at a time. The ASSIST/I text editor may be used to create the program; to do so, choose the Edit option from the main menu by typing:

```
E
```

When you enter the editor mode, the prompt

```
Enter name of file to edit:
```

asks you for the name of the file to edit. Respond to this prompt by typing the file identifier of the program, for example:

```
B:TEST.ASM<cr>
```

The file name entered must be valid; its validity depends upon the operating system under which ASSIST/I is running. In this case, `B:` designates the B drive. If no drive is specified, the program defaults to the current logged drive.

After the editor has initialized this new file, you can begin typing in a program. So long as you make no errors, you can just type in the text of your program, using `<cr>` to end each line. The editor commands are described later. (If you inadvertently introduce an error, skip ahead to Section "Editing Programs" on page 17 to learn how to correct it.) For now we assume that you can type in the following program ADD2 without making any errors:

```
ADD2   CSECT
       L     1,16(,15)     LOAD 1ST NUMBER INTO R1
       L     2,20(,15)     LOAD 2ND NUMBER INTO R2
       AR    1,2           ADD THE TWO NUMBERS IN R1
       ST    1,24(,15)     STORE RESULT INTO 7TH
*                          WORD IN PROGRAM
       BCR   B'1111',14    EXIT FROM PROGRAM
       DC    F'4'
       DC    F'6'
       DS    F
       END   ADD2
$ENTRY
```

Figure 2. Sample ADD2 Program

Now, save a copy of this program to the disk in drive B, by typing:

```
^KX
```

Remember to hold down the Control key while typing K, then type X. The editor saves the program and returns to the ASSIST/I menu. (This program is in the `\BootAsst\` directory as file `DEMOA.ASM`.)

To run the program, select the `Run a program` option from the menu by typing

```
R
```

This option calls the ASSIST/I interpreter. The screen indicates when the first pass of the interpreter is complete and indicates the start of the second pass. In the second pass, the program is actually assembled, and the program listing appears on the screen as each instruction is assembled. If, for some reason, your program does not assemble, the editor can be recalled and the necessary corrections made to the program. The editor commands are covered in Section "Editing Programs" on page 17. Here we assume, for simplicity, that the assembly was successful. After assembly is successfully completed, the debugger is automatically called and the screen shown in Figure 3 on page 4 is displayed.

```
PSW AT BREAK   FFC50000 0F000000

R0-7 :  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 00000020 00000068 00000000

000000       5810F010   5820F014   1A125010   F01807FE       *..0...0...&.0...*
000010       00000004   00000006   F5F5F5F5   F5F5F5F5       *........55555555*
000020       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000030       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000040       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000050       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000060       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000070       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000080       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000090       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000A0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 3. Screen After Assembly

This screen, which displays a snapshot of the current status of the program, should be studied in some detail. Some observations may be made:

1. The first line of the display shows the current contents of the **Program Status Word**. Note that the rightmost three bytes of the PSW contain 000000. This is the **Instruction Address**, which always contains the address of the *next* instruction to be executed. By convention, ASSIST/I always indicates the address of the first instruction as address 0. This is a handy aid in calculating the relative addresses of other memory locations in the program's memory area.
2. The second and third lines show the contents of the 16 General Purpose Registers. Note that only registers 13, 14, and 15 contain anything other than a sequence of F4s. These three registers are special registers used by IBM operating systems. Their contents must never be changed by a programmer unless extreme caution is exercised. By an ASSIST/I convention, the F4s in the other registers indicate that these registers have not been altered by the program. This makes it easy to identify such unmodified registers.
3. The next block of printed lines shows the current status of the program's memory space. The first column of each of these lines contains the hexadecimal address of the first byte displayed to its right in the second column. The next four columns each show the hexadecimal contents of a four-byte fullword in memory. Finally, the lines in the rightmost column, enclosed between asterisks, show the character representation of all printable characters in the respective memory locations; the periods indicate non-printable characters. Examine the contents of the first four fullwords of this memory display, and observe that the executable instructions of the program have been loaded into these words. You should take the time to disassemble these instructions and convince yourself that they have been correctly assembled. Note also that the fifth fullword in memory (at address 000010) contains the number corresponding to the first DC instruction in the program and that the sixth fullword (at address 000014) contains the second number. The remainder of the memory locations in this display are all filled with F5s, which is the conventional way ASSIST/I displays unused memory locations. This convention allows such locations to be readily identified.
4. The last line of the display shows the options offered by ASSIST/I. Only the S(tep) option is used in this section; the other options are covered in “Interactive Debugging With ASSIST/I” on page 8.

We have verified that the program ADD2 has been correctly assembled and loaded into memory. This program is now ready for execution. To observe the execution of the program in detail, we can execute the program instructions one at a time, observing the effect of executing each instruction. This is done using the S (Step) option. To execute the first instruction, type:

```
S
```

After execution of the first instruction, the screen depicted in Figure 4 should be displayed.

```
PSW AT BREAK   FFC50000 8F000004

R0-7 :  F4F4F4F4 00000004 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 00000020 00000068 00000000

000000       5810F010   5820F014   1A125010   F01807FE       *..0...0...&.0...*
000010       00000004   00000006   F5F5F5F5   F5F5F5F5       *........55555555*
000020       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000030       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000040       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000050       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000060       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000070       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000080       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000090       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000A0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 4. Screen After First Instruction is Executed

Study this screen for a moment, and verify the following:

- The PSW has been updated, and the Instruction Address now contains the address (000004) of the next instruction to be executed, the second Load instruction.
- A copy of the first number, which is in memory location 000010, has replaced the contents of Register 1.

Execute the second Load instruction by typing:

```
S
```

Look at the displayed screen, and note that the Instruction Address now contains the address (000008) of the Add instruction. Also note that the contents of Register 2 have been replaced by a copy of the second number, which is located in the sixth fullword in memory.

Execute the Add instruction by typing:

```
S
```

You should now be looking at the screen depicted in Figure 5 on page 6. Note that R1 now contains 0000000A, the hexadecimal equivalent of decimal 10. This indicates that the Add instruction has been successfully executed. Note also that execution of the instructions thus far has not altered the contents of the memory locations. One more step will change this pattern. The Instruction Address now contains the address of the STore instruction.

```
PSW AT BREAK   FFC50000 6F00000A

R0-7 :  F4F4F4F4 0000000A 00000006 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 00000020 00000068 00000000

000000       5810F010   5820F014   1A125010   F01807FE       *..0...0...&.0...*
000010       00000004   00000006   F5F5F5F5   F5F5F5F5       *........55555555*
000020       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000030       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000040       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000050       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000060       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000070       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000080       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000090       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000A0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 5. Screen After Third Instruction is Executed

To execute the STore instruction, type:

```
S
```

The screen shown in Figure 6 should now appear. Note the contents of the seventh fullword in memory (at address 000018), and verify that a copy of the contents of R1 has been stored there.

```
PSW AT BREAK   FFC50000 AF00000E

R0-7 :  F4F4F4F4 0000000A 00000006 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 00000020 00000068 00000000

000000       5810F010   5820F014   1A125010   F01807FE       *..0...0...&.0...*
000010       00000004   00000006   0000000A   F5F5F5F5       *............5555*
000020       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000030       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000040       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000050       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000060       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000070       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000080       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000090       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000A0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 6. Screen After Fourth Instruction is Executed

You have now observed execution of the entire program, and all that remains is to return control to the system.

Perform one final S. Information will be flashed on your screen, indicating that execution of the program has been completed and that no warnings were issued or errors detected. No output is displayed, since there were no instructions in the program that would produce output. The ASSIST/I menu is again displayed. You can get a listing of the program by typing the F option

on the menu; you will be prompted for a string that will appear on the first line of the listing. (The listing for the program just described is in the \BootAsst\ directory as file DEMOA.PRT.)

ASSIST/I supports the pseudo-instructions XREAD, XDECI, XDECO, and XPRNT, which perform simple input and output functions. These instructions are illustrated in "Interactive Debugging With ASSIST/I" on page 8 and are described in detail in "ASSIST/I Pseudo-Instructions" on page 25.

Getting Printed Output

You will frequently wish to get a printed copy of an entire listing. To do so, set the ASSIST/I options to cause a copy to be written to a disk file. The copy on the disk will include all of the characters in each line and can easily be written to your printer.

To set options for ASSIST/I, respond to the main ASSIST/I menu with A (Alter options). This causes the menu displayed in Figure 7 to appear.

```
            A S S I S T / I  Options

      1) Save output listing       - y
      2) Maximum # lines           - 500
      3) Maximum # instructions    - 5000
      4) Maximum # pages           - 100
      5) Maximum size (in bytes)   - 2700


    Enter option number to alter (RETURN to quit): _
```

Figure 7. ASSIST/I Execution Options

By responding with a 1, you can toggle the option to cause a disk file to be created whenever you assemble and execute a program. Thus, if the option is already set to Y, you can just use <cr> to go back to the main menu; otherwise, typing 1 will change the setting from N to Y (and then a <cr> can be used to go back to the main menu). The default options for ASSIST/I are set so that this first option is usually the only one that you alter.

Assuming that the option is set to cause a disk file to be created, a disk file will be produced whenever you assemble and execute a program. The disk file will be given a name based on the name of the file containing the program that was assembled and executed. The name of the disk file is formed from the original file name by altering the file type to PRT. That is, if the program file is named B:SUMUP, the disk file will be named B:SUMUP.PRT. If the program file is named B:SUMUP.ASM, the disk file will be named B:SUMUP.PRT. Because of this specialized use of the file type PRT, you should never use a file type of PRT for files you create.

Once a file has been created on disk (either by using the editor or by assembling and executing a program), it can easily be printed by using the P (Print) option available in the main menu. This option prompts you for the name of the file you want printed and causes the contents of that file to be printed.

Other ASSIST/I options are described at "Altering ASSIST/I Options" on page 38.

Interactive Debugging With ASSIST/I

Introduction

The previous chapter introduced the basic features of ASSIST/I. In this chapter, we discuss the features of ASSIST/I that make it a truly convenient tool for learning IBM assembler language. The basic topics to be covered are:

- How to use XREAD and XPRNT to perform input/output operations.
- How to debug a program interactively.

We assume you have covered the material in the previous chapter.

Much of the time spent writing and debugging a program would be better spent in first gaining a clear understanding of the task and only then carefully writing the program. Use of an interactive environment such as that provided by ASSIST/I cannot reduce the effort expended in the creation of this original program. However, once you have written a program, the time and effort required to verify that it works properly (or, more likely, to correct it and produce the desired program) can be substantially reduced using ASSIST/I. Time spent mastering the techniques in this chapter will be repaid many times as you address more complex programming problems.

A Sample Program

In the previous chapter, you learned how to edit and run a simple program in the interactive environment. However, no attempt was made to read input data or to produce output. To verify that the program worked properly, you simply observed changes to memory and to the registers as the instructions were executed. Here, we will create a program that performs input and output: it adds pairs of numbers (like the one in Figure 2 on page 3), but also reads and prints data.

This example uses two ASSIST/I pseudo-instructions, XPRNT and XREAD:

- The format of the XPRNT instruction is

  ```
  label    XPRNT addr,length
  ```

  where

addr	is the D(X,B) or implicit address of the print line in storage
length	is the number of bytes in the record to be printed.

  The first byte of the print record is a “carriage control character”, explained in section “XPRNT Instruction” on page 27.

- The format of the XREAD instruction is

  ```
  label    XREAD addr,length
  ```

  where

addr	is the D(X,B) or implicit address of the area into which the input should be read
length	is the number of characters to be read. The length should be 80 or less. If it is less than 80, the remaining characters of the input record will be ignored.

  When using XREAD, you must indicate where the input lines are to be found. The most common method is to include the input lines immediately following the $ENTRY line, which must immediately follow the last line of your program.

Detailed descriptions of the XPRNT and XREAD pseudo-instructions can be found in section “ASSIST/I Pseudo-Instructions” on page 25.

To illustrate interactive debugging, suppose you have created the file shown in Figure 8 on page 9, which we will assume is called B:SUMUP. (This program is in the \BootAsst\ directory as file DEMOB.ASM.)

```
*****************************************************
*  THIS PROGRAM READS CARDS, EACH OF WHICH CONTAINS
*  TWO NUMBERS.  THE SUM OF THE TWO NUMBERS IS PRINTED.
*****************************************************
*
SUMUP     CSECT
          USING SUMUP,15
*
          XPRNT HEADING,28   PRINT A PAGE HEADING
*
          XREAD CARD,80      READ THE FIRST CARD
*
CHECKEOF  BC    B'0100',EXIT BRANCH ON EOF
          XDECI 2,CARD       WE ASSUME THAT BOTH NUMBERS
          XDECI 3,0(,1)      ARE VALID
*
          AR    2,3          CALCULATE THE SUM
*
          XDECO 2,OUTPUT     PUT PRINTABLE FORM INTO PRINT LINE
*
          XPRNT CRG,13       PRINT THE SUM (SINGLE SPACED)
          XREAD CARD,80      TRY TO READ THE NEXT CARD
          BC    B'1111',CHECKEOF GO CHECK FOR EOF
EXIT      BCR   B'1111',14   LEAVE THE PROGRAM
*
CARD      DS    CL80         CARD INPUT AREA
*
CRG       DC    C' '         SINGLE SPACE CARRIAGE CONTROL
OUTPUT    DS    CL12         OUTPUT THE SUM HERE
*
HEADING   DC    C'1THIS IS THE OUTPUT OF SUMUP'
          END   SUMUP
$ENTRY
1 2
2 3
```

