Chapter 3: A First Program

In which we look at a complete program, discuss each component, and generally try to get the hang of what an assembler program contains.

First Principles

When writing assembly language, you have complete control over how you arrange the parts of a program. You get to decide the placement of data fields, file descriptors, input and output buffers, entry points, and instructions in memory. In a real sense, you are working with memory directly, arranging the bits and bytes exactly as you see fit. You're not constrained by a high-level programming language or programming paradigm. The language software that typically sits between you and the machine has been removed: It's you and the machine. For many programmers, that's exhilarating. Within the limits of what the operating system will let you do, you have complete control.

With power comes responsibility. Moving away from your trusty compiler and closer to the machine means you have to pay attention to exactly what your specific machine can do. You have to learn the instruction set for your particular box. In our case, that's the family of System/z machines. IBM assembly language. Every assembly language is designed for a particular machine architecture and can't be reused when you change to a different architecture. Even so, learning the principles of one assembly language will help leverage your way into others. It will also make you a better application programmer when you move back to the comfortable world of compilers and interpreters.

There's another constraint we have to deal with: Machine instructions are fine-grained. By that, I mean they're designed to do one small task, like taking a number in memory and putting it into a register. While there are some exotic assembler instructions that are quite powerful, it usually takes multiple (sometimes hundreds) of assembler instructions to do something equivalent to a single instruction in a high-level language.

An assembler programmer is like a neurosurgeon using a scalpel to make a series of small but crucial incisions to achieve the desired result. It's also the case that a skilled assembler programmer can code precisely what is required and no more. Think of a Haiku master who pens a short poem that is more meaningful to you than a ponderous ode, which runs on and on for thousands of lines. (Interestingly, the word Haiku is related to the word "cut.")

As a result of the fine granularity, there are lots of assembly instructions. The System 360 began with more than two hundred instructions, and today's System/z has over two thousand. The sheer number of instructions is daunting. Happily, we don't have to learn them all today. Many instructions have specialized uses. Many others are privileged instructions that are not available for general use.

This book will cover less than a hundred workhorse instructions that every assembly language programmer needs to master. The architecture has been improved greatly over the years so this book will also cover certain newer instructions and techniques. You will also gain the basic knowledge and programming principles required to go further when it's appropriate. Once you catch on to a few basic patterns, whenever you learn one assembler instruction, you'll find you may know four or five others by association.

Our First Program: A complete assembler program

```
         PRINT  ON,NODATA,NOGEN
         TITLE  'SKELETON ASSEMBLER PROGRAM'
ASMSKEL  CSECT
**********************************************************************
* STANDARD ENTRY LINKAGE
**********************************************************************
         STM   R14,R12,12(R13)        SAVE CALLER'S REGS
         BASR  R12,R0                 ESTABLISH
         USING *,R12                  ADDRESSABILITY
         LA    R2,SAVEAREA            POINT TO MY LOWER-LEVEL SA
         ST    R2,8(,R13)             FORWARD-CHAIN MINE FROM CALLER'S
         ST    R13,SAVEAREA+4         BACK-CHAIN CALLER'S FROM MINE
         LR    R13,R2                 SET 13 FOR MY SUBROUTINE CALLS
**********************  BEGIN LOGIC  *********************************
         OPEN  (FILEIN,(INPUT))       OPEN THE INPUT FILE
         OPEN  (FILEOUT,(OUTPUT))     OPEN THE OUTPUT FILE
         GET   FILEIN,RECIN           READ A RECORD
LOOP     EQU   *
         MVC   RECOUT,RECIN           COPY DATA TO OUTPUT RECORD
         PUT   FILEOUT,RECOUT         WRITE THE RECORD TO THE FILE
         GET   FILEIN,RECIN           READ THE NEXT RECORD
         J     LOOP                   JUMP BACK AND DO THIS ALL AGAIN
EXIT     EQU   *
         CLOSE FILEIN                 CLOSE THE INPUT FILE
         CLOSE FILEOUT                CLOSE THE OUTPUT FILE
*********************** STARDARD EXIT ********************************
         L     R13,SAVEAREA+4         POINT TO CALLER'S SAVE AREA
         LM    R14,R12,12(R13)        RESTORE CALLER'S REGS
         LA    R15,0                  SET RETURN CODE REG 15 = 0
         BR    R14                    RETURN TO CALLER
**********************  DATA AREAS   *********************************
FILEIN   DCB   DSORG=PS,                                               X
               MACRF=(GM),                                             X
               DEVD=DA,                                                X
               DDNAME=FILEIN,                                          X
               EODAD=EXIT,                                             X
               RECFM=FB,                                               X
               LRECL=80
FILEOUT  DCB   DSORG=PS,                                               X
               MACRF=(PM),                                             X
               DEVD=DA,                                                X
               DDNAME=FILEOUT,                                         X
               RECFM=FB,                                               X
               LRECL=80
RECIN    DS    CL80       INPUT AREA FOR RECORDS
RECOUT   DS    CL80       OUTPUT AREA FOR RECORDS
SAVEAREA DC    18F'0'     AREA FOR MY CALLEE TO SAVE & RESTORE MY REGS
         LTORG
         YREGS
         END   ASMSKEL
```

