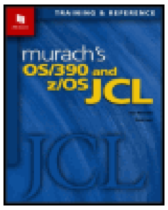


Murach's OS/390 and z/OS JCL

by Raul Menendez and Doug Lowe

Table of Contents

Chapter 17: How to Use JCL Procedures to Compile and Test Programs

In chapter 9, you learned how to create and use your own JCL procedures. Now, you'll learn how to use the program development procedures that are supplied by IBM. These are the procedures that let you compile and test the programs that you develop.

As you read this chapter, you'll see that the focus is on the procedures that apply to COBOL and CICS program development. We took that approach because most mainframe development is still being done with those languages. Once you learn how to use the procedures for those languages, though, you should be able to apply those skills to the procedures for any other language.

Program development concepts

When you develop a program for an IBM mainframe with a language like COBOL or assembler language, you run a procedure that converts your source program into a machine language program that can be run by the system. Within that procedure, two or more job steps are run. In the topics that follow, you'll learn more about some of the common procedures that are used for program development.

A compile−link−and−go procedure

Usually, it takes three steps to compile and test a program on an IBM mainframe system. This is illustrated by the system flowchart in figure 17−1, which presents the three steps that are required for a COBOL program. This is commonly referred to as a *compile−link−and−go procedure*.

In the first step, the COBOL *compiler* reads the program you've written, called the *source program*, and converts it to an *object module* that is stored on disk in an *object library*. If the source program uses *copy members* (pieces of source code that are copied into the program), they're retrieved from a *copy library* and inserted into the program during this step.

Typically, the compiler also produces printed output as part of this step. Among other things, this output can include a *compiler listing* that shows the COBOL statements in the program, including any copy members. The listing also includes a description of any coding errors that make it impossible for the compiler to create the object program.

If the program compiles without any errors, the second step is performed. In this step, a program called a *linkage editor* (or *link editor*) links the object module with any subprograms that it requires. These can be system subprograms or subprograms written by you or another programmer. Either way, the subprograms must be compiled into object modules that are stored in object libraries before the linkage editor can link them.

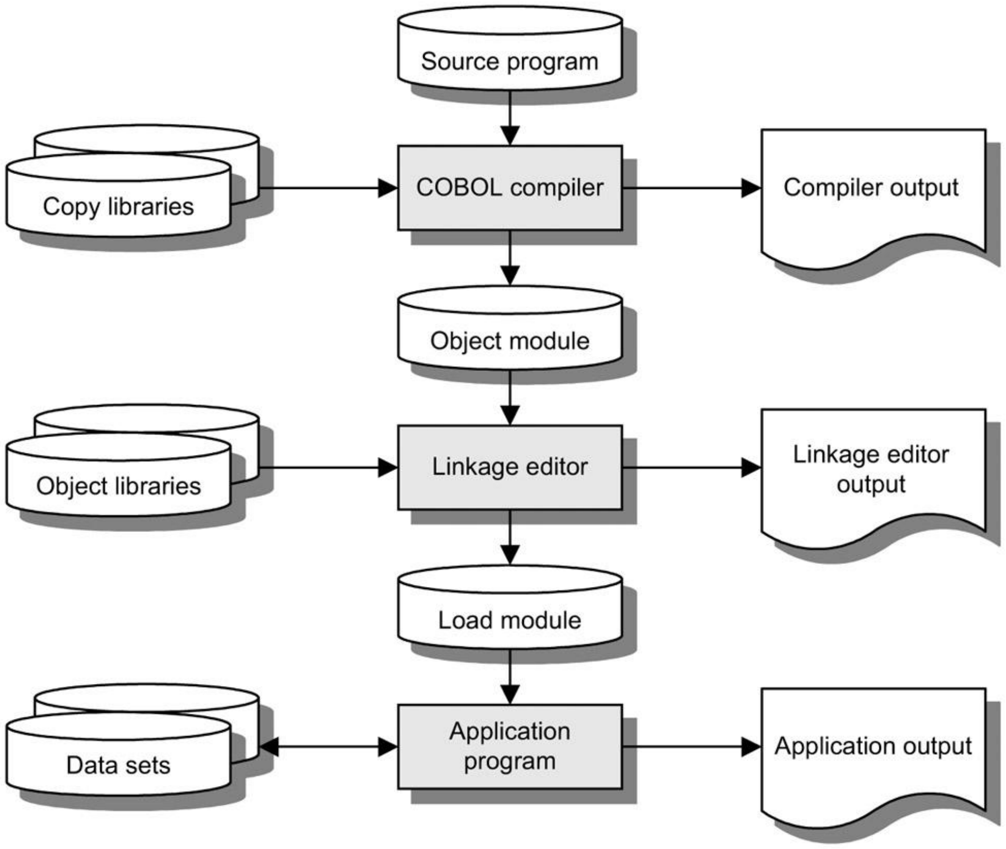
The output of the link−edit step is a *load module* that is stored on disk. This is your program in a form that can be executed. The linkage editor can also create printed output that identifies the programs included in the load module.