Figure 8. Program to Read/Add Numbers

Then, to run the program, you simply use the R (Run) command, specifying B:SUMUP as the file to be assembled and executed. As we pointed out in the last chapter, ASSIST/I pauses after assembling the program. If you were to respond with G (Go), the output shown in Figure 9 on page 10 would be produced. (The listing file is in the \BootAsst\ directory as file DEMOB.PRT.)

```
   LOC OBJECT CODE    ADDR1 ADDR2 STMT SOURCE STATEMENT

                                   1 ****************************************************
                                   2 *  THIS PROGRAM READS CARDS, EACH OF WHICH CONTAINS
                                   3 *  TWO NUMBERS.  THE SUM OF THE TWO NUMBERS IS PRINTED.
                                   4 ****************************************************
                                   5 *
000000                             6 SUMUP     CSECT
000000                             7           USING SUMUP,15
                                   8 *
000000 E020 F08D 001C 0008D        9           XPRNT HEADING,28   PRINT A PAGE HEADING
                                  10 *
000006 E000 F030 0050 00030       11           XREAD CARD,80      READ THE FIRST CARD
                                  12 *
00000C 4740 F02E            0002E 13 CHECKEOF  BC    B'0100',EXIT BRANCH ON EOF
000010 5320 F030            00030 14           XDECI 2,CARD       WE ASSUME THAT BOTH NUMBERS
000014 5330 1000            00000 15           XDECI 3,0(,1)      ARE VALID
                                  16 *
000018 1A23                       17           AR    2,3          CALCULATE THE SUM
                                  18 *
00001A 5220 F081            00081 19           XDECO 2,OUTPUT     PUT PRINTABLE FORM INTO PRINT LINE
                                  20 *
00001E E020 F080 000D 00080       21           XPRNT CRG,13       PRINT THE SUM (SINGLE SPACED)
                                  22 *
000024 E000 F030 0050 00030       23           XREAD CARD,80      TRY TO READ THE NEXT CARD
00002A 47F0 F00C            0000C 24           BC    B'1111',CHECKEOF GO CHECK FOR EOF
00002E 07FE                       25 EXIT      BCR   B'1111',14   LEAVE THE PROGRAM
                                  26 *
000030                            27 CARD      DS    CL80         CARD INPUT AREA
                                  28 *
000080 40                         29 CRG       DC    C' '         SINGLE SPACE CARRIAGE CONTROL
000081                            30 OUTPUT    DS    CL12         OUTPUT THE SUM HERE
                                  31 *
00008D F1E3C8C9E240C9E2           32 HEADING   DC    C'1THIS IS THE OUTPUT OF SUMUP'
                                  33           END   SUMUP

***   0 STATEMENTS FLAGGED - 0 WARNINGS, 0 ERRORS

*** PROGRAM EXECUTION BEGINNING
ANY OUTPUT BEFORE EXECUTION COMPLETE MESSAGE IS PRODUCED BY USER PROGRAM ***

THIS IS THE OUTPUT OF SUMUP
             3
             5

***   EXECUTION COMPLETED ***
```

Figure 9. Listing of Program to Read/Add Numbers

Interactive Debugging with ASSIST/I

"A Brief Introduction to ASSIST/I" on page 2 illustrated the most basic technique for interactive debugging: single-stepping through instructions to verify that they produce the expected results. Here, we discuss the use of:

- Breakpoints
- Interactive alteration of memory, registers, and the PSW
- Instruction traces

To illustrate these techniques, we utilize a slightly modified version of the program displayed in Figure 8 on page 9 and Figure 9. Consider a version of the program in which line 17 of the program listing has an error — rather than adding registers 2 and 3 together, it causes register 4 to be added to register 2. Thus, in the example we are considering, line 17 appears as:

```
000018 1A24                       17           AR    2,4          CALCULATE THE SUM (?)
```

This fairly trivial bug is sufficient to illustrate features of ASSIST/I that can isolate such errors rapidly.

Before continuing, type in the program displayed in Figure 8 on page 9, introducing the minor error discussed in the previous paragraph. Then make sure the ASSIST/I options are set to produce a copy of the listing and output in a disk file. Finally, use the Final listing option to produce a disk copy of the listing and its erroneous output, and use the Print command to get a printed listing of the program. These steps will reinforce your knowledge of the editor, while veri-

fying that you completely understand the topics discussed earlier in this chapter. Our presentation of interactive debugging depends heavily on your ability to actually invoke the debugger to operate on this program, so these preliminary steps are indeed necessary if you are to follow the material in the remainder of this section.

Once you have created the modified file containing the source program, use the Run command to begin debugging it. When the program containing the error is run, ASSIST/I will not detect any syntax errors (even though there is an intentional logic error). That is, the program will assemble, and ASSIST/I will pause before initiating execution of the program. It will print the contents of memory and wait for you to select an item from the following menu:

```
===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
```

With this program (because it is so short) it makes sense to single-step through the execution to verify that every instruction produces the expected results. However, in longer programs, it would be too tedious to have to type S repeatedly for each instruction. Instead, it is a good idea to locate critical spots in the program, points where execution could logically be halted to allow you to verify that an entire sequence of instructions has produced an expected result. In this case, the programmer can set breakpoints. A *breakpoint* designates a specific instruction and causes execution to pause immediately before that instruction is executed. That is, whenever ASSIST/I reaches the point where the designated instruction is the next to be executed, it pauses and allows the programmer to examine the state of memory and the registers. In our short example, it would make sense to set a breakpoint at the instruction on line 19 of the program listing (see Figure 9 on page 10). When execution has reached this instruction, one input record has been read, and the desired result has been calculated. A breakpoint on this instruction allows you to verify that the input had been read correctly and converted into registers correctly and that the computed result was indeed the desired sum. In this program, it really does not make sense to set more than one breakpoint. However, in longer programs, you will frequently set a number of breakpoints before initiating execution.

To set a breakpoint, simply answer B for the above menu. This causes the following sub-menu to be displayed:

```
 BREAKPOINT:  S(et), C(lear), D(isplay)
:Breakpoint: _
```

At this point, you can establish a breakpoint, remove an existing breakpoint, or display the addresses of instructions for which breakpoints are currently established. A <cr> here simply returns you to the previous menu. The first time this menu appears, there are no established breakpoints. You can set one by typing S. This causes you to be prompted for the hexadecimal address of the instruction on which you wish to set the breakpoint. In our example, you would type in:

```
1A<cr>
```

to set the breakpoint on address 00001A. (Note: leading zeros need not be typed, and lowercase letters are accepted.) If you then respond with G to cause the program to begin execution, it will execute instructions until it either terminates or reaches the instruction at address 00001A. At this point, you should Run the program, Set the breakpoint, and use Go to cause the program to execute until the breakpoint is detected during execution.

When we run the program, it reaches address 00001A and then pauses. The exact contents on your terminal screen should be as depicted in Figure 10 on page 12.

```
PSW AT BREAK    FFC50000 5F00001A

R0-7 :  F4F4F4F4 00000033 F4F4F4F5 00000002 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 000000B0 000000F8 00000000

000000       E020F08D   001CE000   F0300050   4740F02E       *\.0...\.0..&. 0.*
000010       5320F030   53301000   1A245220   F081E020       *..0.........0a\.*
000020       F080000D   E000F030   005047F0   F00C07FE       *0...\.0..&.00...*
000030       F140F240   40404040   40404040   40404040       *1 2             *
000040       40404040   40404040   40404040   40404040       *                *
000050       40404040   40404040   40404040   40404040       *                *
000060       40404040   40404040   40404040   40404040       *                *
000070       40404040   40404040   40404040   40404040       *                *
000080       40F5F5F5   F5F5F5F5   F5F5F5F5   F5F1E3C8       * 5555555555551TH*
000090       C9E240C9   E240E3C8   C540D6E4   E3D7E4E3       *IS IS THE OUTPUT*
0000A0       40D6C640   E2E4D4E4   D7F5F5F5   F5F5F5F5       * OF SUMUP5555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5PS   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 10. Screen at First Breakpoint

You again have the same menu you had before execution was initiated.

By examining the contents of memory, starting at address 000030, you can determine that the program successfully read the input line containing the numbers 1 and 2. You would expect register 3 to contain the value 1 and register 2 to contain the sum of the two input values. Register 2, however, contains F4F4F4F5. By examining the short section of code, it should be easy to detect that it was actually the instruction in line 17 that caused the erroneous results. (If it were not clear, you could always rerun the program, single-stepping through the section of code to check the effects of each instruction.) At this point, you could type Q to quit executing instructions, use the editor to correct the program, and reinitiate execution. However, as you become more proficient at interactive debugging, you will find it convenient to attempt to locate multiple bugs during a single execution of the program. For the sake of exploring the facilities offered by ASSIST/I, let us take this approach.

You can avoid actually going back to the editor to reassemble your program by "fixing" the error in the actual memory and registers of the machine and then reinitiating execution. This practice should be performed very carefully, and you must certainly keep track of the changes that are eventually to be made to the file containing the program. To fix this error before reinitiating execution, you can:

1. Alter the contents of register 2 back to 00000001
2. Alter the instruction encoded at address 000018 (i.e., alter the memory at address 000018 from 1A24 to 1A23)
3. Alter the address of the next instruction to be executed (in the PSW) to 000018.

Then, when execution is reinitiated, the AR instruction is executed again (in this case, adding the contents of register 3 to register 2).

To alter the contents of register 2, you simply respond to the prompt with R. When you are prompted for the register to be altered, a response of 2 will cause the cursor to be positioned over the contents of register 2. By typing in the desired contents (00000001) followed by <cr>, you alter the actual contents of register 2. Make this alteration, which should return you to the menu:

```
===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
```

The next step is to alter the encoded version of the instruction:

```
      AR    2,4
```

to its correct form, representing:

```
AR    2,3
```

You should verify that the encoding of the erroneous instruction is 1A24, while the corrected version should be 1A23. Note that the erroneous encoding is located at address 000018 on your screen. To alter the contents of memory, the M option of the menu must be specified. This causes the following sub-menu to be displayed:

```
MEMORY: <RET> for this screen, N(ext scr.), P(rev. scr.),  (hex address)
```

By responding with <cr>, you request the option to alter the section of memory displayed on the current screen. The other options allow you either to scroll forward and backward through memory, which would be useful with large programs in which the entire relevant memory could not be displayed on a single screen, or to move the cursor directly to a specified address in memory, by typing the actual hexadecimal address. Suppose that you respond with a <cr>. Then, you must move the cursor directly over the 4 at address 000019 (which contains the right hexadecimal digit of the byte). This can be achieved by using the same cursor-movement commands that you would normally use while typing in a file with the editor. Once the cursor is over the 4, make the correction by typing 3 followed by <cr>. With these actions, the erroneous encoded version of:

```
AR    2,4
```

is changed to:

```
AR    2,3
```

Make this alteration before continuing.

Finally, we are going to alter the address of the next instruction to be executed, to cause

```
AR    2,3
```

to be re-executed. The address of the next instruction to be executed is kept in the rightmost three bytes of the PSW. To change this address in the PSW you simply use the P option to position the cursor over the right three bytes of the PSW, where the address of the next instruction to be executed is maintained. Move the cursor over the rightmost character, and change A to 8. Then use <cr> to return to the menu. By altering the address from 00001A back to 000018, you have "fixed" the error in the assembled machine-language code, and can reinitiate execution of the program. At this point, your screen should look exactly as shown in Figure 11.

```
PSW AT BREAK    FFC50000 5F000018

R0-7 :  F4F4F4F4 00000033 00000001 00000002 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 000000B0 000000F8 00000000

000000       E020F08D   001CE000   F0300050   4740F02E       *\.0...\.0..&. 0.*
000010       5320F030   53301000   1A245220   F081E020       *..0.........0a\.*
000020       F080000D   E000F030   005047F0   F00C07FE       *0...\.0..&.00...*
000030       F140F240   40404040   40404040   40404040       *1 2             *
000040       40404040   40404040   40404040   40404040       *                *
000050       40404040   40404040   40404040   40404040       *                *
000060       40404040   40404040   40404040   40404040       *                *
000070       40404040   40404040   40404040   40404040       *                *
000080       40F5F5F5   F5F5F5F5   F5F5F5F5   F5F1E3C8       * 5555555555551TH*
000090       C9E240C9   E240E3C8   C540D6E4   E3D7E4E3       *IS IS THE OUTPUT*
0000A0       40D6C640   E2E4D4E4   D7F5F5F5   F5F5F5F5       * OF SUMUP5555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5PS   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 11. Screen After Instruction is Corrected

Now reinitiate execution of your program by specifying G, which causes the single instruction at 000018 to be executed. The breakpoint at 00001A again causes execution to be suspended. If you examine the display on your screen this time, it should match the screen depicted in Figure 12 on page 14 with register 2 containing the desired sum.

```
PSW AT BREAK    FFC50000 6F00001A

R0-7 :  F4F4F4F4 00000033 00000003 00000002 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 000000B0 000000F8 00000000

000000       E020F08D   001CE000   F0300050   4740F02E       *\.0...\.0..&. 0.*
000010       5320F030   53301000   1A245220   F081E020       *..0.........0a\.*
000020       F080000D   E000F030   005047F0   F00C07FE       *0...\.0..&.00...*
000030       F140F240   40404040   40404040   40404040       *1 2             *
000040       40404040   40404040   40404040   40404040       *                *
000050       40404040   40404040   40404040   40404040       *                *
000060       40404040   40404040   40404040   40404040       *                *
000070       40404040   40404040   40404040   40404040       *                *
000080       40F5F5F5   F5F5F5F5   F5F5F5F5   F5F1E3C8       * 5555555555551TH*
000090       C9E240C9   E240E3C8   C540D6E4   E3D7E4E3       *IS IS THE OUTPUT*
0000A0       40D6C640   E2E4D4E4   D7F5F5F5   F5F5F5F5       * OF SUMUP5555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5PS   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 12. Screen After Corrected Instruction is Executed

By typing G again, you cause execution to proceed until the breakpoint is reached again. This time your screen should look like the screen shown in Figure 13.

```
PSW AT BREAK    FFC50000 6F00001A

R0-7 :  F4F4F4F4 00000033 00000005 00000003 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 000000B0 000000F8 00000000

000000       E020F08D   001CE000   F0300050   4740F02E       *\.0...\.0..&. 0.*
000010       5320F030   53301000   1A245220   F081E020       *..0.........0a\.*
000020       F080000D   E000F030   005047F0   F00C07FE       *0...\.0..&.00...*
000030       F240F340   40404040   40404040   40404040       *2 3             *
000040       40404040   40404040   40404040   40404040       *                *
000050       40404040   40404040   40404040   40404040       *                *
000060       40404040   40404040   40404040   40404040       *                *
000070       40404040   40404040   40404040   40404040       *                *
000080       40F5F5F5   F5F5F5F5   F5F5F5F5   F5F1E3C8       *            31TH*
000090       C9E240C9   E240E3C8   C540D6E4   E3D7E4E3       *IS IS THE OUTPUT*
0000A0       40D6C640   E2E4D4E4   D7F5F5F5   F5F5F5F5       * OF SUMUP5555555*
0000B0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000C0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000D0       F5F5F5F5   F5F5F5F5   F5F5F5PS   F5F5F5F5       *5555555555555555*
0000E0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
0000F0       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 13. Breakpoint Display After Second Card is Processed

Note that a new record has been read. (Examine the memory, starting at address 000030, which corresponds to CARD in the program listing.) The input values were 2 and 3, and register 2 actually contains the correct sum.

At this point, if you type G, your program will complete execution. Instead, set a breakpoint on

```
BCR   B'1111',14
```

which is used to exit your program. Then reinitiate execution, which should cause the program to pause just before exiting.