If we are developing programs on a System/z mainframe, the assembler program above would be saved as a member in a library (a PDS or PDSE). In order to assemble, link, and run the program, you would need some job control language (JCL) which directs the operating system how to process your code. That JCL might be contained in a separate member in a library, or it might surround the code above. Check with your local system personnel for details on how to submit and review jobs on your mainframe.

Let's try to get a sense of the program a few lines at a time. Here are the first three lines,

```
         PRINT  ON,NODATA,NOGEN
         TITLE  'SKELETON ASSEMBLER PROGRAM'
ASMSKEL  CSECT
```

These three statements are assembler instructions. They are also known as *assembler directives* – messages to the assembler that control how our program will be assembled. The `PRINT` directive indicates,

1) The source code that follows the directive should be added to the listing (`ON`).
2) We don't need to see all the generated object code for long literals (`NODATA`).
3) We don't need to see the generated object code for assembler macro instructions (`NOGEN`).

We will learn more about literals and macros later. Alternatives for `PRINT` are `OFF`, `DATA`, and `GEN`. Multiple `PRINT` directives can be coded to control exactly how the program listing is formatted. You can assume that `PRINT ON,NODATA,NOGEN` is a reasonable way to tell the assembler how to format our program listing.

The `TITLE` directive tells the assembler to put a title on the top of each page in the printed listing of our program.

The `CSECT` directive tells the assembler to organize the code that follows it into a control section called `ASMSKEL`. Assembler programs consist of one or more CSECTs. In this book, we will concentrate on writing single-CSECT programs.

Here are the next ten Lines:

```
*********************************************************************
* STANDARD ENTRY LINKAGE
*********************************************************************
         STM   R14,R12,12(R13)        SAVE CALLER'S REGS
         BASR  R12,R0                 ESTABLISH
         USING *,R12                  ADDRESSABILITY
         LA    R2,SAVEAREA            POINT TO MY LOWER-LEVEL SA
         ST    R2,8(,R13)             FORWARD-CHAIN MINE FROM CALLER'S
         ST    R13,SAVEAREA+4         BACK-CHAIN CALLER'S FROM MINE
         LR    R13,R2                 SET 13 FOR MY SUBROUTINE CALLS
```

Lines that begin with an asterisk in column 1 are always treated as comments, so, this section begins with three comments. These lines are printed in a program listing, but don't become part of the executable code.

The next seven lines of the program are amazingly rich in terms of programming conventions and functionality. In order for assembler programs to work with other assembler programs (or other programs written in high-level languages) IBM established some programming rules that govern register usage. These rules are referred to as *linkage conventions*.

Suppose Program A is calling Program B. These two programs would occupy different areas in memory as they execute. But our machine has only one set of sixteen general-purpose registers that both programs must share. We need to establish some rules regarding register usage, so that one program doesn't inadvertently destroy a register value established by the other program. By following IBM's linkage conventions, we can write programs that work together seamlessly.

The linkage convention code is a bit daunting for a beginner, so for the moment, we will accept on faith that this linkage code (or something like it) is required for each program. In fact, we will use the same code for all the programs we write as a way of getting into a program. This is the *standard entry code* for us. Eventually, we will return to look at the code in some detail.

For the technically curious: Besides following IBM linkage conventions, the above code sets up addressability for symbols – this frees us from having to code all the variables with explicit addresses.

