3. COBOL COMPILER, ASSEMBLER AND BINDER

3.1 Introduction to the COBOL Compiler, Assembler and Binder

Nearly every mainframe programmer will access and use the COBOL compiler and the binder (new to many older mainframe programmers, the binder contains both the loader and the linkage editor) at some time in their career. In fact, many access both daily. Depending on their responsibilities, some of the lucky ones will even access the high-level assembler once in a while. We often refer to these three modules as "common modules" because they are so commonly used.

Most installations have user-defined JCL procedures and offer IBM’s standard procedures to, for example, compile a COBOL program and then send it through the binder to create an executable program object in a subsequent job step. A fetch step of any job can then execute the program object created in the two steps mentioned above. These three steps, i.e., COBOL compiler followed by the binder step followed by the fetch and execute of the program object, are commonly used in this class. Why? Because it is believed critical that all mainframe programmers know how to implement the three common modules without using pre-defined JCL procedures.

To that end, the three common modules are presented here, each with its basic JCL requirements. The student will learn the names of the modules, which DD cards are required to successfully utilize the modules, and what each of the DD cards represents to each of the three respective common modules.

Please note that the term "load module" may be accidentally used when referring to a "program object" but this should be avoided whenever possible.

3.2 The `LNKLST` (Linklist)

At many installations, there is a collection of commonly used program objects stored in a concatenation of system and user-defined PDSE load libraries known as the `LNKLST`, or "Linklist". `LNKLST` allows common modules to be accessed and used without having to specify where they are stored. The `LNKLST` is part of IBM's `SYS1.LINKLIB` which, on the mainframe, is the main software library.

Use of a `LNKLST` allows the omission of a `STEPLIB` from a job step executing any one of the three common modules, and many others not described here. Nothing need be specified in JCL to have access to the program objects that are among those found in the `LNKLST`.

3.3 COBOL Compiler

The COBOL compiler takes COBOL source code and translates it first to assembler and then to machine language. The output is a non-executable object module. In this course we use one of the most recent versions of the IBM COBOL compiler for z/OS, version 5.1.0.

Module Name: `IGYCRCTL`   (there are other names, or aliases, but please use this one)

Required Region: `REGION=0M*`

Recommended Parameters: `PARM=APOST**`

Required DD Statements:

- `STEPLIB`

  Indicates the data set where the COBOL compiler is stored. It is not required if `IGYCRCTL` is included in the installation's `LNKLST`.

- `SYSLIB`

  Indicates the data set or data sets known as copy libraries and sometimes called "copylibs". It is only required if the COBOL `COPY` statement is used in the COBOL program itself.

- `SYSIN`

  Indicates the COBOL source program. Can be either instream or a member of a source library PDS or PDSE. Note that the member name must match that on the `PROGRAM-ID` in the COBOL program.

- `SYSLIN`

  Indicates where the non-executable object module created by the COBOL compilation process is to be written or stored. This is often a temporary data set that is passed to a subsequent binder step. It should usually require no more than `SPACE=(CYL,(1,1))` and `DISP=(MOD,PASS)`.

- `SYSPRINT`

  Indicates the data set to which the program's source listing and messages from the compilation process are to be written. This is often written to `SYSOUT=*`, or standard output.

- `SYSUT1` through `SYSUT15`

  Indicates work data sets used by the compiler during the compilation process. These should be deleted at the end of the step and should usually require no more than `SPACE=(CYL,(1,1))` each. These data sets are often referred to as "scratch pads".

- `SYSMDECK`

  Indicates a data set that will contain a copy of the updated input source after library processing. Like the `SYSUT` data sets above, it should usually require no more than `SPACE=(CYL,(1,1))`.

Note that, when coding JCL job steps, the order of the steps DOES matter and when coding JCL within a single job step, the order of the `DD` cards DOES NOT matter.

It is a good practice to follow the `EXEC` card with a `STEPLIB` and/or a `SYSLIB` when necessary. Follow these with the inputs, from most important to least. Then, follow the inputs with the outputs, from most important to least. Finally, for the COBOL compiler, put the 15 work data sets followed immediately by the `SYSMDECK` data set at the end of the job step.

*`REGION=0M` gives the job all the storage available below and above the 16 megabyte line but the resulting size of the region below and above 16 megabytes is installation-dependent. When specified, the MEMLIMIT (memory limit) is set to NOLIMIT.