Now, before terminating this exercise, you should examine the information available to you through the Trace command. This command can be used to display either the last 10 instructions executed or just the last 10 branch instructions executed. Type T, followed by I, to produce the trace of the last 10 instructions executed. Your screen should now be identical to the screen displayed in Figure 14.

```
PSW AT BREAK  FFC50000 9F00002E

R0-7 :  F4F4F4F4 00000033 00000005 00000003 F4F4F4F4 F4F4F4F4 F4F4F4FA F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 000000B0 000000F8 00000000

            ** TRACE OF INSTRUCTIONS JUST BEFORE TERMINATION **
        IM = PSW bits 32-39 (ILC,CC,MASK) before instruction decoded

     IM  LOCATION   INSTRUCTION
     ==  ========   ===========
     CF   00002A     47F0 F00C
     8F   00000C     4740 F02E
     8F   000010     5320 F030
     AF   000014     5331 0000
     AF   000018     1A23
     6F   00001A     5220 F081
     AF   00001E     E020 F080 000D
     EF   000024     EC00 F030 0050
     DF   00002A     47F0 F00C
     9F   00000C     4740 F02E      <-- Last instruction executed.

 RETURN to continue:
 :Trace:    Instruction Trace
```

Figure 14. Instruction Trace

Now type T, followed by B, to get a trace of just the last branch instructions. Your screen should now match the screen displayed in Figure 15.

```
PSW AT BREAK  FFC50000 9F00002E

R0-7 :  F4F4F4F4 00000033 00000005 00000003 F4F4F4F4 F4F4F4F4 F4F4F4FA F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 000000B0 000000F8 00000000

            ** TRACE OF LAST 10 INSTRUCTIONS EXECUTED **
        IM = PSW bits 32-39 (ILC,CC,MASK) before instruction decoded

     IM  LOCATION   INSTRUCTION
     ==  ========   ===========
     CF   00000C     4740 F02E
     CF   00002A     47F0 F00C
     8F   00000C     4740 F02E
     DF   00002A     47F0 F00C
     9F   00000C     4740 F02E

 RETURN to continue:
 :Trace:    Branch Trace
```

Figure 15. Branch Trace

Note that every branch instruction executed is displayed, even those branches that “failed.” That is, the first two instances of the instruction:

```
CF   00000C     4740 F02E
```

correspond to instances of the source instruction

```
CHECKEOF BC    B'0100',EXIT
```

which did not result in a branch to EXIT.

Once you have studied the trace information, use the Go command to complete execution of your program. Then make sure that you actually go back and correct the original program in the source file. Always remember that changes made temporarily during interactive debugging are not reflected in your actual program unless (and until) you use the editor to make them after the debugging run has been completed. (The listing for the program we have just described is in the `\BootAsst\` directory as file `DEMOB.PRT`.)

ASSIST/I User's Guide

This chapter is a reference manual for ASSIST/I. Each section presents guidance for the use of a facility that appears on the main menu when ASSIST/I is invoked.

Editing Programs

The ASSIST/I text editor is used to create new datasets and to change existing ones (programs, program data, documentation, etc.). It is a full-screen text editor modeled after MicroPro International's WordStar word processor, as used in non-document mode. The WordStar format was chosen because of its popularity and its adaptability to most terminals.

The editor comes with preset tab stops for easy writing of assembler programs. You are encouraged to use tabbing, for storage savings as well as convenience. The editor will not allow more than 79 characters (including the spaces taken up by expanded tabs) on a line.

Note

The editor program provided with ASSIST/I is flexible and can do many useful things, but it can be somewhat awkward to use. Programs and data for input to ASSIST/I can be prepared on host systems; tab characters are OK. Save your host files as variable-length (V-format) records, and then download them to your chosen workstation directory. Also, you can use any PC-based line editor to create and modify files.

Starting the Editor

The following four steps should be followed in all normal editing sessions:

1. Choose the `Edit a program` option in the ASSIST/I menu by typing `E`
2. To the `Enter name of file to edit:` prompt, type in the name of the file to be edited. A file name is a unique identifier assigned to a dataset. It can be any combination of characters that conforms to the host operating system's naming conventions. After typing the file name, enter `<cr>`. One of two possibilities will occur:
   - If a file with that name exists, it is brought into memory for editing, and its first full screen of lines is displayed.
   - If no file by that name exists, a new file is created.

   The validity of a file name depends on the operating system under which ASSIST/I is running. Most systems accept file names following these conventions:
   - You can use from 1 to 8 alphanumeric characters, with the first character being alphabetic.
   - This 1-to-8-character name can be followed by an optional 1-to-3-character alphanumeric suffix (extension), which is preceded by a period.
   - Special characters (punctuation, `$`, `%`, `@`, etc.) differ from system to system and should be avoided.

   An example that follows the conventions is

   ```
   ASSGT1.ASM
   ```
3. Type in the program, data, text, or whatever. View and modify the file, using the editor commands described in “Entering Editor Commands” on page 18.
4. Finally, to save the file and exit the editor, type ^KX. (See “Entering Editor Commands” on page 18 for an explanation of the characters “^KX”.) See also “More on the Editor: Block and File Commands” on page 22 for other file operations.

Entering Editor Commands

Editor commands are entered by holding down the Control key. The ^ symbol indicates that the Control (or Ctrl) key is to be held down while the letter immediately following the symbol is typed. ^K can be thought of as one keystroke that requires two keys to be depressed. The Control key is used much the same as a Shift key, which works with a letter to create its uppercase. In English, ^K is pronounced “Control K”, and ^QX is pronounced “Control QX.”

Assume that your file already exists (or has just been typed in) and that the file contains errors. To make corrections, you must be able to move the cursor around the screen to any position, insert and replace characters, delete characters, and save the corrected file. The edit commands that provide these facilities may be viewed by typing:

```
^U
```

This causes the Main Edit Menu to be displayed. This menu is shown in Figure 16.

```
   NAME=TEST.ASM          LINE=1   COL=1   ^U for HELP   INSERT= ON

            <<<<<<  M A I N   E D I T   M E N U  >>>>>>

    --Cursor Movement--     | -Delete-  |  -Miscellaneous-   | -Other  Menus-
^S char left ^D char right  | ^G  char  | ^V INSERT ON/OFF   | ^KU Block & File
^A word left ^F word right  | DEL chr lf| ^I (or TAB key) Tab| ^QU Quick
^E line up   ^X line down   | ^T word rt| RETURN to end line |
       --Scrolling--        | ^Y line   | ^N Insert a RETURN |
^Z line up   ^W line down   |           | ^L Repeat Find/Repl|
^C screen up ^R screen down |           |                    |

Enter any key to continue:
```

Figure 16. Editor Main Help Menu

Take a little time to study this menu. At first you will display it often as a reminder, when a basic editor command does not readily come to mind.

The less frequently used commands (that require two keystrokes) are found in the “Block & File” and “Quick” menus. They are described in “More on the Editor: Block and File Commands” on page 22.

Cursor Movement

The four most basic commands move the cursor one character to the right, one character to the left, up one line, or down one line. These four commands are:

^*S Char Left*	Moves the cursor one character to the left. If the cursor is on the first character of a line when ^S is issued, the cursor wraps to the last character of the previous line. ^S has no effect when the cursor is on the first character of the file.
^*D Char Right*	Moves the cursor one character to the right. If the cursor is on the last character of a line, the cursor is moved to the first column of the next line. ^D has no effect when the cursor is at the end of the file.
^*E Line Up*	Moves the cursor up one line. When the cursor is on the top line of the file, ^E has no effect.
^*X Line Down*	Moves the cursor down one line. When the cursor is on the last line of the file, ^X has no effect.

Note the location of these four letter keys on your keyboard. The four together form a “diamond”, and their relative positions in this picture indicate the direction of the cursor movement they effect. See Figure 17 on page 19.

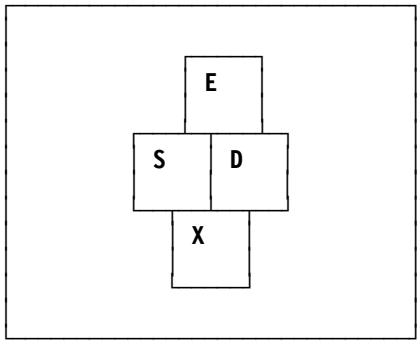


Figure 17. Editor Cursor-Movement Keys

Remember this configuration; you will frequently need these keys when using the text editor.

Two other commands move the cursor one word to the left or right:

^*A Word Left*	Moves the cursor one word to the left, unless the cursor is at the beginning of the file.
^*F Word Right*	Moves the cursor one word to the right, unless the cursor is at the end of the file.

These commands may be used to move the cursor through a line more efficiently than can be done with ^S and ^D. Check the relative positions of these two command keys on your keyboard, and note that these keys and the four command keys above still form a "diamond", as shown in Figure 18.

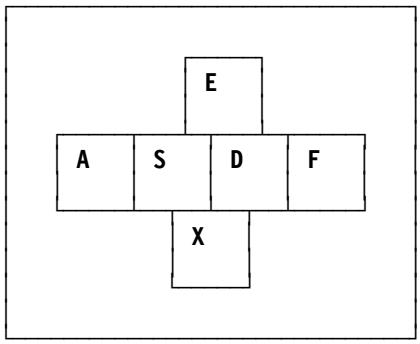


Figure 18. Editor Cursor-Movement Keys

The key to the left of the key that moves the cursor one *character* to the left moves the cursor one *word* to the left. A similar rule holds for moving the cursor to the right.

When text of any kind is entered, it should be checked on the screen to determine whether errors have been made. Errors such as the omission of words or the incorrect entry of a single character occur quite frequently and may easily be corrected using the facilities provided by the text editor.

Insert Mode

The editor functions in one of two modes when it is active. These modes are `Insert = On` and `Insert = Off` (or `Replace`). An editor command lets you toggle between these two modes:

^*V Insert On/Off*	Toggles between `Insert = On` and `Insert = Off` modes.

When this command is issued, the current Insert mode status is displayed at the top right of the screen. The toggle action of this command reverses the input mode. That is, if ^V is issued when Insert is On, then Insert is turned Off, and vice versa.

When Insert is on, each character typed is inserted at the cursor position, and the characters to the right of the cursor move one position to the right to make room for what is inserted. If the cursor line fills during an insertion, the terminal "bell" sounds, indicating that the line is full. When Insert is Off, a text character typed replaces the character at the cursor.

Deleting Data

You can delete an unwanted character from the text with the following text editor command:

^G Character at Cursor Deletes the character at the current cursor position. If the character is past the last text character on a line, the next line is wrapped up to the cursor position. However, if there is not enough room on the current line to fit the characters of the next line (since the ASSIST/I editor limits lines to 79 characters), the terminal "bell" beeps, and ^G does nothing.

Del Character before Cursor
The Del (Delete) key deletes the character to the left of the cursor. When the cursor is in column 1 and the Del key is pressed, the current line is wrapped up to the end of the previous line if the length of both lines is not greater than 79 characters. If the lines cannot fit together, the terminal "bell" sounds, and Del does nothing.

^T Del Word Deletes all characters over to the next word boundary. If ^T is issued when the cursor is past the last text character of a line, the effect is the same as that of ^G under the same circumstances.

^Y Del Line Deletes the entire cursor line.

Scrolling

The set of editor commands introduced above is sufficient for correcting any errors that occur on a single screen of text. However, most nontrivial files contain more lines than can be displayed on a single screen. To deal with longer files, it is necessary to scroll the screen through the text so that any portion of the file may be viewed. This facility is provided by the following two commands:

^C Screen Down Scrolls the screen down (forward in the file) approximately one full screen. (Not quite a full screen is scrolled, so you have a point of reference between ^C's.) ^C has no effect when the cursor is at the end of the file. Successive ^Cs can be issued without waiting for the screen to be rewritten.

^R Screen Up Scrolls the screen up (backward in the file) approximately one full screen, in the same manner that ^C scrolls forward. ^R has no effect when the cursor is at the beginning of the file. Successive ^Rs can be issued without waiting for the screen to be rewritten.

The only way to become proficient with any text editor is through practice. You are urged to enter some files and use the editor commands. The ASSIST/I editor is not restricted to the entry of programs or program data; it may be used to create *any* type of data file. Call the editor, create a file, and experiment with correcting errors and changing the contents of the file. You will probably be surprised how little practice is required to master the basic editor commands in this section. Remember that the help menu may be displayed at any time when using the editor by simply typing ^U.

Saving Files and Exiting the Editor

Files that have been created or edited usually should be saved. When an existing file has been edited, the Save command causes the edited version of the file to replace the existing, unedited file. There are occasions, however, when you may decide not to replace an existing file with an edited version. The following two commands accommodate these contingencies:

^KX Save & Exit — Saves the file being edited and returns control to the ASSIST/I supervisor.

^KQ Quit No Save — Ends the editing session and returns control to the ASSIST/I supervisor. Any changes made to the edited file are lost.

Only the basic editor commands have been discussed. Consult the help menu or the Users' Guide in "ASSIST/I User's Guide" on page 17 when questions arise.

Other Useful Editor Commands

^I (or Tab Key) — Tab moves the cursor to the next tab stop, according to the following rules:

1. If Insert is on, all characters at the cursor and to its right are moved to the next tab stop. A Tab causes the terminal to "beep" if there is no room for the insertion.
2. If Insert is off, the cursor is moved to the next tab stop without moving or disrupting any text characters.

Tab stops are preset to the following column positions:

`10, 16, 35, 40, 45, 50, 55, 60, 65, 70, and 75.`

Most terminals have a Tab key, which can be used interchangeably with ^I.

Return — The Return key moves the cursor to column 1 of the next line, according to the following rules:

1. When Insert is on, any characters at the cursor and to the right of it are forced down to create a new line.
2. When Insert is off, the cursor line is kept intact, and the cursor is moved to column 1 of the next line.

^N Insert Return — Inserts a Carriage Return (or, more accurately, a new-line or line-feed) at the cursor. The cursor remains in the same position as it was when ^N was typed, and the cursor character and all characters to its right (if any) are forced down to create a new line. ^N has the same effect as typing Return except that the cursor does not move.

^L Next Find & Replace

Performs the next Find or Find & Replace, according to the same criteria supplied in the last Find (^QF) or Find & Replace (^QA) command. This single-keystroke command saves having to retype the ^QF or ^QA commands (two-keystroke commands) and respond to the prompts when the criteria are the same. See the ^QA and ^QF descriptions in the "Quick Menu" shown in Figure 20 on page 23.

Other Menus

^KU Block/File Menu — Displays the Block & File Operations menu, described shortly. The ^K-prefixed commands in this menu allow copying, deleting, and moving text blocks, and exiting the editor.