For the especially curious: The IBM Language Environment (LE) is a runtime system that provides some common functions across languages and makes it easier for programs written in different languages to work together. Programs written in high-level languages will always run in the LE. IBM Assembler is the one exception: Assembly language programs can run in the LE or not. The above program would not be suitably written to run in the LE because LE reserves register 12 for a special use. We will assume that writing LE-compliant assembly programs is an advanced topic, and that the programs we write will not run in the LE.

Once we get past the linkage convention code, we are free to write our program logic,

```
*********************  BEGIN LOGIC  *********************************
         OPEN  (FILEIN,(INPUT))       OPEN THE INPUT FILE
         OPEN  (FILEOUT,(OUTPUT))     OPEN THE OUTPUT FILE
         GET   FILEIN,RECIN           READ A RECORD
LOOP     EQU   *
         MVC   RECOUT,RECIN           COPY DATA TO OUTPUT RECORD
         PUT   FILEOUT,RECOUT         WRITE THE RECORD TO THE FILE
         GET   FILEIN,RECIN           READ THE NEXT RECORD
         J     LOOP                   JUMP BACK AND DO THIS ALL AGAIN
EXIT     EQU   *
         CLOSE FILEIN                 CLOSE THE INPUT FILE
         CLOSE FILEOUT                CLOSE THE OUTPUT FILE
```

Input and Output operations in System/z are usually record-oriented. Our programs read and write records as the basic I/O unit. Records are often combined to make larger entities called **blocks** for efficient transfer between disk and memory. Files have to be opened prior to use and should be closed when we are done processing them. The above code opens `FILEIN1` for input operations and `FILEOUT` for output processing.

The `GET` macro instruction reads a record from `FILEIN1` and delivers it to a record structure in our program called `RECIN`. You can think of this first `GET` as a "priming read" for the loop that appears underneath. The definitions of the files and the record areas occur later in the code.

The next five lines make up the heart of the program. We use another assembler directive called `EQU` (equate) to establish `LOOP` as a symbol representing a location that can be the target of a branch or jump operation. The * refers to the current value of the location counter. When the assembler processes the EQU, the * refers to the address of the next instruction – the MVC instruction. When we jump (J) back to LOOP later, the next instruction that will be executed is the MVC. EQU is not an executable instruction. It's a directive to the assembler to associate a number (in this case, the address of the next instruction) with a symbol (LOOP).

MVC is an executable instruction that copies the contents of RECIN (a record we just read) to RECOUT, which is part of an output record structure. PUT writes the contents of RECOUT to the output file. After that, GET reads another record, and then we unconditionally jump back to the top of the loop with J.

There are two executable instructions in the five lines of code: MVC and J. PUT and GET are examples of executable macro instructions. The assembler expands each of these macro instructions into a sequence of executable instructions that read or write records. Commonly needed tasks are often coded as macros. Macros can be supplied by the system – like PUT and GET, or defined by a programmer. The instructions these macros represent are currently hidden from us because we specified PRINT NOGEN at the beginning of the program.

You might think that the code above contains an infinite loop, but it doesn't. There is a feature of the GET macro that isn't obvious at first glance. When GET is executed against an empty file, it causes an unconditional branch to a specified address (an EODAD value - End of Data Address) in the program. The target of that branch is in the EXIT equate in the next block of code. That EODAD address is identified later when the file is defined in a DCB.

We have written a processing loop to process a file of records. Before falling into the loop, the first GET operation reads the first record in the file. Upon entering the loop, we "process" a record by moving it from an input buffer to an output buffer. After printing the new record, we read the next record on the file with a "continuation read". This pattern of reading a sequential file with a priming read before the loop body and a continuation read at the bottom of the loop body, is a good one. If the file contains records, the priming read will read the first record and the continuation read will read all the others. If the file is empty, we discover that before falling into the loop, so we don't accidentally process a record that doesn't exist.

After branching to the EXIT label, the program closes both files, releasing system resources. The rest of the instructions are our standard exit code that satisfy IBM's linkage conventions. This is important code that will require a good bit of explanation later. We postpone that task for the moment. Let's take it on faith that the last four lines provide a clean exit from our program and carry us back to the operating system.