**The parameter `APOST`, passed to the COBOL compiler is short for apostrophe or, as we call them, single quotes or "tick" marks. This means that we can use tick marks to delimit all of the character strings in our COBOL source code. Please be sure to use tick marks and NOT double quotes throughout your code.

3.4 High-Level Assembler

Module Name: `ASMA90` (there are other names, or aliases, but please use this one)

The high-level assembler, or assembler, takes assembly language source and translates it to machine language. Like that of the COBOL compiler, the output is a non-executable object module.

Required Parameter: `PARM=ASA`

Required `DD` Statements:

- `STEPLIB`

  Indicates the data set where the assembler is stored. It is not required if `ASMA90` is included in the installation's `LNKLST`.

- `SYSLIB`

  Indicates the data sets where IBM macros are to be found. IBM's principal macro library is named `SYS1.MACLIB`. Some installations concatenate their own macro library or libraries to `SYS1.MACLIB`.

- `SYSIN`

  Indicates the assembly language source program. Can be either instream or a member of a source library PDS or PDSE. Note that the member name must match that of the program's `CSECT`.

- `SYSLIN`

  Indicates where the non-executable object module created by the assembly process is to be written or stored. This is often a temporary data that is passed to a subsequent binder step.

  As taken directly from the IBM cataloged procedures for the assembler, it should usually require no more than `SPACE=(3040,(40,40),,,ROUND)` and `DISP=(MOD,PASS)`.

- `SYSPRINT`

  Indicates the data set to which the program's source listing and messages from the assembly process are to be written. This is often written to `SYSOUT=*`, or standard output.

- `SYSUT1`

  Work, or scratch, data set used by the assembler during the assembly process. It should be deleted at the end of the step and requires, as taken directly from the IBM cataloged procedures for the assembler, `SPACE=(16384,(120,120),,,ROUND)`.

3.5 Binder

Module Name: HEWL or HEWLH096 or HEWLKED or IEWBLINK or IEWL or LINKEDIT (use HEWL)

The binder takes a non-executable object module as input and produces an executable version of the program known as a program object. The binder **DOES NOT** execute the load module.

Using the binder, the program object produced is saved as a member in a partitioned data set extended (PDSE), commonly known as a program library, "load library", or simply as a "loadlib". The program object member produced by the binder can then be executed whenever it is needed. It is most common to create this “permanent” program object once the program has been thoroughly tested and is ready to promote to the next level of development on its way to being implemented in production.

Required `DD` Statements:

- `STEPLIB`

  Indicates the data set where the binder is stored. It is not required if `HEWL` is included in the installation’s `LNKLST`.

- `SYSLIN`

  One or more object modules.

- `SYSLIB`

  This IBM data set contains Language Environment resident library routines that will be automatically included as secondary input into your executable program. This DD should be set to:

  ```
  CEE.SCEELKED,DISP=SHR
  ```

- `SYSLMOD`

  Load module, or program object, created by the binder. Must be stored as a PDSE member.

  A fail-safe way to code the `SYSLMOD` card is as follows:

  ```
  //SYSLMOD  DD DSN=KC0nnnn.CSCI465.LOADLIB(lmodname),
  //            SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY,
  //            DISP=(MOD,KEEP,KEEP)
  ```

  In this `DD` card, the `DISP=(MOD,KEEP,KEEP)` works if the data set already exists. If it does not, it will be allocated with the suggested `SPACE` and the `DSNTYPE=LIBRARY` will insure it is a PDSE that is allocated. `lmodname` is the name of the load module. It must match the program's name.

- `SYSPRINT`

  Messages from the binder.

Note that the binder has replaced the linkage editor and batch loader programs as the system default linker and linker-loader, respectively. The binder assumes all of the functions of the other two linking programs. Invoking any of the common linkage editor or batch loader entry points, such as IEWL, HEWL, LINK, and LOADER, will result in execution of the binder.

While the binder includes all of the functions of the linkage editor and batch loader, it is not fully compatible with those programs. It was developed in response to many customer, vendor and internal requirements requesting relief from various restrictions and processing anomalies in the older programs. The binder attempts to satisfy many of those requirements as well as provide a consistent processing model. As a result it provides a set of externals, which is similar but not identical to the linkage editor and batch loader externals.