In the third step, the load module is run. The input to this program is whatever data sets the program calls for, and the output is whatever data sets the program produces. Throughout this book, you've learned how to allocate the data sets for a program and execute its load module.

This procedure is the same for other languages too, although some of the terms may differ. If you're using assembler language, for example, the compiler is usually referred to as an *assembler* and compiling is referred to as *assembling* (although many people use the terms *compiler* and *compile* for both compilers and assemblers). Also, the copy library is referred to as the *source statement library*.

Figure 17−1: A compile−link−and−go procedure

The system flowchart for a COBOL compile−link−and−go procedure



Description

- In step 1, the COBOL *compiler* compiles the *source program* into an *object module*. If necessary, it gets groups of source statements called copy members from the *copy library.* When you use other programming languages, the copy library is called the *source statement library.*
- In step 2, the *linkage editor* links edits the object module with any other object modules that it needs. These can be IBM−supplied object modules or object modules for subprograms that have been written by application programmers. The result is an executable program called a *load module*.
- In step 3, the load module is executed. The program then processes its input data sets and produces its output data sets, which can include printed output.
- When you develop a program with assembler language, the compiler is often called an *assembler* and compiling is called *assembling*. Otherwise, this procedure works the same way for an assembly as it does for a compilation.
- A procedure like this can be referred to as a *compile−link−and−go procedure*.

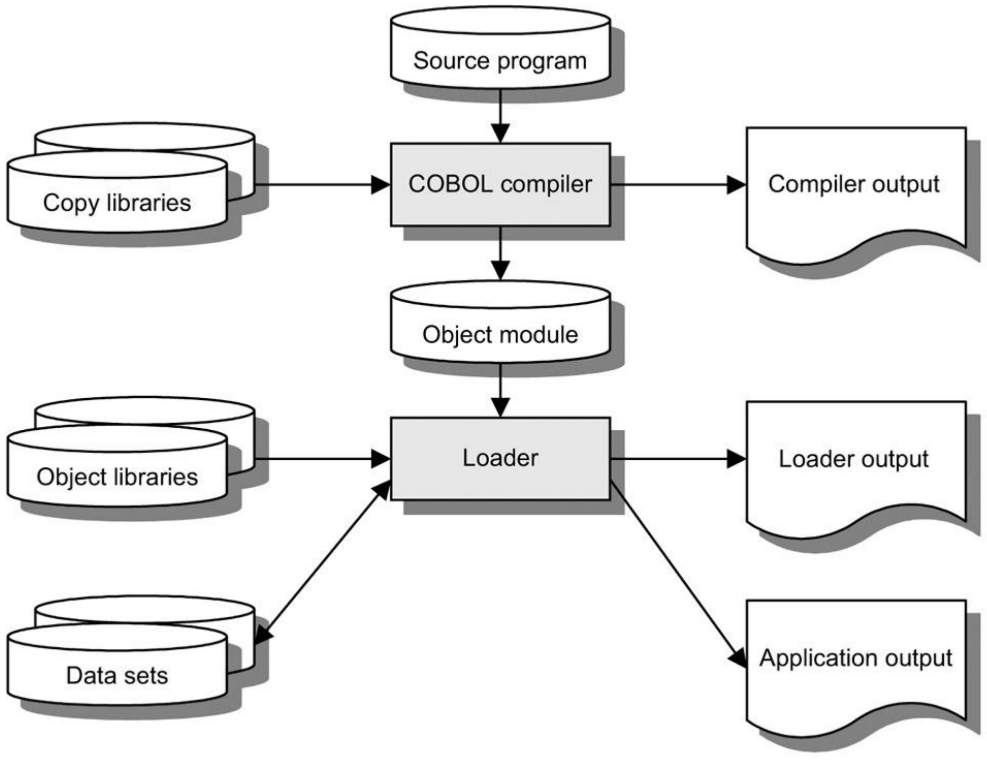
A compile–load–and–go procedure

A *compile–load–and–go procedure* (or just *compile–and–go*) is similar to the procedure you've just seen. However, it only requires two steps. In the second step, a program called the *loader* link edits the object modules, loads your executable program into storage, and runs it…without saving a load module. This is illustrated by the system flowchart in figure 17–2.