The last line above is an executable instruction that branches unconditionally to the address currently stored in register 14. If everything goes well, register 14 will contain the return address to the program that called ours. In this case, that's the operating system. One consequence of coding an unconditional branch is that execution will never fall into the area under this branch (unless register 14 is pointing there).

Our file definitions follow the exit code. In a real sense, files represent our most precious resources, so placing their definition just under the branch instruction is a conscious design decision to protect the DCB macro information. The branch instruction that lives above the DCB carries us out of the program in every case, so it is impossible to fall into the storage area represented by the DCB macro.

The DCB macro doesn't generate executable code but simply assembles information about the file that is needed by the operating system. Since it is positioned under the branch, it is unlikely that a rogue instruction will corrupt this storage area and destroy access to the files we are processing. There is a method in this madness.

```
FILEIN   DCB   DSORG=PS,                                      X
               MACRF=(GM),                                    X
               DEVD=DA,                                       X
               DDNAME=FILEIN,                                 X
               EODAD=EXIT,                                    X
               RECFM=FB,                                      X
               LRECL=80
```

Next, let’s tackle the DCB (Data Control Block) macro parameters in some detail.

DSORG=PS – Data Set Organization – PS indicates that the files' records are “physical sequential,” meaning the records occur and are processed in sequence, beginning with the first record and moving forward in the file to the last record.

MACRF=(GM) – the G indicates we will be issuing GET macros on the file. In other words, this is an input file. The M indicates that records are moved from system buffers to a record area in our program. (Move mode I/O.)

DEVD=DA – The records are being written to a direct access device (a disk, not a tape)

DDNAME=FILEIN – DD stands for Data Definition and represents the file's name inside the JCL that runs the program. We don’t want to tie the program down to a specific physical file, so we use a symbolic filename like FILEIN to refer to the file inside a program. The JCL associates the internal filename with a real physical file on the disk.

EODAD=EXIT – this parameter (End of Data Address) denotes the target label we branch to when we issue a GET on an empty file. There’s nothing magic about the word EXIT – we could have used HERE, THERE, or YON. It’s just a symbol that denotes a target address.

RECFM=FB – The file will contain fixed-size records that are blocked together into larger units for efficient I/O.

LRECL=80 – The logical record size is 80 bytes.

The DC X’FFFFFFFF’ that precedes the DCB is an “eye-catcher” in case we want to find the DCB easily in a storage dump. The X at the end of some of the lines is a continuation. The DCB is one long statement. Any non-blank character in column 72 acts as a continuation character for the line. Commas must separate the parameters.

The DCB for the output file follows next.

There are two differences in the output DCB that we note:

1) MACRF=(PM) – the P indicates we will be issuing PUT macros on the file. In other words, this is an output file. The M indicates that records are moved from system buffers to a record area in our program. (Move mode I/O.)

2) Since this is an output file, the EODAD parameter is omitted.

After the DCBs, we coded three consecutive definitions for fields needed in our program.

```
RECIN    DS    CL80       INPUT AREA FOR RECORDS
RECOUT   DS    CL80       OUTPUT AREA FOR RECORDS
SAVEAREA DC    18F'0'     AREA FOR MY CALLEE TO SAVE & RESTORE MY REGS
```

- RECIN – An 80-byte storage area in which we expect to find character data.
- RECOUT – An 80-byte storage area in which we expect to find character data
- SAVEAREA – a 72-byte storage area used for saving registers. We define this with eighteen repetitions of a fullword field (4 bytes). This allocates 72 consecutive bytes.

Following IBM's linkage conventions, every program needs a save area. We will revisit this topic when we discuss the standard entry and exit code.

The program ends with the following lines,

```
         LTORG
         YREGS
         END   ASMSKEL
```

The LTORG assembler directive tells the assembler to generate any literals (like integers or spaces) needed by the program at this location. We will see examples of literals in future programs.

The YREGS macro generates a sequence of sixteen equates,

```
      R0      EQU    0
      R1      EQU    1
      R2      EQU    2
              …
      R15     EQU    15
```

This macro provides symbolic names for each of the registers. Instead of using this macro, you could code the equates by hand. Some programmers prefer to not to use symbolic names for registers. Instead they simply code integers, for example,

```
      STM  14,12,12(13)
```

For a introductory book, I think the symbolic names will help you distinguish when a number like 12 is used as a register and when it is used as a displacement, as in the previous statement. While we are on the subject of style, the assembler is case insensitive, so a capital *A* is equivalent to a lowercase *a*. If you write a program in all lowercase and omit the register equates, the result is a very modern looking and clean program. Being an old dog, I find it easier to read assembler programs in uppercase, so I'm sticking to an older-looking style.