The linkage editor and batch loader are also available in z/OS. There are no plans to withdraw either of those programs at this time, but all users are encouraged to begin using the binder as early as possible. In cases where the binder appears unsuitable for a specific application, the older programs are unchanged and can be invoked by entry names HEWLKED or HEWLF064 (linkage editor) or HEWLDIA (batch loader). Note, however, that all future enhancements will be made to the binder and loader exclusively. Other IBM products might have dependencies on functions provided only by these components.

3.6 Binding Program Objects

The following is an excerpt from *IBM's z/OS MVS Program Management: User's Guide and Reference SA23-1393-00*:

Note that this document indicates that there is a difference between a load module and a program object.

You can use the binder to:

- Convert object or load modules, or program objects, into a program object and store the program object in a partitioned data set extended (PDSE) program library or in a z/OS UNIX file.

- Convert object or load modules, or program objects, into a load module and store the load module in a partitioned data set (PDS) program library. This is equivalent to what the linkage editor can do with object and load modules.

- Convert object or load modules, or program objects, into an executable program in virtual storage and execute the program. This is equivalent to what the batch loader can do with object and load modules.

The binder processes object modules, load modules and program objects, *link-editing* or *binding* multiple modules into a single load module or program object. Control statements specify how to combine the input into one or more load modules or program objects with contiguous virtual storage addresses. Each object module can be processed separately by the binder, so that only the modules that have been modified need to be recompiled or reassembled. The binder can create programs to be loaded into either 24-bit address or 31-bit address storage (for example, `RMODE=24` or `RMODE=ANY`) and programs that execute in 24-bit, 31-bit, or 64-bit addressing mode (including support for 8-byte address constants). The binder can also create overlay load modules or program objects. Programs can be stored in program libraries and later brought into virtual storage by the program management loader.

The binder can also combine basic linking and loading services into a single job step. It can read object modules, load modules and program objects from program libraries into virtual storage, relocate the address constants, and pass control directly to the program upon completion. When invoked in this way, the binder does not store any of its output in program libraries after preparing it for execution. Like the batch loader, you can use the binder for high-performance loading of modules that do not need to be stored in a program library.

3.7 How a Load Module is Created

The following is an excerpt from *IBM's z/OS Basic Skills: Application Programming on z/OS*:

Note that this document also indicates that there is a difference between a load module and a program object.
In processing object decks and load modules, the linkage editor assigns consecutive relative virtual storage addresses to control sections, and resolves references between control sections. Object decks produced by several different language translators can be used to form one load module.

An output load module is composed of all input object decks and input load modules processed by the linkage editor. The control dictionaries of an output module are, therefore, a composite of all the control dictionaries in the linkage editor input. The control dictionaries of a load module are called the composite external symbol dictionary (CESD) and the relocation dictionary (RLD). The load module also contains the text from each input module, and an end-of-module indicator.

Figure 3.7.1 shows the process of compiling two source programs: PROGA and PROGB. PROGA is a COBOL program and PROGB is an assembler language program. PROGA calls PROGB. In this figure we see that after compilation, the reference to PROGB in PROGA is an unresolved reference. The process of link-editing the two object decks resolves the reference so that when PROGA is executed, the call to PROGB will work correctly. PROGB will be transferred to, it will execute, and control will return to PROGA, after the point where PROGB was called.