You can use this type of procedure when you're testing a program, and you're making changes to the source program after each test run. If you want to test the same version of the program more than once, though, you should use a *compile–and–link procedure* to create a load module. Then, you can run that load module repeatedly with separate JCL as you test your program. This is also the way production programs are run.

Figure 17–2: A compile–load–and–go procedure

The system flowchart for a COBOL compile–load–and–go procedure



Description

- Like the linkage editor, the *loader* links the object modules for a program. However, it doesn't actually create a load module on DASD. Instead, it links the object modules, loads the executable program into storage, and executes it…all in one step.

- A procedure like this can be referred to as a *compile–load–and–go* or just a *compile–and–go procedure*. It is only used for testing, because production programs are run from load modules.

A translate–compile–and–link procedure

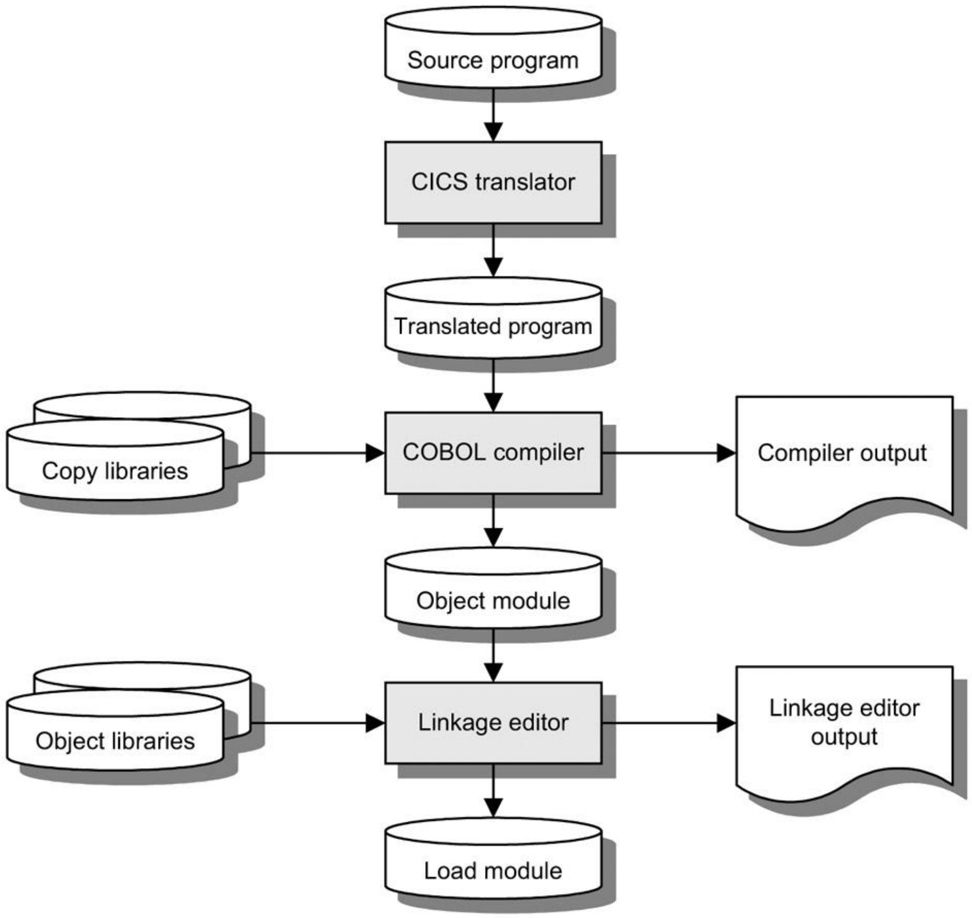
When you use some programming subsystems like CICS, DB2, or IMS, you embed their commands within a language like COBOL, PL/I, or assembler language. For instance, most CICS programs are developed by embedding CICS commands within COBOL programs. Then, the CICS commands need to be translated into COBOL before the COBOL program can be compiled and link edited into a load module.