Finally, the physical end of the program is denoted by the END directive. The label that follows it, ASMSKEL, represents the entry point where execution should begin. In assembler, you have complete control, and execution doesn't have to start at the program's first executable instruction.

At this point, we've tried to wrap our minds around an entire program. You may be missing many of the details, but don't dismay. This chapter represents a flyover from 20,000 feet. Over time, the details will be filled in. At this point, just try to get a sense of the landscape

A Programming Exercise

Try typing in this first program and getting it to run on your system. Start from scratch and think about the commands you are typing. After you get it running, we will use it as a skeleton or template for building other programs.

There are some formatting details you will need to be learn to get this done. Traditionally, assembler programs have conformed to the following rules for coding statements

1) Most assembler statements are written in a single 80-byte line. Columns are numbered 1-80, left to right.

2) Longer assembler statements can be continued by coding any non-blank character in column 72.

3) Continued statements begin in column 16 on the next line.

4) An assembler statement can have up to four parts from left to right:
   a) A Name field,
   b) An Operation field,
   c) An operands field that has a sequence of operands separated by commas,
   d) A comment field.
   Each of the four parts of an assembler statement are separated by one or more spaces.

5) If the name field isn't empty, the symbol there conforms to the following rules:
   a) The symbol must consist of 63 or fewer alphanumeric characters – 8 or 10 characters is not uncommon,
   b) The first character must be alphabetic or national ($, #, @)
   c) Additional characters are alphabetic or the digits 0-9,
   c) The first character must start in column 1
   c) No spaces or double-byte data can appear in the symbol.

6) The operation field is the only required field and denotes an op-code for an instruction (like `MVC`), a declarative (like `DS`), a directive (like `PRINT`), or a macro (like `PUT`).

7) The operands field contains one or more operands separated by commas. No space (except within quotes) can occur here. An encountered space denotes the end of the operands field.

8) Anything following the operands is treated as a comment.

If you code the following line starting in column 1,

```
HERE     AP   TOTAL,ICOST     KEEP RUNNING TOTAL OF ICOST
```

The name field is `HERE`, the operation or op-code is `AP`, the operands are `TOTAL` and `ICOST` and "`KEEP RUNNING TOTAL OF ICOST`" is a comment. It is common to comment most of the lines in an assembler program. Since you can assume that anyone looking over your code knows *how* the assembly language works, comments should describe *why* you are doing something, not *what* you are doing. For example, "`ADD ICOST TO TOTAL`" is a poor comment for the statement above.

If you code the following line starting in column 10,

```
         AP   TOTAL, ICOST      KEEP RUNNING TOTAL OF ICOST
```

The name field was `omitted`, `AP` is the operation, the operands field consists of `TOTAL,` while "`ICOST      KEEP RUNNING TOTAL OF ICOST`" is treated as a comment. Since the assembler expects two operands for an `AP` operation, the space after the comma will cause assembly errors.

Getting Your Program to Run on Z

After you have typed in the program and saved it, it's time to see if it works. This is a three-step process that is controlled by the JCL you are using.

Here's some of the JCL I am using:

```
//KC02486A JOB (KC024861),'YOUR NAME',REGION=3M,CLASS=A,MSGCLASS=H,
// NOTIFY=KC02486,MSGLEVEL=(1,1),TIME=(0,1)
//STEP1    EXEC PROC=HLASMCLG
```

This isn't a book on JCL, so I won't explain this line-by-line or provide you a complete listing of the JCL. The important part is that we are invoking a JCL procedure called `HLASMCLG`. The `CLG` stands for *compile, link, and go*. It might more accurately be called *assemble, link, and go*.

In the first step, the assembler reads your program and produces a listing, and if the program is correct, an *object module* that is a machine language version of your assembler program. The object module is then passed to the binder (linkage editor is an older term for something similar). The binder can combine your object module with other object modules and load modules to make a complete program called a *load module*. A load module is always ready to run, so in the third step, your load module is loaded into memory and control is turned over to it to execute. The output of these steps can be examined by looking at the results in a spool manager like SDSF.