^QU Quick Menu Displays the menu of the ^Q-prefixed Quick commands. These commands allow fast cursor movement within the file and fast deleting within a line. They also provide global Find and Find & Replace of text strings.

More on the Editor: Block and File Commands

Another useful set of editor commands allows you to manipulate entire blocks of text. That is, you can move, copy, or delete entire sequences of lines in a single operation. To see the menu of available block and file commands, simply type ^KU, which causes the menu shown in Figure 19 to appear. The desired menu option may be entered when this menu is displayed, or the space bar may be used to exit the menu without performing any operation.

Block & File Operations commands do not have to go through this menu. The desired option may be selected after ^K has been entered.

```
      <<<<<<  B L O C K  &  F I L E  O P E R A T I O N S  >>>>>>

                          BLOCK  OPERATIONS
=====================================================================
Set BEGIN block marker  ==>  B           Move block to cursor  ==>  V
Set END block marker    ==>  K           Copy block to cursor  ==>  C
Hide/Display marker(s)  ==>  H           Delete marked block   ==>  Y

                          FILE   OPERATIONS
=====================================================================
Write a block to a file ==>  W           Read a file to cursor ==>  R
Save & Exit             ==>  X           Quit NOSAVE           ==>  Q
```

Figure 19. Block and File Operations Help Menu

Block Operations

To manipulate a block of text, you must first mark the first (Begin) and last (End) characters in the block of text. (It is logical that the End-block marker comes after the Begin-block marker, and that both markers must be set correctly before the editor can perform any block commands.) The following commands are provided for this purpose:

^KB Set Begin Block Marker
Sets a Begin block marker to mark the beginning of a text block. To mark a block, place the cursor at the first character of that block, and type ^KB. The display of the text character is replaced by > to indicate the position of the beginning marker; the actual text character is not changed. There can be only one Begin marker in a file. If a Begin marker was active somewhere else in the file, it is moved when ^KB is issued.

^KK Set End Block Marker
Sets an End block marker to mark the end of a text block, replacing the displayed character with <. The actual text character is not changed. If the last character of a line (the line-feed) is to be marked, the cursor must be positioned at that character. (^QD is helpful for positioning the cursor at the end of a line.) There can be only one End marker in a file. If an End marker was active somewhere else in the file, it is moved when ^KK is issued. If for some reason the same character is marked for Begin and End, the replacement display character is *.

^KH Hide/Display Markers
This is a toggle used to hide and to re-display markers. Any attempt to perform a copy-, move-, or delete-block operation when markers are hidden results in an error message.

Markers can be repositioned to any point without being hidden first. The normal procedure is to mark the block, perform the block operation, and then hide the markers. The following rules apply to all the block operations:

1. Both markers must have been set and must not be hidden.
2. The End marker must occur after the Begin marker.
3. The cursor cannot be positioned on or between the markers.

Once a block of text has been marked, it can be moved, copied, deleted, or written to a disk file by using the following commands:

^KV Move Block to Cursor
: Moves a marked block to the position of the cursor.

^KC Copy Block to Cursor
: Copies a marked block to the cursor position.

^KY Delete Marked Block
: Deletes all characters in the marked block.

File Operations

^KW Write Block to File
: Writes the marked block of text to a disk file. You will be prompted for the name of the file to be created. This command creates a new file or, if the name of an existing file is given, overwrites the contents of the existing file. It does not append the block to the end of an existing file.

To retrieve the text from a disk file and insert it into the current text, you can use the following command:

^KR Read a File
: Reads the contents of a disk file, inserting the lines into the current text at the position of the cursor. The file being read is truncated if there is not enough available space in memory.

This last command can be quite useful when you wish to include the same sequence of lines in several files.

Note that when a file is saved, markers are not saved with it.

Advanced Features of the Editor: the QUICK Commands

You should become completely familiar with the basic commands before attempting to use these more advanced commands.

The Quick commands involve typing two characters: ^Q followed by a second character designating a specific operation. To see the menu displaying the available Quick commands, simply type ^QU, and the menu in Figure 20 will appear.

```
              <<<<<<  Q U I C K   M E N U  >>>>>>

                   ----- CURSOR MOVEMENT -----
Line LEFT (col 1)   ==>  S          Line RIGHT (end of line) ==>  D
TOP of screen       ==>  E          BOTTOM of screen         ==>  X
TOP of file         ==>  R          BOTTOM of file           ==>  C

                   -------- DELETES ----------

RIGHT (to end line) ==>  V          LEFT (to begin of line)  ==> DEL

                   ------ MISCELLANEOUS ------
FIND text in file   ==>  F          FIND & REPLACE text      ==>  A
```

Figure 20. QUICK Commands Help Menu

The desired menu option can be entered when this menu is displayed, or the space bar can be used to exit the menu without performing any operation.

Quick commands do not have to go through this menu. The desired option may be selected after the ^Q has been entered.

Cursor Movement

The most commonly used Quick commands are those for cursor movement. You have already used the basic cursor-control commands: ^S, ^D, ^E, ^X, ^R, and ^C. The Quick versions of these commands can be used to replace sequences of these basic commands. Their effects may be summarized as follows:

^QS Line Left	Moves the cursor to the first character (column 1) in the current line.
^QD Line Right	Moves the cursor to the last character of the current line (end of line).
^QE Top of Screen	Positions the cursor to the first character (column 1) in the first line on the screen.
^QX Bottom of Screen	Moves the cursor to the first character (column 1) of the last line on the screen.
^QR Top of File	Moves the cursor to the first character in the file.
^QC Bottom of File	Moves the cursor past the last character in the file.

As you edit increasingly larger files, you will find these commands for rapid movement of the cursor quite convenient.

Delete Operations

Two Quick commands are provided for deleting portions of the current line:

^QY Delete Right	Deletes the cursor character and all characters from there to the end of the line, except the “new-line” or “line-feed” character.
^QDel Delete Left	Deletes all characters in the current line to the left of the cursor. (Here ^QDel means holding down the control key and typing ^Q then depressing the Delete key on your terminal.)

These two commands can save time in deleting portions of sentences, removing the necessity of manually deleting a sequence of individual characters or words.

Search Operations

Occasionally, you will wish to scan a file for a particular string of characters. For example, you might wish to locate the string `' *'` to find a comment that began in column 2, rather than in column 1. In this case, you can use the following command:

^QF Find Text	Finds the first occurrence of a string, starting from the current position of the cursor. You are prompted for the string to be located and for whether to search forward or backward from the position of the cursor. (By specifying the option `B`, you can cause the scan to proceed backward from the position of the cursor; the default is to move forward.) A message is displayed if no occurrence of the string is found in the direction of the search. The cursor is positioned at the file's end or at its beginning, depending on the search direction.

To search the entire text for the string, it is frequently a good idea to use ^QR first, to position the cursor at the start of the text. Once you have located one occurrence, you may wish to continue the search by scanning for the next occurrence of the string. Do this by typing ^L.

The next Quick command can be used to replace one or more occurrences of a string by a replacement string. This command requires you to indicate whether you wish to be prompted to verify the replacement.

^QA Find/Replace Text Finds the next occurrence of a string and optionally replaces it with a specified replacement string. You are prompted for the search string, for the replacement string, and for options. The replacement options are

- `<cr>` Replace the next occurrence (a prompt asks whether or not you really wish the replacement to occur)
- `N` Make the replacement without prompting for permission
- `G` Replace all remaining occurrences of the string
- `GN,NG` Replace all remaining occurrences, without prompting for permission. In addition, you can specify the option B to cause the search to proceed backward from the current cursor position.

Just as with ^QF, ^L can be issued to repeat the last find-and-replace command.

ASSIST/I Pseudo-Instructions

ASSIST/I supports four pseudo-instructions for performing basic I/O and for processing simple numeric data: `XREAD`, `XPRNT`, `XDECI`, and `XDECO`. All pseudo-instructions can have a statement label, as noted in the instruction formats.

XREAD and XDECI Instructions

In the following discussion, we assume that the maximum length of an input record to be entered by the `XREAD` instruction is 80 characters. (This is due to the historical fact that, during the period in which IBM assembler became widely used, punched cards were the medium in which programs were encoded. Indeed, a majority of the example programs discussed here were first tested using versions of the programs punched into 80-column cards.)

The practice of using punched cards made terms like “source deck” (to refer to the cards that contained the symbolic version of a program) both natural and common. Therefore, you will find occasional references to punched cards throughout this book, and we trust it will be obvious that, in today's context, each of these references is to “an 80-character input record.”

Input records are normally of some fixed size. Here, most input records contain up to 80 characters of data. When these characters are read into storage, each will occupy one byte. There are 256 legitimate values that can be stored in one character of an input record. Each of these possible values is converted into a unique image when the record is read into storage. When punched cards were used as input, there were 256 unique patterns of punches for each column, each of which corresponded to hexadecimal values from `00` through `FF`. Most input records contain only “readable” characters which translate into a subset of the possible 256 values. The correspondence between the “readable” characters' hexadecimal values produced in storage, and how these were encoded in punched cards is illustrated in Figure 21.

Punches	Repres-enting	Image in Storage
none	blank	40
12-1	A	C1
12-2	B	C2
11-1	J	D1
11-2	K	D2
0-2	S	E2
0-3	T	E3
0	0	F0
1	1	F1
2	2	F2
3	3	F3
4	4	F4
5	5	F5
6	6	F6
7	7	F7
8	8	F8
9	9	F9

Figure 21. Sample Character Representations

Example: If an input record containing 1234 in the first four character positions is read into an area called CARD, the first byte at CARD will contain the value that corresponds to a 1. Since that is F1, the first byte at CARD is F1. The second will be F2; the third, F3; and the fourth, F4 Thus, the first four bytes at CARD would contain F1F2F3F4.

The DC (Define Constant) instruction is used to create constant data. Some examples for creating character data are described in "DC Instruction for Character Data" on page 39.

XREAD Instruction

The XREAD pseudo-instruction reads an input record into an area of storage from the source defined on the $ENTRY line. Data can be read from:

1. The console
2. Lines immediately following $ENTRY
3. A disk file.

Its format is:

```
label    XREAD addr,length
```

where

`addr` is the address, in D(X,B) or implicit form, of the area into which the input should be read

`length` is the number of characters to be read.

- length should be 80 or less. If it is greater than 80, at most 80 characters will be read.
- If the number of characters in the record is fewer than length, the extra characters will be filled with blanks.
- If length is less than the number of characters in the record, the remaining characters of the input record will be ignored.
- If length is omitted, the length of the addr operand is used if easily determined; otherwise 80 is used.

Each XREAD statement reads exactly one line of input data. The data obtained by XREAD is stored in character format at the D(X,B) location.

Execution of an XREAD statement after the end-of-file condition has occurred results in a *Read Past End of File* program exception.

The encoded form of XREAD resides in 3 halfwords.

Section "The $ENTRY Record" on page 33 describes how to specify the input data source.

An example:

```
XREAD CARD,40
```

will read 40 characters from the next input record and place the first character at CARD+0, the second at CARD+1, and so on.

The settings of the CC after the execution of an XREAD instruction convey the following information:

CC	Meaning
0	The record was read successfully.
1	No record could be read. This is called an end-of-file or EOF condition.
2	— (The CC is never set to this value.)
3	—

Note: A dash is used to indicate condition code settings that cannot occur.

XPRNT Instruction

Just as data that is read in from a record is stored in character format, so too is data that is to be printed. The XPRNT statement provides an easy way to print program output lines. XPRNT lines appear in the program listing (file type .PRT) just after the assembly listing.

There are two steps involved in printing a line:

1. The line to be printed must be constructed in an area of storage. The actual characters that will appear in the printed line begin in the second byte of the area. The initial byte is reserved for a carriage control character, a byte that controls the positioning of the printed line on a page.

   The codes that can be used in the first byte and their meanings are

   ```
   blank   Single-space before printing
       0   Double-space before printing
       -   Triple space before printing
       +   Suppress spacing (overprint)
       1   Skip to the head of the next page before printing.
   ```

   Carriage-control bytes containing non-printable values are treated as if they were blanks.

   The entire print line (including the carriage control byte) must contain 133 or fewer characters, because there are only 132 print positions on most printers.

2. After the print line has been constructed, an XPRNT instruction can be used to print the line. The format of the XPRNT instruction is

   ```
   label    XPRNT addr,length
   ```

   where

   addr — is the D(X,B) or implicit address of the print line in storage

   length — is the number of bytes to be taken from storage in constructing the record to be sent to the printer. The length should be 133 or fewer characters, including the carriage-control character. (Bad things happen if it's longer!) If no length operand is specified, it is taken from the length attribute of the addr operand.

XPRNT does not alter the condition code. The encoded form of XPRNT resides in 3 halfwords.

Examples:

```
XPRNT =C'1THIS IS A PAGE HEADER',22
```

will print the line THIS IS A PAGE HEADER at the top of the next page.

```
XPRNT CARD-1,80
```

will print a line beginning with the byte before CARD for a length of 80 bytes. The carriage control byte would be at CARD-1.