This is illustrated by the procedure in figure 17–3. Here, the source program is first processed by the CICS *translator*, which converts the CICS code into COBOL. Then, the COBOL compiler compiles the source code into an object module, and the linkage editor links the object modules into a load module. The load module can then be run under the CICS subsystem.

Because CICS programs can only be run under the CICS subsystem, you can't run CICS load modules as JCL batch jobs. As a result, you can't use a link–and–go or load–and–go procedure for CICS programs. Instead, you set up the appropriate CICS tables for running the load modules, which you can learn how to do in our CICS book, *Murach's CICS for the COBOL Programmer*.

Figure 17–3: A translate–compile–and–link procedure

The system flowchart for a CICS translate–compile–and–link procedure



Description

- When you develop a CICS program, you embed CICS commands within another programming language like COBOL, PL/I, or assembler language.
- Before a CICS program can be compiled, it must be translated into the base programming language. This is done by the *CICS translator*.
- For a CICS/COBOL program, the CICS translator converts CICS commands to COBOL Move and Call statements that can be compiled by the COBOL compiler.
- After a CICS program is translated, it is compiled and link edited just like any other source program.
- CICS programs run under the CICS subsystem, not as OS/390 batch jobs. As a result, you can't run CICS load modules with JCL.

The IBM procedures for program development

To make it easier for you to develop programs, IBM provides JCL procedures for common program-development jobs. In the topics that follow, you'll learn more about them.

A summary of the IBM procedures

Figure 17-4 summarizes the procedures for six of the programming languages that IBM supports. As you can see, there are four procedures for each of these languages. You can use the *compile-only procedure* when you want to create an object module for a subprogram. You can use the *compile-and-link procedure* when you want to create a load module that you can repeatedly execute as a batch JCL job. And you can use the other two procedures when you want to compile, link, and run a program.

Besides these procedures for programming languages, IBM provides other procedures for program development. Later in this chapter, for example, you'll see the translate-compile-and-link procedure for CICS/COBOL programs. You'll also see a procedure for creating CICS mapsets.

Figure 17-4: A summary of the IBM procedures for program development

IBM procedures for six programming languages

Language	Compile only	Compile & link	Compile, link & go	Compile & go
COBOL for OS/390	IGYWC	IGYWCL	IGYWCLG	IGYWCG
VS COBOL II	COB2UC	COB2UCL	COB2UCLG	COB2UCG
High Level Assembler	ASMAC	ASMACL	ASMACLG	ASMACG
PL/I for OS/390	IBMZC	IBMZCPL	IBMZCPLG	IBMZCPG
VS Fortran 2	VSF2C	VSF2CL	VSF2CLG	VSF2CG

C for OS/390	EDCC	EDCCL	(n/a)	EDCCBG
C++ for OS/390	CBCC	CBCCL	CBCCLG	CBCCBG

The four types of procedures

- A *compile–only procedure* compiles the source program and usually creates an object module. The name for this type of procedure ends with a C for compile (even if you're using assembler langauge).
- A *compile–and–link procedure* compiles the source program and link edits it. The name for this type of procedure usually ends with CL.
- A *compile–link–and–go procedure* compiles, link edits, and runs the resulting load module. The name for this type of procedure usually ends with CLG.
- A *compile–and–go procedure* compiles the source program and then uses the loader to link and run the program without creating a load module. The name for this type of procedure usually ends with CG.

Description

- For each programming language that is offered with OS/390, IBM supplies the four types of cataloged JCL procedures that are summarized above.
- Besides the procedures for programming languages, IBM supplies other procedures for program development. These include procedures for developing programs for the CICS, DB2, and IMS subsystems as well as for developing CICS mapsets.
- Often, the IBM procedures are modified within a shop to meet its specific requirements.

The step names and data sets for COBOL procedures

To use one of the IBM procedures, you need to know the names of the job steps within the procedure. You also need to know what ddnames you may have to provide DD statements for, depending on your system and program requirements. For instance, figure 17–5 shows the step names and ddnames that are used for the COBOL procedures.