Reading an Assembler Listing

One of the interesting things about programming in assembly language is that the assembler produces a line-by-line translation of the assembler statements you write into machine code. Below is the assembly listing of our first program.

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                  HLASM R6.0  2021/08/25 12.48
                                      1          PRINT  ON,NODATA,NOGEN                                          00010006
         SKELETON ASSEMBLER PROGRAM                                                                            Page    4
  Active Usings: None
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                  HLASM R6.0  2021/08/25 12.48
000000                00000 00230     3 ASMSKEL  CSECT                                                           00020006
                                      4 **********************************************************************  00031006
                                      5 * STANDARD ENTRY LINKAGE                                                 00032006
                                      6 **********************************************************************  00033006
000000 90EC D00C            0000C     7          STM   R14,R12,12(R13)        SAVE CALLER'S REGS                00120006
000004 0DC0                           8          BASR  R12,0                  ESTABLISH...                      00130007
                 R:C  00006           9          USING *,R12                  ADDRESSABILITY                    00140007
000006 4120 C1E2            001E8    10          LA    R2,SAVEAREA            POINT AT MY SAVE AREA             00170015
00000A 5020 D008            00008    11          ST    R2,8(,R13)             FORWARD CHAIN MINE FROM CALLER    00180007
00000E 50D0 C1E6            001EC    12          ST    R13,SAVEAREA+4         BACK-CHAIN CALLER'S FROM MINE     00190007
000012 18D2                          13          LR    R13,R2                 SET 13 FOR MY SUBROUTINE CALLS    00191007
                                     14 **********************  BEGIN LOGIC  *********************************  00192006
000014 4D10 C016            0001C    15          OPEN  (FILEIN,(INPUT))       OPEN THE INPUT FILE               00192107
00001E 0700                          22          OPEN  (FILEOUT,(OUTPUT))     OPEN THE OUTPUT FILE              00192207
00002A 4110 C082            00088    29          GET   FILEIN,RECIN           READ A RECORD                     00192307
                      0003A          38 LOOP     EQU   *                                                        00193007
00003A D24F C192 C142 00198 00148    39          MVC   RECOUT,RECIN           COPY DATA TO OUTPUT RECORD        00193107
000040 4110 C0E2            000E8    40          PUT   FILEOUT,RECOUT         WRITE THE RECORD TO THE FILE      00193207
000050 4110 C082            00088    49          GET   FILEIN,RECIN           READ THE NEXT RECORD              00193307
000060 A7F4 FFED            0003A    58          J     LOOP                   JUMP BACK AND DO THIS ALL AGAIN   00193407
                      00064          59 EXIT     EQU   *                                                        00193507
000064 4D10 C066            0006C    60          CLOSE FILEIN                 CLOSE THE INPUT FILE              00193607
00006E 0700                          67          CLOSE FILEOUT                CLOSE THE OUTPUT FILE             00193707
                                     74 *********************** STARDARD EXIT *********************************  00193807
00007A 58D0 C1E6            001EC    75          L     R13,SAVEAREA+4         POINT TO CALLER'S SAVE AREA       00194007
00007E 98EC D00C            0000C    76          LM    R14,R12,12(R13)        RESTORE CALLER'S REGS             00194107
000082 41F0 0000            00000    77          LA    R15,0                  SET RETURN CODE REG 15 = 0        00194207
000086 07FE                          78          BR    R14                    RETURN TO CALLER                  00194307
                                     79 **********************  DATA AREAS   *********************************  00199002
                                     80 FILEIN   DCB   DSORG=PS,                                               X00199107
                                                       MACRF=(GM),                                             X00199207
                                                       DEVD=DA,                                                X00199307
                                                       DDNAME=FILEIN,                                          X00199407
                                                       RECFM=FB,                                               X00199507
                                                       EODAD=EXIT,                                             X00199616
000088 0000000000000000                                LRECL=80                                                 00199716
                                    122 FILEOUT  DCB   DSORG=PS,                                               X00200007
                                                       MACRF=(PM),                                             X00210002
                                                       DEVD=DA,                                                X00220002
                                                       DDNAME=FILEOUT,                                         X00230007
                                                       RECFM=FB,                                               X00240002
0000E8 0000000000000000                                LRECL=80                                                 00250002
000148                              164 RECIN    DS    CL80       INPUT AREA FOR RECORDS                         00251007
000198                              165 RECOUT   DS    CL80       OUTPUT AREA FOR RECORDS                        00251107
0001E8 0000000000000000             166 SAVEAREA DC    18F'0'     AREA FOR MY CALLEE TO SAVE & RESTORE MY REGS  00252002
000230                              167          LTORG                                                          00253002
                                    168          YREGS                                                          00254002
000000                              187          END   ASMSKEL                                                  00255017
```

Each line consists of a number of parts. There is a heading that looks like this:

```
 Loc  Object Code    Addr1 Addr2  Stmt   Source Statement
```

Let’s examine each part.

LOC – By default, the assembler consecutively numbers each byte it generates starting at 0 (usually). Each number corresponds to a **location** within the program. Each location counter value is expressed using a six-digit hexadecimal value, so the location of the first byte of the program is 000000. The first instruction above (`STM`) is four bytes long, so the second instruction (`BASR`) starts at 000004. Notice that some statements don’t have an associated location. That’s because some assembler statements don’t generate machine code. For example, `USING` is a directive that tells the assembler how to process the code that follows it. To be a good assembler programmer, you need to be adept at using hexadecimal numbers. The first skill you need is to be able to count in hexadecimal. That’s easy. After 9 we get A, B, C, D, E, and F (10-15). In multidigit integers, after F, the digit resets to 0 and there is a carry of 1 in the next column. So, after 1F we get 20, after 3FF we get 400, after 1A8CF we get 1A8D0.

Object Code

Next you see the bytes of machine code (or object code) that were generated by the assembler. Assembler instructions are either 2, 4, or 6 bytes long. Each byte is represented by two hex digits – eight bits – one byte. The first instruction in our first program was the following `STM` – store multiple instruction,

```
90EC D00C            0000C     7          STM   R14,R12,12(R13)
```

The assembler generated four bytes - `90EC D00C`. The first byte of every object code instruction is the operation code or op-code. For some instructions, the last byte is also part of the operation code, but STM has a one-byte op-code – 90. Every object code instruction follows an instruction format that explains what each byte represents. Later we will examine a few common instruction formats. Instruction formats can help us learn how whole classes of instructions work.

Addr1 Addr2

The next two columns are mostly for your convenience. They represent location counter values for fields in the current instruction. They can help you find a symbol in a listing that extends for many pages. Since the programs we are writing are fairly small, we can ignore these fields for the moment.

Stmt

Each statement is numbered in decimal starting with 1. Notice there are some jumps in the sequencing above, for example after 15 we see 22. That's because statement 15 was an `OPEN` macro. Macros can generate multiple lines of code and since they can usually be assumed to be correct, they are often not printed. Because we also coded `PRINT   ON,NODATA,NOGEN` , the generated lines were suppressed in the listing. If you want to see these lines, change `NOGEN` to `GEN` and reassemble your code.

Removing Errors in an Assembler Listing

I've changed the first program, adding a few errors and reassembled it.