XREAD/XPRNT Sample Program

The sample program in Figure 22 illustrates uses of the XPRNT instruction. The program reads any number of records, printing one line per record. The first line will be printed at the top of a page, and all successive lines will be double-spaced.

```
*********************************************************
* THIS PROGRAM READS AND PRINTS CARDS UNTIL EOF OCCURS.
*********************************************************
PRINT  CSECT
       USING PRINT,15
*
       XREAD CARD1,80       READ THE FIRST CARD
       BC    B'0100',EXIT   EXIT ON EMPTY FILE
*
       XPRNT CC1,81         PRINT THE FIRST CARD
       XREAD CARD2,80       NOW READ THE SECOND CARD
LOOP   BC    B'0100',EXIT   BRANCH ON EOF
*
       XPRNT CC2,81         PRINT THE CARD
       XREAD CARD2,80       TRY TO READ THE NEXT CARD
       BC    B'1111',LOOP   GO BACK TO TEST FOR EOF
*
EXIT   BCR   B'1111',14     LEAVE THE PROGRAM
*
CC1    DC    CL1'1'         CAUSE SKIP TO TOP OF PAGE
CARD1  DS    CL80           INPUT AREA FOR FIRST CARD
CC2    DC    C'0'           DOUBLE SPACE THE REST
CARD2  DS    CL80           ALL BUT FIRST CARD GET READ HERE
       END  PRINT
$ENTRY
 Record 1
  Record 2
   Last record.
```

Figure 22. XREAD/XPRNT Sample Program to Read and Print Cards

This program is in the \BootAsst\ directory as file DEMOC.ASM, and the following listing shown in Figure 23 on page 29 is in the \BootAsst\ directory as file DEMOC.PRT.

```
ASSIST/I Version 2.03, Copyright 1984, BDM Software.

  LOC  OBJECT CODE    ADDR1 ADDR2   STMT    SOURCE STATEMENT

                                     1 ********************************************************
                                     2 * THIS PROGRAM READS AND PRINTS CARDS UNTIL EOF OCCURS.
                                     3 ********************************************************
000000                               4 PRINT  CSECT
000000                               5        USING PRINT,15
                                     6 *
000000 E000 F02D 0050 0002D          7        XREAD CARD1,80       READ THE FIRST CARD
000006 4740 F02A            0002A    8        BC    B'0100',EXIT   EXIT ON EMPTY FILE
                                     9 *
00000A E020 F02C 0051 0002C         10        XPRNT CC1,81         PRINT THE FIRST CARD
000010 E000 F07E 0050 0007E         11        XREAD CARD2,80       NOW READ THE SECOND CARD
000016 4740 F02A            0002A   12 LOOP   BC    B'0100',EXIT   BRANCH ON EOF
                                    13 *
00001A E020 F07D 0051 0007D         14        XPRNT CC2,81         PRINT THE CARD
000020 E000 F07E 0050 0007E         15        XREAD CARD2,80       TRY TO READ THE NEXT CARD
000026 47F0 F016            00016   16        BC    B'1111',LOOP   GO BACK TO TEST FOR EOF
                                    17 *
00002A 07FE                         18 EXIT   BCR   B'1111',14     LEAVE THE PROGRAM
                                    19 *
00002C F1                           20 CC1    DC    CL1'1'         CAUSE SKIP TO TOP OF PAGE
00002D                              21 CARD1  DS    CL80           INPUT AREA FOR FIRST CARD
00007D F0                           22 CC2    DC    C'0'           DOUBLE SPACE THE REST
00007E                              23 CARD2  DS    CL80           ALL BUT FIRST CARD GET READ HERE
                                    24        END  PRINT

***   0 STATEMENTS FLAGGED - 0 WARNINGS, 0 ERRORS

*** PROGRAM EXECUTION BEGINNING -
ANY OUTPUT BEFORE EXECUTION COMPLETE MESSAGE IS PRODUCED BY USER PROGRAM ***

  Record 1

  Record 2

   Last record.

*** EXECUTION COMPLETED ***
```

Figure 23. Listing of Program to Read/Add Numbers

XDECI Instruction

The XDECI pseudo-instruction converts numbers in their character representation in storage to their corresponding binary representation in a register. Its format is:

```
label    XDECI  reg,addr
```

where

reg — is the number of the general register into which the binary form of the number will be inserted.

addr — is the D(X,B) or implicit address of the number in its character format.

Execution of an XDECI instruction has the following effects:

1. Beginning at the location given by addr, memory is scanned for the first character that is not a blank.
2. If the first character found is anything other than a decimal digit or a plus or minus sign, R1 is set to the address of that character and the condition code is set to 3 to show that no decimal number could be converted. The contents of register reg are not changed, and nothing more is done.
3. If the first character is a plus or minus sign or a decimal digit, from one to nine decimal digits are scanned, and the number is converted to binary and placed in reg.
4. If ten or more decimal digits are found before a blank separator, R1 is set to the address of the first character that is not a decimal digit, the CC is set to 3, and reg is left unchanged. A plus or minus sign alone causes a similar action, with R1 set to the address of the character following the sign character.

The encoded form of XDECI resides in 2 halfwords.

R1 is set to the address of the first non-digit after the string of decimal digits. Thus, reg should not usually be 1. This lets you scan for any number of decimal values. The values should be separated by blanks.

Remember that execution of XDECI alters R1. Therefore, don't expect to save a value in R1 if you issue XDECI.

The CC is set by XDECI as follows:

CC	Meaning
0	The number converted was 0
1	The number converted was < 0
2	The number converted was > 0
3	An attempt was made to convert an invalid number

The scan performed by the execution of an XDECI instruction will continue, perhaps beyond the field to be scanned, until a non-blank character is encountered. Thus, if the image of an input record is to be scanned, it is advisable to mark the end of the image with a non-blank character other than a decimal digit, so that the condition code may be checked to determine when the end of the image has been reached. For example, this could be done as follows:

```
CARD     DS    CL80
         DC    C'*'
```

Assuming that the input cards contained one or more numbers and that the above marker technique was used, the following code segment can serve as an example:

```
         SR    R3,R3
         XDECI R5,CARD          GET 1st NUMBER
LOOP     BC    B'0001',EXITLUP  EXIT NONE LEFT
         ST    R5,TABLE(R3)     SAVE THE NUMBER
         LA    R3,4(,R3)        INCREMENT INDEX
         XDECI R5,0(R1)         GET NEXT NUMBER
         BC    B'1111',LOOP
```

XREAD/XDECI Sample Program

The little program in Figure 24 on page 31 reads input records with two numbers on each record. Each number and the difference between the numbers is XPRNTed. (This program is in the \BootAsst\ directory as file DEMOD.ASM.)

```
*****************************************************************
*  THIS PROGRAM READS NUMBERS (TWO PER CARD) AND XPRNTS THEM AND
*  THEIR DIFFERENCE.
*****************************************************************
DIFF     CSECT
         BALR  9,0
         USING *,9
*
         XREAD CARD,80           READ THE FIRST CARD
LOOP     BC    B'0100',EXIT      BRANCH IF EOF HAS OCCURRED
         XDECI 2,CARD            GET THE FIRST NUMBER ON THE CARD
         BC    B'0001',GETNXT    SKIP THIS CARD ON A BAD VALUE
         XDECO 2,NUMBER1         PUT INTO PRINT LINE
         XDECI 3,0(1)            NOW GET THE SECOND NUMBER
         BC    B'0001',GETNXT    SKIP THIS CARD ON A BAD VALUE
         XDECO 3,NUMBER2         PUT INTO PRINT LINE
         SR    2,3               GET THE DIFFERENCE
         XDECO 2,DIFFRNCE        FORMAT THE DIFFERENCE
         XPRNT LINE,LINEL        PRINT THE RESULTS
GETNXT   XREAD CARD,80           TRY TO READ THE NEXT CARD
         BC    B'1111',LOOP      GO BACK UP TO TEST FOR EOF
*
EXIT     BCR   B'1111',14        EXIT FROM THE PROGRAM
*
CARD     DS    CL80              CARD INPUT AREA
LINE     DC    C'ONUM1='         BEGINNING OF PRINT LINE
NUMBER1  DC    CL12' '           SPACE FOR FIRST NUMBER
         DC    C', NUM2='        CONTINUATION OF LINE
NUMBER2  DC    CL12' '           SPACE FOR SECOND NUMBER
         DC    C', DIFF='        CONTINUATION OF LINE
DIFFRNCE DC    CL12' '           SPACE FOR DIFFERENCE
LINEL    EQU   *-LINE            LINE LENGTH
         END   DIFF
$ENTRY
 2  6
 9 35
```

Figure 24. XREAD/XDECI Sample Program

(The listing shown below in Figure 25 for this program is in the \BootAsst\ directory as file DEMOD.PRT.)

```
ASSIST/I Version 2.03, Copyright 1984, BDM Software.

  LOC  OBJECT CODE    ADDR1 ADDR2   STMT    SOURCE STATEMENT

                                      1 ********************************************************************
                                      2 *  THIS PROGRAM READS NUMBERS (TWO PER CARD) AND PRINTS THE
                                      3 *  DIFFERENCE OF THE TWO NUMBERS.
                                      4 ********************************************************************
000000                                5 DIFF     CSECT
000000 0590                           6          BALR  9,0
000002                                7          USING *,9
                                      8 *
000002 E000 903A 0050 0003C           9          XREAD CARD,80           READ THE FIRST CARD
000008 4740 9038            0003A    10 LOOP     BC    B'0100',EXIT      BRANCH IF EOF HAS OCCURRED
00000C 5320 903A            0003C    11          XDECI 2,CARD            GET THE FIRST NUMBER ON THE CARD
000010 4710 902E            00030    12          BC    B'0001',GETNXT    SKIP THIS CARD ON A BAD VALUE
000014 5220 9090            00092    13          XDECO 2,NUMBER1         PUT INTO PRINT LINE
000018 5331 0000            00000    14          XDECI 3,0(1)            NOW GET THE SECOND NUMBER
00001C 4710 902E            00030    15          BC    B'0001',GETNXT    SKIP THIS CARD ON A BAD VALUE
000020 5230 90A3            000A5    16          XDECO 3,NUMBER2         PUT INTO PRINT LINE
000024 1B23                          17          SR    2,3               GET THE DIFFERENCE
000026 5220 90B6            000B8    18          XDECO 2,DIFFRNCE        FORMAT THE DIFFERENCE
00002A E020 908A 0038 0008C          19          XPRNT LINE,LINEL        PRINT THE RESULTS
000030 E000 903A 0050 0003C          20 GETNXT   XREAD CARD,80           TRY TO READ THE NEXT CARD
000036 47F0 9006            00008    21          BC    B'1111',LOOP      GO BACK UP TO TEST FOR EOF
                                     22 *
00003A 07FE                          23 EXIT     BCR   B'1111',14        EXIT FROM THE PROGRAM
                                     24 *
00003C                               25 CARD     DS    CL80              CARD INPUT AREA
00008C F0D5E4D4F17E                  26 LINE     DC    C'0NUM1='         BEGINNING OF PRINT LINE
000092 4040404040404040              27 NUMBER1  DC    CL12' '           SPACE FOR FIRST NUMBER
00009E 6B40D5E4D4F27E                28          DC    C', NUM2='        CONTINUATION OF LINE
0000A5 4040404040404040              29 NUMBER2  DC    CL12' '           SPACE FOR SECOND NUMBER
0000B1 6B40C4C9C6C67E                30          DC    C', DIFF='        CONTINUATION OF LINE
0000B8 4040404040404040              31 DIFFRNCE DC    CL12' '           SPACE FOR DIFFERENCE
000038                               32 LINEL    EQU   *-LINE            LINE LENGTH
                                     33          END   DIFF

***   0 STATEMENTS FLAGGED - 0 WARNINGS, 0 ERRORS

*** PROGRAM EXECUTION BEGINNING -
ANY OUTPUT BEFORE EXECUTION COMPLETE MESSAGE IS PRODUCED BY USER PROGRAM ***


NUM1=           2, NUM2=           6, DIFF=          -4

NUM1=           9, NUM2=          35, DIFF=         -26

*** EXECUTION COMPLETED ***
```

Figure 25. Listing of Program to Read/Add Numbers

XDECO Instruction

XDECO is used to convert a binary number in a register to its printable decimal equivalent. Once converted, the number can be printed with XPRNT. The format of the XDECO statement is

```
label    XDECO reg,addr
```

where

reg — is the number of the general register that contains the binary number to be converted. This register number is not altered by XDECO.

addr — is the D(X,B) or implicit address in which to place the 12-byte character representation of the number.

XDECO does not alter the condition code. The encoded form of XDECO resides in 2 halfwords.

Execution of this pseudo-instruction causes the number in the register given as the first operand (reg) to be converted to a 12-byte character representation and stored at the D(X,B) or implicit address given by the second operand (addr). The contents of the register are unaltered.

For example, execution of the instruction

```
XDECO 10,ANSWER
```

causes the binary number in R10 to be converted to its character format and the result to be stored in 12 bytes of storage, starting at ANSWER. The contents of R10 are unaltered.

The number will be right-justified in the 12-byte field, with leading blanks. A minus sign will be printed to the left of the first significant digit if the number is negative.

XDUMP Instruction

The XDUMP pseudo-instruction was intentionally left out of ASSIST/I because of the interactive debugger. With the capability of single-stepping and setting breakpoints, you should have no need to take memory snapshots.

XSAVE Instruction

The XSAVE pseudo-instruction can be used to generate the instructions to set up a base register and establish a register save area. The only form of the XSAVE instruction supported by ASSIST/I is

```
label   XSAVE BR=reg[,SA=NO]
```

where the label is required, and reg must be a number in the range 3-12. The SA=NO option, which should be specified only for lowest-level subroutines, suppresses the generation of a save area for the routine. Thus,

```
XSAVE BR=12
```

can be used to establish register 12 as a base register and to establish a save area, while

```
XSAVE BR=12,SA=NO
```

establishes register 12 as a base register but does not generate a save area for the routine.

If no BR operand is specified, register 12 will be set as a base register.

XRETURN Instruction

The XRETURN instruction can be used to generate the code to exit from a routine. It supports a single optional operand, to indicate whether or not a save area was established for the routine. Thus,

```
XRETURN
```

could be used to exit from a routine for which a register save area was established, while

```
XRETURN SA=NO
```

would be used for a lowest-level routine in which no save area was needed.

The $ENTRY Record

When using XREAD, you must indicate where the input lines are to be found. $ENTRY tells ASSIST/I where to look for a program's input (XREAD) data. Depending on the $ENTRY parameter, input data can be read from one of three sources:

1. Include the input lines immediately following the $ENTRY line, which must immediately follow the last line of your program.
2. A second method is to enter the input lines interactively. To request this option, use the following form of the $ENTRY line:

   ```
   $ENTRY CONSOLE
   ```

   As the program is executing, it prompts you to enter input lines whenever an XREAD instruction is executed.
3. You can request that your input lines be read from a separate disk file. This makes it convenient to create a common file of test data that can be shared by multiple users by simply copying the data file. To invoke this option, you simply specify the file name on the $ENTRY statement.

The first option is the most commonly used.

$ENTRY is coded immediately following the program's END statement. It must conform to one of the following formats:

1. $ENTRY with no parameters is used for programs that will read in-stream data (data coded on the lines following $ENTRY) or for programs not requiring data. This allows you to edit your program and input data in a single file. However, you must re-edit the file every time you wish to change the input data.
2. $ENTRY CONSOLE causes XREAD data to be obtained from the console. ASSIST/I issues the prompt:

   ```
   ENTER XREAD DATA NOW (ENTER CONTROL Z FOR EOF)
   ```

   When this prompt is displayed, you can type in any desired input line, terminated by a <cr>. ^Z is used to indicate end-of-file from the console.

   This method allows you to rapidly retest your program using a variety of different sets of input data.
3. $ENTRY CONSOLE NOPROMPT works the same as the $ENTRY CONSOLE, except that the prompt to enter data from ASSIST/I is not issued.

   Care should be taken when using the NOPROMPT option. You should XPRNT a prompt from the program before executing XREAD, so the program user knows what is expected by the program.
4. $ENTRY file name causes XREAD to take data from the named file. For example,

   ```
   $ENTRY  B:PROG1.DAT
   ```

   specifies that the input data is to be read from the file PROG1.DAT (located on disk drive B).