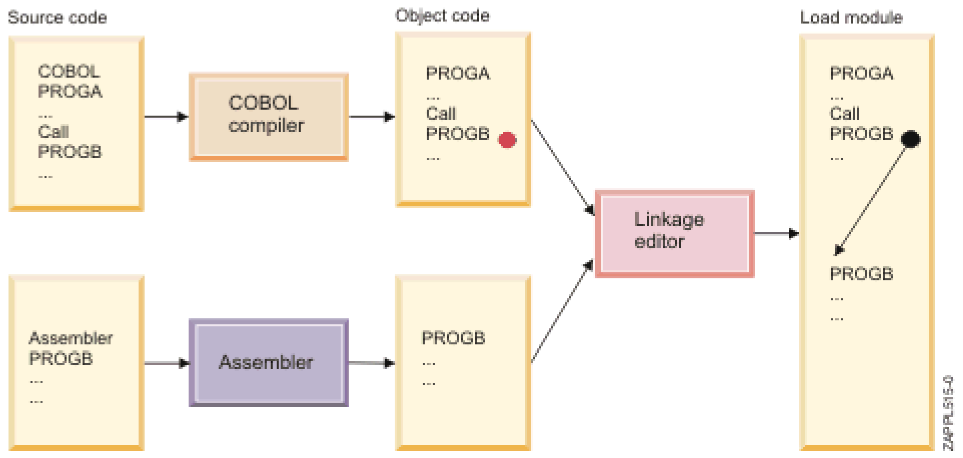

Figure 3.7.1. Resolving references during load module creation

The Binder

The binder provided with z/OS performs all of the functions of the linkage editor. The binder link-edits (combines and edits) the individual object decks, load modules, and program objects that comprise an application and produces a single program object or load module that you can load for execution. When a member of a program library is needed, the loader brings it into virtual storage and prepares it for execution.

You can use the binder to:

- Convert an object deck or load module into a program object and store it in a partitioned data set extended (PDSE) program library, or in a z/OS UNIX file.

- Convert an object deck or program object into a load module and store it in a partitioned data set (PDS) program library. This is equivalent to what the linkage editor does with object decks and load modules.

- Convert object decks or load modules, or program objects, into an executable program in virtual storage and execute the program. This is equivalent to what the batch loader does with object decks and load modules.

The binder processes object decks, load modules and program objects, link-editing or binding multiple modules into a single load module or program object. Control statements specify how to combine the input into one or more load modules or program objects with contiguous virtual storage addresses. Each object deck can be processed separately by the binder, so that only the modules that have been modified need to be recompiled or reassembled. The binder can create programs in 24-bit, 31-bit and 64-bit addressing modes.

You assign an addressing mode (**AMODE**) to indicate which hardware addressing mode is active when the program executes. Addressing modes are:

- 24, which indicates that 24-bit addressing must be in effect
- 31, which indicates that 31-bit addressing must be in effect
- 64, which indicates that 64-bit addressing must be in effect
- ANY, which indicates that 24-bit, 31-bit, or 64-bit addressing can be in effect
- MIN, which requests that the binder assign an **AMODE** value to the program module.

The binder selects the most restrictive **AMODE** of all control sections in the input to the program module. An **AMODE** value of 24 is the **input to the program module. An AMODE value of 24 is the most** restrictive; **an AMODE value of ANY is the least** restrictive.

All of the services of the linkage editor can be performed by the binder.

Binder and Linkage Editor

The binder relaxes or eliminates many restrictions of the linkage editor. The binder removes the linkage editor's limit of 64 aliases, allowing a load module or program object to have as many aliases as desired. The binder accepts any system-supported block size for the primary (SYSLIN) input data set, eliminating the linkage editor's maximum block size limit of 3200 bytes. The binder also does not restrict the number of external names, whereas the linkage editor sets a limit of 32767 names.

By the way, the prelinker provided with z/OS Language Environment® is another facility for combining object decks into a single object deck. Following a pre-link, you can link-edit the object deck into a load module (which is stored in a PDS), or bind it into a load module or a program object (which is stored in a PDS, PDSE, or zFS file). With the binder, however, z/OS application programmers no longer need to pre-link, because the binder handles all of the functionality of the pre-linker. Whether you use the binder or linkage editor is a matter of preference. The binder is the latest way to create your load module.

The primary input, required for every binder job step, is defined on a DD statement with the ddname SYSLIN. Primary input can be:

- A sequential data set
- A member of a partitioned data set (PDS)
- A member of a partitioned data set extended (PDSE)
- Concatenated sequential data sets, or members of partitioned data sets or PDSEs, or a combination
- A z/OS UNIX file.

The primary data set can contain object decks, control statements, load modules and program objects. All modules and control statements are processed sequentially, and their order determines the order of binder processing. The order of the sections after processing, however, might not match the input sequence. Figure 3.7.2 shows a job that can be used to link-edit an object deck. The output from the LKED step will be placed in a private library identified by the SYSLMOD DD. The input is passed from a previous job step to a binder job step in the same job (for example, the output from the compiler is direct input to the binder).

```
//LKED   EXEC PGM=IEWL,PARM='XREF,LIST',             IEWL is IEWBLINK alias
//           REGION=2M,COND=(5,LT,prior-step)
//*
//*        Define primary input
//*
//SYSLIN   DD  DSN=&&OBJECT,DISP=(MOD,PASS)          required
//         DD  * inline control statements           optional
   INCLUDE  PRIVLIB(membername)
   NAME     modname(R)
/*
//*
//*        Define secondary input
//*
//SYSLIB   DD  DSN=language.library,DISP=SHR         optional
//PRIVLIB  DD  DSN=private.include.library,DISP=SHR  optional
//*
//*        Define output module library
//*
//SYSLMOD  DD  DSN=program.library,DISP=SHR          required
//*
//SYSPRINT DD  SYSOUT=*                              required
//*
//SYSTERM  DD  SYSOUT=*                              optional
//
```