```
000000                00000 00230     3 ASMSKEL  CSECT
                                      4 **********************************************************************
                                      5 * STANDARD ENTRY LINKAGE
                                      6 **********************************************************************
000000 90EC D00C            0000C     7          STM   R14,R12,12(R13)        SAVE CALLER'S REGS
000004 0DC0                           8          BASR  R12,0                  ESTABLISH...
                 R:C  00006           9          USING *,R12                  ADDRESSABILITY
000006 4120 C1DE            001E4    10          LA    R2,SAVEAREA            POINT AT MY SAVE AREA
00000A 5020 D008            00008    11          ST    R2,8(,R13)             FORWARD CHAIN MINE FROM CALLER
00000E 50D0 C1E2            001E8    12          ST    R13,SAVEAREA+4         BACK-CHAIN CALLER'S FROM MINE
000012 18D2                          13          LR    R13,R2                 SET 13 FOR MY SUBROUTINE CALLS
                                     14 **********************  BEGIN LOGIC  *********************************
000014 4D10 C016            0001C    15          OPEN  (FILEIN,(INPUT))       OPEN THE INPUT FILE
00001E 0700                          22          OPEN  (FILEOUT,(OUTPUT))     OPEN THE OUTPUT FILE
00002A 4110 C07E            00084    29          GET   FILEIN,RECIN           READ A RECORD
                      0003A          38 LOOP     EQU   *
                                     39          MCV   RECOUT,RECIN           COPY DATA TO OUTPUT RECORD
** ASMA057E Undefined operation code - MCV
** ASMA435I Record 19 in KC02486.KC02486A.JOB06662.D0000101.? on volume:
00003A 4110 C0DE            000E4    40          PUT   FILEOUT,RECOUT         WRITE THE RECORD TO THE FILE
00004A 4110 C07E            00084    49          GET   FILEIN,RECIN           READ THE NEXT RECORD
00005A A7F4 FFF0            0003A    58          J     LOOP                   JUMP BACK AND DO THIS ALL AGAIN
                      0005E          59 EXIT     EQU   *
00005E 0700                          60          CLOSE FILEIN                 CLOSE THE INPUT FILE
00006A 0700                          67          CLOSE FILEOUT                CLOSE THE OUTPUT FILE
                                     74 ********************** STARDARD EXIT ********************************
000076 58D0 C1E2            001E8    75          L     R13,SAVEAREA+4         POINT TO CALLER'S SAVE AREA
00007A 98EC D00C            0000C    76          LM    R14,R12,12(R13)        RESTORE CALLER'S REGS
```

```
00007E 41F0 0000            00000    77          LA    R15,0                  SET RETURN CODE REG 15 = 0
000082 07FE                          78          BR    R14                    RETURN TO CALLER
                                     79 *********************  DATA AREAS   *********************************
                                     80 FILEIN   DCB   DSORG=PS,                                              X
                                                       MACRF=(GM),                                            X
                                                       DEVD=DA,                                               X
                                                       DDNAME=FILEIN,                                         X
                                                       RECFM=FB,                                              X
                                                       EODAD=EXIT,                                            X
000084 0000000000000000                                LRECL=80
                                    122 FILEOUT  DCB   DSORG=PS,                                              X
                                                       MACRF=(PM),                                            X
                                                       DEVD=DA,                                               X
                                                       DDNAME=FILEOUT,
** ASMA431W Continuation statement may be in error - continuation indicator column is blank.
** ASMA435I Record 42 in KC02486.KC02486A.JOB06662.D0000101.? on volume:
                                    164                RECFM=FB,                                              X
LRECL=80
** ASMA141E Bad character in operation code - RECFM=FB,
** ASMA435I Record 44 in KC02486.KC02486A.JOB06662.D0000101.? on volume:
000144                              165 RECIN    DS    CL80       INPUT AREA FOR RECORDS
000194                              166 RECOUT   DS    CL80       OUTPUT AREA FOR RECORDS
0001E4 0000000000000000             167 SAVEAREA DC    18F'0'     AREA FOR MY CALLEE TO SAVE & RESTORE MY REGS
000230                              168          LTORG
                                    169          YREGS
000000                              188          END   ASMSKEL
```

If a program has assembly errors, you will get a report of the errors by line number at the end of the listing. Here is the listing of errors for the code above.

```
Statements Flagged
    39(P1,19), 122(P1,42), 164(P1,44)

      3 Statements Flagged in this Assembly        8 was Highest Severity Code
```

You can see that the errors occurred at line numbers 39, 122, and 164. As a practical matter, I rarely look for errors this way. Since every error message is prefixed with the letters `ASMA`, I find it easier to issue a *find command* on the listing for `ASMA` to find the first occurrence, and a *repeat find (F5)* for all the others.

The first error message above is for an undefined op-code. I coded `MCV` instead of `MVC`. The second error is for a continuation – a common error for beginners. I failed to put a non-blank character in column 72. The DCB statement with all the parameters is a single statement that has been continued on multiple lines for readability. The comma error also generated a third error on line 164.

The first few errors in a listing are usually the most important ones. One error can lead to tens of others, so correcting the early errors may fix many later ones. My advice is to correct the first few errors and then reassemble the program. It usually takes multiple iterations to remove all the assembly errors.