   If the file does not exist, a message is issued, and any XREAD data are obtained as if $ENTRY CONSOLE had been coded.

Note: A program that does not have a $ENTRY record will assemble but not execute. An attempt to read data past the end of file results in abnormal program termination.

Running Programs

There are two options in the ASSIST/I main menu that can be used to run programs: the R (Run) option and the F (Final Run) option. The F option is discussed shortly.

Providing the Save Output Listing option is on (see section “Altering ASSIST/I Options” on page 38), both the R and F options create a print file with the same name as the program being run, but with a .PRT file type. If a file with the same name exists, it will be overwritten. For this reason, source programs should *never* be given a name with the format filename.PRT.

The R option is used for program debugging. It invokes the ASSIST/I two-pass assembler to assemble the source code and generate the object code. If a program does not assemble correctly, control is returned to the ASSIST/I main menu. When a program does assemble correctly and a valid $ENTRY was included, execution control is passed to the program.

With the R option, an automatic breakpoint is set on the program's first instruction, to invoke the ASSIST/I debugger before any instructions have executed. The user can then decide whether to execute multiple statements via the debugger's Go option or to execute one instruction at a time with the debugger's Step option.

Making Final Runs

ASSIST/I provides a facility for making final program runs through the selection of the F option of the ASSIST/I main menu. The option assembles and executes programs but does not provide access to the debugger. For user convenience, the F option prompts for the user's name. The name is printed with the ASSIST/I output-page header.

Printing Programs and Listings

The Print a Program (P) option in the ASSIST/I main menu can be used to print programs created by the editor and assembly/program output listings (listings with .PRT file types). Tab characters are expanded to the same positions as the editor expands them. (See ^I in section "Other Useful Editor Commands" on page 21.)

The Print option writes to the operating system's assigned list device.

Using the ASSIST/I Debugger

The ASSIST/I debugger allows you to view a program's status during the execution phase. The debugger is invoked at execution time only under one of the following four conditions:

1. When control is initially passed to the program
2. Upon encountering a user-defined breakpoint (set by the B(rkpt.) facility described below)
3. On most implementations, when a key is struck during program execution
4. When a program exception (abnormal termination) is encountered.

Figure 26 is an example of a screen display by the debugger when it is first invoked (i.e., the automatic break before control is passed to the program for execution).

```
PSW AT BREAK   FFC50000 0F000000

R0-7 :  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4
R8-15:  F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 F4F4F4F4 00000020 00000068 00000000

000000       5810F010   5820F014   1A125010   F01807FE       *..0...0...&.0...*
000010       00000004   00000006   F5F5F5F5   F5F5F5F5       *........55555555*
000020       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*
000030       F5F5F5F5   F5F5F5F5   F5F5F5F5   F5F5F5F5       *5555555555555555*

      ... etc. ...

===>  B(rkpt.), D(ump), G(o), M(emory), P(SW), Q(uit), R(eg.), S(tep), T(race)
:
```

Figure 26. Screen Prior to Execution, Showing Debug Options

At the bottom of the screen, the line with the arrow at the left lists the primary debugger options available to the user. These options are explained as follows:

- B(reakpoint)

  The B(rkpt.) option provides a way to dynamically set, clear, and display program breakpoints (addresses where the debugger will automatically be called). The following prompt will be issued:

  ```
   BREAKPOINT:   S(et), C(lear), D(isplay)
  :Breakpoint: _
  ```

  where

  S(et) Prompts for the hex address at which to set a breakpoint. Leading zeroes of an address need not be entered. A breakpoint is executed only when it is set at an instruction's beginning address (the opcode byte).

C(lear) Clears an individual breakpoint or all breakpoints, depending on the response to the following prompt:

```
 Breakpoint CLEAR: I(ndividual), A(ll)
:Breakpoint: Clear: _
```

where

I prompts for the address of the breakpoint to clear.
A clears all the program breakpoints currently set.

D(isplay) Displays the first 140 program breakpoint addresses.

Return exits the breakpoint display.

- D(ump)

  The D option is selected to write an ASSIST/I dump to the user's output-listing file. The dump is representative of the PSW, registers, and memory at the time the D option was selected. The following confirmation prompt is issued:

  ```
  DUMP user Memory to disk and QUIT Debugger? (Y/N):
  ```

  Note: Responding with Y stops program execution, exits the debugger, writes the dump to disk, and returns control to the ASSIST/I menu.

- G(o)

  The G option simply causes the debugger to exit and return control to program execution. Execution is resumed at the instruction address located in the PSW.

- M(emory)

  The M option allows the user to display and alter the contents of any memory location within the user area. This allows the user to make most program changes without having to reassemble the program. After the program area is modified, the PSW can optionally be reset with the P(SW) option, and execution can be resumed via the G or S option.

  The following prompt is issued for the M option:

  ```
  MODIFY MEMORY: <RET> for this screen,  N(ext scr.), P(rev. scr.), (hex address)
  : Modify: _
  ```

  where

  <Ret> (depressing the Return key) moves the cursor to the first byte of the current screen. The cursor may then be positioned via the "Modify Memory" commands (described below), and the valid hex digits may be entered at any location.

  N(ext screen) Displays the next full screen of memory-dump lines, beginning with the line that follows the bottom line displayed on the current screen, and positions the cursor on the first byte.

  P(revious screen) Displays the screen-full of memory-dump lines located before the current screen's top memory-dump line, and positions the cursor at its first byte.

  (Hex address) Allows an address to be entered by simply entering a valid hex address. It is assumed that a hex address is being entered when a valid hex digit (0-F) is typed at this prompt level. The screen dump is then updated if necessary, and the cursor is positioned at the specified address.

  Modify Memory Commands

  In memory modification, any valid hexadecimal digit may be typed to replace the hex digit at the cursor location. As new hex digits are typed, the EBCDIC display at the right of the screen is updated to reflect the correct translated character.

  The cursor can be positioned within the hexadecimal memory-display portion via the following sequences:

  ^*S or Backspace* Moves the cursor left to the previous hex digit.

^D or Space Bar	Moves the cursor right to the next hex digit.
^A	Moves the cursor left to the previous fullword boundary.
^F	Moves the cursor right to the next fullword boundary.
^E	Positions the cursor up one line.
^X	Positions the cursor down one line.

Return is used to exit memory modification.

Note: The M option can be used to display memory by requesting the location to modify and then immediately exiting via Return.

- P(SW)

  The P(SW) option allows the PSW to be modified. It positions the cursor at the second byte of the address portion of the PSW's instruction address. The cursor may then be positioned anywhere within the last four bytes of the PSW. ^S is used to position left, and ^D is used to position right.

  Only hex digits are accepted as replacement data. Return is used to exit the P(SW) modification.

- Q(uit)

  The Q(uit) option causes the following prompt to be issued:

  ```
  Quit ASSIST Debugger & return to main menu? (Y/N):
  ```

  Answering Y causes the debugger to exit to the ASSIST/I main menu. Program execution is terminated.

- R(eg)

  The R option provides the capability of altering the contents of any of the general registers. A prompt asks you to enter the register number. Register numbers 0 through 15 are accepted, and the cursor is positioned at the register's leftmost byte.

  Only hex digits are accepted as replacement data. Return is used to exit the R modification.

- S(tep)

  The S option causes the next instruction to be executed, without leaving the debugger. Single-step has the same effect as setting a temporary breakpoint at the next instruction.

- T(race)

  The T option is selected to display the ASSIST/I branch trace or instruction trace. The following second-level prompt is issued:

  ```
   TRACE:  B(ranch Trace), I(nstruction trace)
  :Trace: _
  ```

  where

`B(ranch trace)`	This selection displays a trace of the last branch instructions that were executed before the interrupt that invoked the debugger. As many as 10 branches can be displayed with the PSW bits 32-39 (the Instruction Length Code, Condition Code, and Program Mask), the location of the branch instruction, and the encoded form of the instruction. Return exits the branch trace and refreshes the screen dump.
`I(nstruction trace)`	This selection displays a trace of the last 10 instructions that were executed before the interrupt that caused the editor to be invoked. The instructions are displayed with the PSW bits 32-39 (the Instruction-Length Code, Condition Code, and Program Mask), the instruction location, and the encoded form of the instruction. Return exits the instruction trace and refreshes the screen dump.

Figure 27 on page 38 is an example of the instruction trace display.

```
     ** TRACE OF LAST 10 INSTRUCTIONS EXECUTED **
IM = PSW bits 32-39 (ILC,CC,MASK) before instruction executed at LOCATION.

     IM  LOCATION   INSTRUCTION
     ==  ========   ===========
     20   0000D4     4570 F10A
     A0   0001BA     F3F7 F1E1 F1F1
     E0   0001C0     96F0 F1F0
     90   0001C4     E020 F1CC 0025
     D0   0001CA     07F7
     50   0000D8     F3F7 F1C0 F201
     D0   0000DE     96F0 F1DC
     90   0000E2     47F0 F11E
     90   00011E     47F0 F12E
     90   00012E     FCB7 F122 F201 <-- Last instruction executed.
```

Figure 27. Example of Instruction Trace Display

The last instruction in the list was executed just before the debugger was called. The cause of termination is probably either this instruction or the fact that a breakpoint had been set on the instruction following it.

Altering ASSIST/I Options

ASSIST/I provides the facility to alter program execution options by selecting the “A” option in the ASSIST/I main menu. The example screen displayed in Figure 28 lists the items that are alterable.

```
                A S S I S T / I  Options

          1) Save output listing       - y
          2) Maximum # lines           - 500
          3) Maximum # instructions    - 5000
          4) Maximum # pages           - 100
          5) Maximum size (in bytes)   - 2700


        Enter option number to alter (RETURN to quit): _
```

Figure 28. ASSIST/I Execution Options

The meaning of each option selection is:

Option	Description
1	Is a toggle to determine whether or not the output listing (source code listing and program-generated output) are to be written to an output disk file. If the option is `n`, no listing is generated; `y` insures that an output listing is generated, with a file name the same as the source code file, but with file type .PRT.
2	Indicates the maximum number of lines a program output listing is allowed to contain. The output listing includes source-code lines and program-generated lines (lines generated by the `XPRNT` statement).
3	Indicates the maximum number of instructions ASSIST/I can execute. A program that tries to execute more instructions is terminated. (This is handy for dealing with endless loops.)
4	Indicates the maximum number of pages an output listing may contain. Since this number includes the source listing and `XPRNT`-generated pages, it must be set high enough to accommodate both.
5	Reserves storage for the program's object code. This value may need adjustment, depending on the system ASSIST/I is running under.

Other Helpful Information

This chapter contains additional notes that may help with preparing and understanding programs.

DC Instruction for Character Data

The DC statement can be used to generate character constants. The format of the appropriate DC statement is

```
label    DC    mCLn'character string'
```

where

`m` is a duplication factor (a non-negative integer)

`Ln` gives the length `n` of the constant to be generated

`character string`
is a string of characters

For example, the instruction

```
F1       DC    2CL3'A B'
```

causes two consecutive three-byte fields to be generated, each containing `C140C2` (the hexadecimal representation). The following additional rules apply to this use of the DC statement:

1. If the duplication factor is omitted, it is assumed to be 1. Hence,

   ```
            DC     CL3'*A*'
   ```

   and

   ```
            DC    1CL3'*A*'
   ```

   generate the same thing.

2. If the length that is specified is greater than that required to hold the character string, blanks will be padded on the right. Thus,

   ```
            DC    CL6'*A*'
   ```

   would generate `5CC15C404040`.

3. If the specified length does not allow enough bytes to represent all of the characters in the character string, the rightmost characters will be truncated. For example,

   ```
            DC    CL2'*A*'
   ```

   would generate the two-byte field `5CC1`.

4. If the length is unspecified, a field exactly large enough to represent the character string will be generated. Therefore,

   ```
            DC    2C'ABC'
   ```

   and

   ```
            DC    2CL3'ABC'
   ```

   each generate two three-byte fields.

5. There are two special characters, `'` (apostrophe or single quote) and `&` (ampersand), that are unique in the following respect: to generate a string containing either an `'` or an `&`, *two adjacent occurrences* of the character must occur in the character string, rather than a single occurrence. Thus,

   ```
            DC    C'A''B'
   ```

   generates the three-byte field `C17DC2`, and

   ```
            DC    C'&&'
   ```

   generates just one byte containing `50`.

The following examples should help to clarify the above comments:

```
*              Coded            Generated
         DC    CL1'O'           F0
         DC    CL2'O'           F040
         DC    2CL2'89'         F8F9F8F9
         DC    C'WORD'          E6D6D9C4
         DC    C'&&AB''C'       50C1C27DC3
         DC    CL3'ABCD'        C1C2C3
```

Continued Statements

Sometimes a statement will be too long to fit conveniently on one line. (Remember that it may occupy only columns 1-71.) If you must create a longer statement, do the following:

1. Enter the initial part of the statement that fits in columns 1-71.
2. Put a non-blank (and easily recognizable) character in column 72. This is called the *continuation character*.
3. Continue your statement in column 16 of the next line; columns 1-15 *must* be blank. This is a *continuation statement*.
4. If the rest of the statement still won't fit, go back to step 2.

Normally, the character in column 16 of a continuation statement will be non-blank. However, if you are continuing a quoted string, a blank that is part of that string may validly appear in column 16.

For example: suppose you define a long character constant (the `1...:....|10..` is there just to help you see the column positions; it isn't part of your constant!):

```
1...:....|10..:....|20..:....|30..:....|40..:....|50..:....|60..:....|70

LongC    DC    C'1This is an extremely long character string that we wi*
               sh to appear at the top of a page when it is printed.'
```

The `'*'` character in column 72 is the continuation character.

If we modify the text slightly so that a blank appears where the continuation character is required, the statement might look like this:

```
1...:....|10..:....|20..:....|30..:....|40..:....|50..:....|60..:....|70

LongC    DC    C'1This is an extremely long character string that is to*
                appear at the top of a page when it is printed.'
```

In this case, the blank after the words “is to” is part of the character string, so its presence in column 16 of the continuation statement is correct.

Host-System Macros

Several sets of macros can help you with simple tasks like data conversions, dumping, and I/O. These macros should be extracted from the diskette or CD-ROM files, uploaded to your host system as fixed-length 80-byte records, and added to your macro library. That library must then be available to the High Level Assembler when your programs are assembled.

1. A set of eight macros is in the diskette or CD-ROM

   \BootMacs\ directory:

   - READCARD, PRINTOUT, PRINTLIN, DUMPOUT, $$GENIO, XDECI, XDECO, and XHEXO.

   These macros are "self-contained" and do not require linking with any run-time support routines. The program containing them must be run with AMODE(24) and RMODE(24). The first five are discussed at "Useful I/O Macros" on page 50.