Figure 3.7.2. binder JCL example

An explanation of the JCL statements follows:

EXEC

Binds a program module and stores it in a program library. Alternative names for IEWBLINK are IEWL, LINKEDIT, EWL, and HEWLH096. The PARM field option requests a cross-reference table and a module map to be produced on the diagnostic output data set.

SYSLMOD

Defines a temporary data set to be used as the output module library.

SYSPRINT

Defines the diagnostic output data set, which is assigned to output class A.

SYSLIN

Defines the primary input data set, &&OBJECT, which contains the input object deck; this data set was passed from a previous job step and is to be passed at the end of this job step.

INCLUDE

Specifies sequential data sets, library members, or z/OS UNIX files that are to be sources of additional input for the binder (in this case, a member of the private library PRIVLIB).

NAME

Specifies the name of the program module created from the preceding input modules, and serves as a delimiter for input to the program module. (R) indicates that this program module replaces an identically named module in the output module library.

3.8 The Binder and Subprograms

The program object of any subprogram that is statically called must be concatenated to the binder step's SYSLIN DD card along with the object module of the calling program. Here is an example:

```
//JSTEP02  EXEC PGM=HEWL,COND=(0,LT)
//*
//SYSLIB   DD DSN=CEE.SCEELKED,DISP=SHR
//*
//SYSLIN   DD DSN=&&OBJMOD1,DISP=(OLD,DELETE,DELETE)
//         DD DSN=&&OBJMOD2,DISP=(OLD,DELETE,DELETE)
//*
//SYSLMOD  DD DSN=KC0nnnn.CSCI465.LOADLIB(MAINPGM),
//            SPACE=(1024,(50,20,1)),DSNTYPE=LIBRARY,
//            DISP=(MOD,KEEP,KEEP)
//*
//SYSPRINT DD SYSOUT=*
//
```

In the example above, there are two object modules concatenated on the SYSLIN DD card.

As an example, we can say that &&OBJMOD1 is the object module of a program named MAINPGM which will be the caller, or driver, program. And we can say that &&OBJMOD2 is the object module of a subprogram called statically by MAINPGM. The output program object written to the program library PDSE will contain the executables of both caller and called.

Subprograms called dynamically are NOT link-edited together with the caller. They are link-edited on their own and their own program object is written to a program library. This program library only need be referenced on the STEPLIB DD in the fetch and execution of the caller program object. More on this follows in discussing the fetch step.

Note that the SYSLMOD statement above includes both SPACE and DSNTYPE parameters and a DISP parameter with MOD,KEEP,KEEP. This is in the case that the LOADLIB does not exist. If it does not, it will be allocated before the new load module is added to it.

3.9 The Fetch Step

Whenever an existing load module or program module is needed in a job, a fetch step is used to execute it. This is sometimes called "program fetch". The following is an example of a job with a single step executing a load module named `BITMANIP`:

```
//KC0nnnnA JOB ,'JCL EXAMPLE',MSGCLASS=H
//*
//*********************************************************
//*                                                       *
//*  JSTEP01 - PROGRAM FETCH STEP                         *
//*                                                       *
//*********************************************************
//*
//JSTEP01  EXEC PGM=BITMANIP,COND=(0,LT)
//*
//STEPLIB   DD DSN=KC0nnnn.CSCI465.LOADLIB,DISP=SHR
//*
//INDATA    DD DSN=KC01234.CSCI465.DATA(ENCODED1),DISP=SHR
//*
//PRINTOUT  DD SYSOUT=*
//*
//SYSUDUMP  DD SYSOUT=*
//
```

Of course, many more steps and as many program fetches as desired can be – and commonly are – included in a single job as desired.

Note that the program fetched in `JSTEP01`, of the job above is named `BITMANIP` and it probably reads data from member `ENCODED1` stored in a PDS or PDSE named `KC01234.CSCI465.DATA`. It also probably writes data to `SYSOUT=*` on DD card `PRINTOUT`.