   It is recommended that a PRINT NOGEN statement be included at the start of any program using them; otherwise, the generated code will show all the extra code needed to provide the requested functions. (Once you're comfortable with Assembler Language, remove the PRINT NOGEN statement if you want to take a look at the extra code.)

2. A set of 31 "X" macros is in the diskette \PSUMacs\ directory; they are listed in file PSUMACS.TXT in that directory. These macros require link-time availability of the run-time support routines in the PSUPROGS directory. The macros are discussed at "ASSIST Input/Output and Debugging Instructions/Macros" on page 42.

3. A set of 13 runtime support routines is in the \PSUPROGS\ directory; they are listed in file PSUMODS.TXT in that directory. These macros must be assembled separately and linked into a library that will be available at the time any program using the "X" macros is to be linked and executed. Assembling these programs requires that the macros from the \PSUMacs\ directory be previously installed in a library available to the assembler.

   Note: The names of these support routines contain the character '#', which may have special meanings to your file-transfer software. Be sure to test your transfer procedures with one of the files first. (For example, on CMS you may need to SET LINEDIT OFF.)

Origins

These macros originated in two student-oriented systems:

- The first five macros in the \BootMacs\ directory (with names not starting with the letter X) were part of the "SPASM" ("Single Pass Assembler") written at Stanford University. As with ASSIST/I, they were "built in" to the SPASM assembler-interpreter; these host macros were provided for applications that ran in a normal operating system environment.

  The "SPASM" macros were at one time distributed by the SHARE Program Library Agency as program number 360D-04.0.011. They are intended to run under MVS, CMS, and VSE.

  The simple XDECI, XDECO, XHEXO macros perform the same functions as the equivalent ASSIST/I pseudo-instructions, and are included here because they require no link-time library providing run-time support routines.

- The ASSIST "X" macros and runtime support modules were written at Penn State University as part of the ASSIST/360 system.

ASSIST Input/Output and Debugging Instructions/Macros

This is an excerpt from the *ASSIST INTRODUCTORY ASSEMBLER USER'S MANUAL* originally written by John R. Mashey of the Computer Science Department, Pennsylvania State University, in March 1974. The full and original form of this chapter can be found in the \BootAsst\ directory, as file ASUSERGD.HTML.

Input/Output Instructions - XREAD, XPRNT, XPNCH

Basic input/output facilities are provided by XREAD (card READer), XPRNT (line PRiNTer), and XPNCH (card PuNCH). They are written using the following format:

```
label     XMACRO area,length
```

where

label is an optional statement label

XMACRO is XREAD, XPRNT, XPNCH

area is the address in memory to be read or written. This area may be specified by an RX-type address, i.e., anything legal as the second operand of a LA instruction, such as:

```
0(1,2), AREA2+10, CARD+1(3), or =CL30'0 MESSAGE'
```

length specifies the number of bytes to be read or written.

This length can range from 1 to the maximum length for the appropriate device (80 for XREAD and XPNCH, 133 for XPRNT). The length field may be omitted, in which case the maximum length is used by default. It may also be specified as a register enclosed in parentheses, indicating that the length will be supplied at execution time from the designated register.

Condition Code

XPRNT and XPNCH do not change the condition code. XREAD sets the condition code to indicate normal processing or end-of-file as follows:

CC=0 a card was read, and length characters placed in user's area

CC=1 end-of-file encountered, no more cards can be read (/* found).

Carriage Control

XPRNT requires that the first character of the area be a valid carriage control character, such as blank (single space), '0' (double space, and '1' (new page), or any others which are available.

Examples of XREAD, XPRNT, XPNCH Usage

The following section of a program reads in a deck of cards until an end-of-file (/* card) is found, punches the last 70 characters of each card into the first 70 columns of each card punched, and prints some number of characters from each card, where the number +1 had been previously loaded into register 5 (the +1 is for the carriage control character). The cards are double-spaced on the printer.

```
READLOOP  XREAD CARD               read card, using omitted length
          BNZ   NOMORE             if CC=1, branch out.  BC 4,NOMORE
                                   or BM NOMORE would also work
          XPNCH CARD+10,70         punch 70 bytes, explicit length
          XPRNT CARD-1,(5)         print number of bytes, using
*                                  carriage control
          B     READLOOP           go back for next card to be read
NOMORE    EQU   *                  branch here when no more cards
.........more program statements.................
          DC    C'0'               carriage control for printing
*                                  card, right before CARD
CARD      DS    CL80               space for card to be read in
```

The following statements show how the programmer may easily produce messages and headings for his output, using XPRNT with literal character constants or related methods:

```
          XPRNT =CL30'1 A HEADING FOR NEW PAGE',30
          XPRNT =CL50' SECOND HEADING IMMEDIATELY UNDER FIRST',50
          XPRNT MSG,L'MSG          LET ASSEMBLER COMPUTE LENGTH
          XPRNT MSGX,MSGXL         ASSEMBLER COMPUTES LENGTH WITH EQU
MSG       DC   C'0 THIRD MESSAGE, SINGLE CONSTANT WITH LENGTH'
MSGX      DC   C' FOURTH MESSAGE, WHICH INCLUDES A SECTION FILLED IN'
          DC   C' DURING EXECUTION '
MSGNMBR   DS   CL12                SPACE FOR DECIMAL NUMBER-XDECO
          DC   C' END OF IT'
MSGXL     EQU  *-MSGX              MSGXL IS SET TO LENGTH OF MESSAGE
```

Debugging Instruction - XDUMP

One basic debugging command is provided, called XDUMP. It can be used in two different ways, to print either registers or storage areas:

General Purpose Register Dump

```
          XDUMP
```

Coding XDUMP with no operands prints the contents of the user's general purpose registers, in hexadecimal notation. The registers are preceded by a header line like the following:

```
BEGIN XSNAP - CALL    # AT CCAAAAAA USER REGISTERS
```

where

#	is the number of calls made to XDUMP so far, for identification.
CCAAAAAA	shows the last 32 bits of the user's PSW, in hexadecimal.
CC	gives the ILC, CC, and Program Mask at the time of the XDUMP.
AAAAAA	gives the address of the instruction following the XDUMP, and thus can be used to distinguish between the output of different XDUMP statements.

Note: XDUMP1 is the same as XDUMP with no operand.

Storage Dump

```
          XDUMP area,length
```

Coding XDUMP with an address and length produces a dump of a user storage area, beginning at the address given by area, and ending at the address area+length. The operands are specified like those of XREAD, XPRNT, XPNCH, except the length may not specify a register, but must be an explicit length.

The resulting output includes a header line like the above, followed by a hexadecimal and alphanumeric dump of the selected storage area. The storage is printed in lines showing two groups of four fullwords, preceded by the memory address of the first word in each line, and followed by the alphanumeric representation of the 32 bytes on the line, with letters, numbers, and

blanks printed directly, and all other characters translated to periods. The storage printed is also preceded by a line giving the address limits specified in the XDUMP.

If the length is omitted, the value 4 is used as a default.

Examples of XDUMP Usage

```
XDUMP AREA+10,80
XDUMP 8(1,4),100
XDUMP FULLWORD          use default value of 4
XDUMP TABL(3),12
```

Decimal Conversion Instructions - XDECI, XDECO

To facilitate numeric input/output, ASSIST accepts the commands XDECI (eXtended DECimal Input), and XDECO (eXtended DECimal Output). XDECI can be used to scan input cards for signed or unsigned decimal numbers and convert them to binary form in a general purpose register, also providing a scan pointer in register 1 to the end of the decimal number. XDECO converts the contents of a given register to an edited, printable, decimal character string.

Both instructions follow the RX instruction format, as shown:

```
XDEC#  REG,ADDRESS
```

where REG is any general purpose register, and ADDRESS is an RX-type address, such as

```
LABEL    0(R4,R5)     LABEL+3(2).
```

XDECI

XDECI is generally used to scan a data card read by XREAD. The sequence of actions performed by XDECI is as follows:

1. Beginning at the location given by ADDRESS, memory is scanned for the first character which is not a blank.
2. If the first character found is anything but a decimal digit or plus or minus sign, register 1 is set to the address of that character, and the condition code is set to 3 (overflow) to show that no decimal number could be converted. The contents of REG are not changed, and nothing more is done.
3. From one to nine decimal digits are scanned, and the number converted to binary and placed in REG, with the appropriate sign. The condition code is set to 0 (0), 1 (−), or 2 (+), depending on the value just placed in REG.
4. Register 1 is set to the address of the first non-digit after the string of decimal digits. Thus REG should not usually be 1. This permits the user to scan across a card image for any number of decimal values. The values should be separated by blanks, since otherwise the scanner could hang up on a string like -123*, unless the user checks for this himself. I.e. XDECI will skip leading blanks but will not itself skip over any other characters.
5. During step 3, if ten or more decimal digits are found, register 1 is set to the address of the first character found which is not a decimal digit, the condition code is set to 3, and REG is left unchanged. A plus or minus sign alone causes a similiar action, with R1 set to the address of the character following the sign character.

XDECO

XDECO converts the value from REG to printable decimal, with leading zeroes removed, and a minus sign prefixed if needed. The resulting character string is placed right-justified in a 12-byte field beginning at ADDRESS. It can then easily be printed using an XPRNT instruction. The XDECO instruction modifies NO registers.

Sample Usage of XDECI

The following program segment reads a card, and converts one decimal value of 1-9 digits punched anywhere on the card, placing this value in general register R0.

```
          XREAD  CARD         read card into a workarea
          XDECI  R0,CARD      scan and convert the number
```

XDECI can be used to convert an unknown number of decimal values from a card. This can be done by punching the values anywhere on the card, separated by one or more blanks. The last number on the card is then followed by a '$', which indicates the end of the data values to the program. The following program reads a card and converts numbers, storing their values in an array for later use, and stopping when the '$' is found.

```
          SR    2,2           zero for index to first word of NUMBERS
          XREAD CARD          read cardimage into input area
          LA    1,CARD        intialize R1 as scan pointer register
LOOP      XDECI 0,0(,1)       scan and convert next number
          BO    OVER          skip if bad number or $ (BC 1,OVER)
          ST    0,NUMBERS(2)  store legal value into array
          LA    2,4(2)        increment index value 1 fullword
          B     LOOP          go back for next number
OVER      CLI   0(1),C'$'     was this delimiter $
          BE    DONE          yes, so branch out
          XPRNT =CL30'0*** BAD INPUT ***STOP',30
DONE      ...... more instructions ........
NUMBERS   DS    20F           space for 20 values to be stored
CARD      DS    CL80          input workarea
```

Sample Usage of XDECO

The following converts the contents of register 4 to decimal and prints it. It assumes a reasonable value in R4, so that the first character of OUT is a blank for carriage control.

```
          XDECO 4,OUT         convert the number
          XPRNT OUT,12        print value
          ..... other assembler statments .....
OUT       DS   CL12           typical output area
```

Hexadecimal Conversion Instructions - XHEXI, XHEXO

Note: Some versions of ASSIST may not provide these instructions.

XHEXI and XHEXO provide easy conversion of hexadecimal numbers for input and output. The value of a hexadecimal number can be read from a card using XREAD, converted from character mode to a hexadecimal number, and the converted number is placed in the specified general purpose register with XHEXI. XHEXO provides an easy way to convert internal hexadecimal to an output form that can be printed using XPRNT.

XHEXI also places the address of the first non-hexadecimal number in register one, but if more than eight digits are scanned, the address of the ninth is placed in register 1.

XHEXI

```
XHEXI   REGISTER,ADDRESS
```

XHEXI, in the general form shown above where REGISTER is any general purpose register and ADDRESS is anything legal in an RX instruction, is used to do the following:

1. Beginning at the location ADDRESS, memory is scanned until the first non-blank character is found.
2. If the first character found is anything but a legal hexadecimal character (0-9,A-F), the condition code is set to overflow and this address is placed in register 1. If the REGISTER is anything but register 1, its contents remain unchanged.

3. One to eight hexadecimal characters are scanned, the number converted to hexadecimal, and the result is placed in REGISTER. The value placed in the register is internal hexadecimal with leading zeros included and the number is right justified.
4. Register one is set to the address of the first non-hexadecimal character. With this in mind, the user should not code register one as REGISTER. This allows you to scan across the card for any number of character strings. The strings should be separated by blanks. The end of the string could be flagged with any non-hexadecimal character and a test could be made after a Branch Overflow (see sample program).
5. If more than eight hex digits are found, register one is set to the address of the ninth. This allows the user to scan across long strings of numbers.

XHEXO

```
XHEXO   REGISTER,ADDRESS
```

XHEXO in the general form shown above converts the value in REGISTER and places it in a right-justified 8-byte field beginning at ADDRESS. It can be easily printed using an XPRNT instruction. The XHEXO instruction modifies NO registers.

Sample Program Using XHEXI and XHEXO

This program reads a data card with an unknown number of hexadecimal numbers on it. The end of the data is denoted by a '%' punched after the last number. The numbers are stored after being converted using XHEXI, and then converted for output using XHEXO.

```
         LA    3,STORAGE           WHERE NUMBERS STORED
         XREAD CARD,80             READ IN CARD
         XPRNT CARD,80             ECHO PRINT
         LA    1,CARD              ADDRESS OF CARD FOR SCANNING
LOOP     XHEXI 2,0(1)              CONVERT NUMBER PUT IN 2
         BO    ILLEGAL             CHECK FOR END
         XHEXO 2,AREA              PUT NUMBER IN OUTPUT AREA
         XPRNT REP,28              PRINT CARD AND MESSAGE
         ST    2,0(3)              STORE NUMBER
         LA    3,4(3)              INCREASE INDEX
         B     LOOP                GET NEXT NUMBER
ILLEGAL  CLI   0(1),C'%'           SEE IF END OF STRING
         BE    DONE                YES DONE
         XPRNT =CL50' ILLEGAL CHARACTER STOP',50
DONE     ....MORE INSTRUCTIONS.....
CARD     DC    81C' '              STORAGE FOR CARD
STORAGE  DS    20F                 STORAGE FOR NUMBERS
REP      DC    C' THE NUMBER IN R2 IS'
AREA     DC    CL8' '              STORAGE FOR OUTPUT NUMBER
```

Limit Dump Instruction - XLIMD

In order to conserve output records when necessary (for instance, when ASSIST is being used from a remote terminal of any sort), the XLIMD instruction is provided to enable the user to limit the size of his completion dump and choose the area to be printed. In general, it is used to eliminate the user's program code, leaving only his data areas in the completion dump.

The instruction is coded as follows:

```
XLIMD area,length
```

where

`area` is the beginning address where the completion dump should start. The area address is specified by an RX-type address, and must be within the user program area.

`length` is the length in bytes of the area the user wishes to be printed if a completion dump occurs.

Note that the XLIMD instruction format is exactly the same as that for the instructions XREAD, XPRNT, XPNCH. Thus the length may be given as a register number, enclosed in parentheses, or may be omitted, in which case a length of 1 is assumed. If the combined area address plus the length yields an address greater than the highest user address, or if the length is 1, the highest user address is used as an upper limit instead. Thus, storage will be printed to the end of the user program.

The suggested method of using XLIMD is to place all variables at the end of the program, then execute an XLIMD with an area address specifying the first variable desired, and omitting the length. This will cause the storage to be printed starting at the specfied address and going to the end of the program.

Sample Usage of XLIMD

The following program gives a typical way of using XLIMD.

```
DUMPTEST  CSECT
          USING *,15
          XLIMD VARIABL1           set dump limit right away
          ..........
          large number of machine instructions
          ..........
VARIABL1  DS    D                  first variable area
          ..........
          variable areas likely to be required for debugging
          ..........
          END
```

XLIMD may be executed any number of times during a program, but it is suggested that it be called early in any large program, if there is any possiblity that record limits could be exceeded.

Optional Input/Output Instructions - XGET AND XPUT

These instructions are similar to XREAD/XPRNT/XPNCH, but are more general, allowing the user to specify any filename to be read or written. WARNING: not all versions of ASSIST support these instructions. Also, a particular version may only support a specific set of file names, which can differ from installation to installation. It is advisable to check on local procedures. The instructions are coded as follows:

```
   label   xmacro    area,length
```

where

`label` is an optional statement label

`xmacro` is either XGET or XPUT

`area` is the address in memory to be read or written.

This area may be specified by an RX-type address, i.e., anything legal as the second operand of a LA instruction, such as:

```
 0(1,2)   AREA2+10   CARD+1(3)   or  =CL30'0***MESSAGE***'
```

`length` specifies the number of bytes to be read or written.

This length can range from 1 to the maximum length for the appropriate device (80 for cards, 133 for printer, etc.). The length field must not be omitted. it may also be specified as a register enclosed in parentheses, indicating that the length will be supplied at execution time from the designated register.

If during execution, the length has a value of zero, the file will be closed.

Note: During execution, register 1 must point to an eight byte character string which is the name of the file to be manipulated.

Condition Code

XGET and XPUT both change the condition code as follows:

CC=0	normal input/output occurred
CC=1	GET only: end of file occurred
CC=2	shows an error (like invalid data address) which causes the individual operation to be ignored.
CC=3	shows that the file could not be opened (because it is wrong direction, or DD card missing, or not enough room in tables, etc.).

Carriage Control

XPUT only requires the first character of the area to be a valid carriage control character, if the output device is the printer.

Closing of File

Performing an XGET or XPUT with a length of zero supplied in any GP register causes the designated file to be closed, so that it may then be reread; i.e.

```
          LA    1,=CL8'ddname'
          SR    0,0
          XGET  area,(0)
```

does a close.

Example of XGET and XPUT Usage

The following program will read and write a few files in parallel.

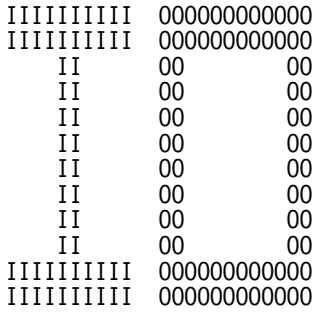
```
TEST1     CSECT
          BALR  12,0
          USING *,12
          SR    0,0
*
*    THIS PROGRAM WILL PROCESS A FEW FILES IN PARALLEL:
*
LOOP      LA    1,=CL8'CARD'       point to an input file
          XGET  AREA,80            do the input
          BNE   DONE               branch on endfile,
*                                  file automatically closed
          XREAD AREA2,80           do normal input
          LA    1,=CL8'PAPER'      point to a printer file
          XPUT  AREA-1,81          do output, note carriage control
          LA    1,=CL8'PAPER2'     point to other printer file
          XPUT  AREA2-1,81         do output on other file
          B     LOOP               try again
DONE      BR    14                 RETURN, IMPLICITLY CLOSE OTHER FILES
          DC    CL1' '
AREA      DS    CL80
          DC    CL1' '
AREA2     DS    CL80
          END
```

The extra JCL for the above is as follows:

```
//DATA.PAPER DD SYSOUT=A,DCB=(RECFM=FA,LRECL=133,BLKSIZE=133)
//DATA.PAPER2 DD SYSOUT=A,DCB=(RECFM=FA,LRECL=133,BLKSIZE=133)
//DATA.CARD DD *
THIS STUFF IS READ
  AT THE SAME TIME AS ANOTHER
 FILE IS READ
******  THE LAST CARD *******
//DATA.INPUT DD *
THIS IS THE NORMAL INPUT FILE
 AND IS READ AT THE SAME TIME AS ANOTHER FILE
  IS READ
  ********* THE LAST CARD *********
```

Note: A common usage for XGET might be to access files of test data.

Useful I/O Macros



To avoid getting tangled in the rules associated with any one Operating System, we will assume that the simple needs of your programs can be satisfied by providing the following facilities:

1. A means for reading 80-character card images into a named buffer area in the program, with provision for optionally transferring control to some out-of-line location if no further records are available from the input stream.
2. A means for printing line images on a printer file, with carriage control characters and an optional specification of the length of the character string to be printed.
3. A means for printing the value of the contents of an area of memory, giving the name as well as the value of the contents in an easily-readable format.
4. A means for printing the contents of the general registers and Floating-Point registers 0, 2, 4, and 6.
5. A means for returning control to the Supervisor when program execution has been successfully completed.
6. A means for giving a formatted hexadecimal dump of specified areas of memory.

Macro Facilities

These facilities are provided by the READCARD, PRINTLIN, DUMPOUT, and PRINTOUT macro-instructions. The properties assumed for each will be described below; first, we will give a summary of the abbreviations to be used in the descriptions.

<name>	any valid symbol naming an area of memory which is addressable from the point where it is used in a macro-instruction.
<number>	any valid self-defining term; the limits on the size of the term will be described for each macro-instruction.
<nfs>	a valid (and optional) name-field symbol that names the macro-instruction in whose name field it appears.
[optional]	the use of square brackets around a term means that its use is optional.
...	an ellipsis means that the preceding item may be repeated any number of times.

On MVS and CMS, input is from DDname SYSIN and printed output goes to DDname SYSPRINT. On VSE, input is from SYSIPT and printed output goes to SYSLST.

The READCARD Macro-Instruction

READCARD reads card images into an 80-byte buffer in the program. This macro-instruction is written

```
<nfs>    READCARD  <name>[,<name>]
```

and will cause a card to be read from the input file into the 80-byte area beginning at the first operand address. If no card is available, then (1) control is returned to the instruction specified by the second operand if present, or (2) the run is terminated if no second operand is present. For example,

```
         READCARD  INCARD
```

will read the next input card and store it as an 80-byte EBCDIC character string at the location named INCARD. If no card is present, the job will be terminated. The instruction

```
GETCARD  READCARD  INCARD,ENDFILE
```

will do the same as the previous example, except that if no card is available control will be transferred to the instruction named ENDFILE.

The PRINTLIN Macro-Instruction

The PRINTLIN macro-instruction is written in the form

```
<nfs>    PRINTLIN  <name>[,<number>]
```

and causes the character string beginning at the location defined by the first operand to be printed. The *first* character of the string will be *detached* and used for spacing control. The (ASA Standard) carriage control characters used are:

- EBCDIC blank means single space
- EBCDIC zero means double space
- EBCDIC minus means triple space
- EBCDIC one means start at the top of a new page
- EBCDIC plus means *no* spacing (the new line will be printed over the previous one).

If the second operand (<number>) is omitted, the length of the character string is assumed to be 121 bytes, which means that 120 characters will be printed after the first is detached. If the second operand is present, it is taken to be the length of the string; the number of characters specified will be placed at the left end of an internal buffer, extended to 121 bytes with blanks if necessary, and then sent to the printer file. If the second operand exceeds 121, only the first 121 characters are sent to the printer file. For example, we could write

```
PRTTL    PRINTLIN  TITLE
      - - -
TITLE    DC   CL121'1TITLE FOR TOP LINE OF THE PAGE'
```

to print the indicated title at the top of a new page. If we wrote simply

```
         PRINTLIN  TITLE,1
```

then we would skip to the top of a new page and print a blank line there, because only the spacing control character (the "1") is to be transmitted from the program to the print file.

The PRINTOUT Macro-Instruction

The PRINTOUT macro-instruction provides the ability to print the value of the contents of named areas of memory, the contents of registers, and to terminate execution.

The operand field of the PRINTOUT macro-instruction may contain any number of <name>s or <number>s separated by commas, with no intervening blanks. An operand consisting of a single asterisk will terminate execution. The simple forms of the PRINTOUT macro-instruction are therefore written

```
<nfs>    PRINTOUT  <name>[,<name>...]

<nfs>    PRINTOUT  <number>[,<number>...]

<nfs>    PRINTOUT  *
```

where any combination of the above operands may be used in an operand list; it will be assumed that if the asterisk is used, it is the last operand in the list. For example,

```
ALLDONE  PRINTOUT  *
```

would terminate execution.

A <number> operand which has a value between 0 and 15 will cause the contents of the corresponding general register to be printed; if the value of the <number> is between 16 and 19, then the contents of F0, F2, F4, or F6 respectively will be printed; if the <number> is 20 or greater, it will be treated as a storage address. For example, to print the contents of the floating-point registers, we could write

```
ALLFPRS  PRINTOUT  16,17,18,19
```

To print the contents of R14 and then terminate execution, we could write

```
         PRINTOUT  X'E',*
```

To print the contents of memory areas named A, B, and C, we could write

```
         PRINTOUT  A,B,C
```

The DUMPOUT Macro-Instruction

The DUMPOUT macro-instruction is written in the form

```
<nfs>    DUMPOUT   <name>[,<name>]
```

It prints a formatted hexadecimal dump of the area of memory starting with the fullword containing the first operand, 32 bytes to a line. If the second operand is omitted, one line will be printed. If the second operand is given, all of memory between the two addresses will be dumped. For example,

```
ADUMP    DUMPOUT   A
```

will cause the 32-byte area of memory starting at (or very near) A to be dumped. Similarly,

```
ABDMP    DUMPOUT   A,B
```

would print a dump of the area of memory starting with a line containing the byte at A and ending with a line which includes the byte named B.

PRINTOUT and DUMPOUT Header

Normally, the output produced by the PRINTOUT and DUMPOUT macros will be preceded by a "header" line:

```
   *** PRINTOUT REQUESTED AT LOCATION xxxxxx, CC=n
or
   ***  DUMPOUT REQUESTED AT LOCATION xxxxxx, CC=n
```

where CC=n shows the current Condition Code setting.

To suppress this header line, you can specify an additional operand HEADER=NO on the PRINTOUT or DUMPOUT macro. For example:

```
         PRINTOUT  A,B,C,HEADER=NO
ABDMP    DUMPOUT   A,B,HEADER=NO
```

The default is HEADER=YES.

Memory References

These macros do not check for valid storage references. Addresses of operands in unavailable storage may cause program interruptions or abnormal terminations.

The Macro Instruction Definitions

The macro definitions implement the READCARD, PRINTLIN, PRINTOUT, and DUMPOUT macro-instructions. An important feature of the macro-instructions is that they may be used *anywhere* in a program: they make no changes to any register, do not affect the Condition Code, and do not contain any USING or DROP instructions.

The fifth macro definition (for $$GENIO) generates the code that performs the functions requested by the other four macros. The code does *not* require the user to do anything special about addressability, so the macros may be used in any program with no fears that registers or USING specifications will be changed in any way.

The macro definitions should be placed into a macro library accessible to the Assembler.

The macros have been extensively tested under both MVS, CMS, and VSE. As written, they are set up to run under MVS/CMS as the default. To run under VSE, change the &$$DOS SETB statement (near the front of the $$GENIO macro definition) according to the instructions there. Similarly, to change the default file names or print line length, modify the following SETC statements as indicated there.

All symbols generated in the expansions of these macros begin with the two characters $$. If this conflicts with your conventions (or desires), run the deck through a program which changes each occurrence of '$$' to whatever two other characters you like. You may prefer to precede your assemblies with a PRINT NOGEN statement, to suppress the (lengthy) listing of the macro expansions.

Assembler Boot Camp: PC and Lab Usage Notes

1. To use ASSIST/I from a CD-ROM, install it on a hard drive. (You may wish to copy the entire CD contents to a **ABC** directory on your hard drive.)
   - Click **Start**, then **Programs**, then **Accessories**, then **Command Prompt**. (It may already be an icon on your desktop.)
   - If the prompt shows anything other than **C>ABC**, enter whatever **cd ..** and **cd ABC** commands are needed to get to the **ABC** directory, and then type **cd BOOTASST**. Then, enter **CAS**, and you're ready to go.
2. The ASSIST/I software is also installed on the workstations in the SHARE "Lab".
   - From the Windows desktop, click on "'My Computer'".
   - Click the **BootAsst** folder; then click **CAS.EXE**, and you're ready to go.

Running a Program

To run a program: enter **R** (**Don't press Enter!**), then the name of the program (something like, **XXX.ASM**). For example, to run the first demo program, enter **DEMOA.ASM**; then press Enter.

Program Entry and Editing

- Click **Start**, then **Programs**, then **Accessories**, then **Notepad**, and enter your program. Click **File**, then **Save As**, and enter the program name (say, **XXX.ASM**). Then, in the **BootAsst** folder, right-click the file name, then **Rename**, and remove the **.txt** extension, if any.[1]
  - Or, in CAS, enter **E**, then the name of the file (as described in "Editing Programs" on page 17). Be careful: the Assist/I editor is quite difficult to use, and it's easy to get lost.
- The first line of your program should be a comment (* in column 1) with your name. (This is so your printed output — in case we have access to a printer! — can find its way back to you. The printer may be in a different or restricted-access area, so someone may have to pick up the outputs for you.)
- The rest of the program, except for comments, should be in upper case letters.
- Put a $ENTRY line after the END statement.

Program Printing

To print a program: enter P, then the name of the program output (say, **XXX.PRT**). Or, in the **BootAsst** folder, right-click the file and then click **Print**.

Alternatively, at the command prompt, enter **notepad XXX.PRT**, and then click the **File** drop-down list, and then click **Print**.

[1] Under Windows (TM) you can change the Folder Options to no longer "Hide extensions for known file types", so that Windows won't automatically append a file type without your knowing it.