If you're familiar with COBOL program development on IBM mainframes, you know that two different COBOL compilers are available. The older one is called VS COBOL II. The newer one is called COBOL for OS/390. For each of these compilers, four different procedures are available as shown in the last figure. The step names in these procedures are the same, though, except for the compile step, which is either COB2 or COBOL. The ddnames that can be used for the steps in these procedures are the same for both compilers.

To code the JCL for running one of these IBM procedures, you need to provide DD statements for any ddnames required by the job that are not provided for by the IBM procedure. For the compile step, for example, you always need to provide a SYSIN DD statement that allocates the source program (normally, a member of a source library). If your program uses copy members, you also need to provide a SYSLIB DD statement that allocates the copy library.

For the link–edit step, you need to override the SYSLMOD DD statement in the procedure so the load module for your program gets stored in a user library. If your program calls subprograms, you may also need to concatenate a user object library to the system object library by coding a SYSLIB DD statement.

Finally, for the go step, you need to code DD statements for the data sets used by your program. You may also want to allocate data sets for any of the ddnames listed for the GO step in this figure.

To introduce you to the JCL for running a COBOL program–development procedure, this figure provides an example. As you can see, this JCL starts by executing the procedure named IGYWCLG, which is the compile–link–and–go procedure for COBOL for OS/390. This assumes that the procedure is in SYS1.PROCLIB. Otherwise, you need to code a JCLLIB statement that identifies the procedure library.

Next, the JCL provides SYSIN and SYSLIB DD statements for the COBOL step of the procedure. Then, it provides SYSLMOD and SYSLIB DD statements for the LKED step of the procedure. Note here that the SYSLIB statement concatenates a user library with the library that's provided by the IBM procedure because the COBOL program uses object modules from both libraries. Last, the JCL provides for two data sets in the GO step that are required by the application program.

If you remember what you learned in chapter 9 about using procedures, you should understand how this JCL works right now. If not, you may want to review that chapter to refresh your memory. In a moment, though, you'll actually review some of the IBM procedures so you'll get a better idea of what's going on.

Figure 17–5: The step names and data sets for COBOL procedures

Step names for the COBOL procedures

Step	VS COBOL II	COBOL for OS/390
Compile	COB2	COBOL
Link	LKED	LKED
Go	GO	GO

DD statements used with the cataloged procedures

Step	ddname	Description
COB2/COBOL	SYSIN	Source program input for the COBOL compiler.
	SYSLIB	The copy library or libraries.
	SYSLIN	Object module output.
LKED	SYSLIB	The subprogram library or libraries.
	SYSLIN	Object module input.
	SYSIN	Additional object module input.
	SYSLMOD	Load module output.
GO	SYSOUT	Output from DISPLAY statements.
	SYSIN	Input for ACCEPT statements.
	SYSDBOUT	Symbolic debugging output.
	SYSUDUMP	Abnormal termination (or storage dump) output.

	SYSABEND	

JCL that invokes the compile–link–and go procedure for OS/390 COBOL

```
//MM01CLG      JOB   (36512),'R MENENDEZ',NOTIFY=MM01
//STEP1        EXEC PROC=IGYWCLG
//COBOL.SYSIN  DD   DSN=MM01.PAYROLL.SOURCE(PAY4000),DISP=SHR
//COBOL.SYSLIB DD   DSN=MM01.PAYROLL.COPYLIB,DISP=SHR
//*------------------------------------*
//LKED.SYSLMOD DD   DSN=MM01.PAYROLL.LOADLIB(PAY4000),DISP=SHR
//LKED.SYSLIB  DD
//             DD   DSN=MM01.PAYROLL.OBJLIB,DISP=SHR
//*------------------------------------*
//GO.PAYTRAN   DD   DSN=MM01.PAYROLL.TRANS,DISP=SHR
//GO.PAYRPT    DD   SYSOUT=A
```

Description

- To use a cataloged procedure, you need to know the step names used in the procedure. You also need to know what the ddname requirements are.
- You use these step names and ddnames to supply the members, libraries, or data sets needed by a procedure.
- To find out the names of the procedures that are available for a specific compiler, you need to refer to the appropriate reference manual. It will also tell you what the step names and ddname requirements are for each procedure.

The translation and compiler options for CICS and COBOL procedures

When a translator or compiler is installed on an IBM mainframe, various options are set. These are the default options. Some of the IBM procedures, however, override these default options. Then, when you code the JCL for using one of these procedures, you can override the options set by the procedure or the original system defaults.

Some of the CICS and COBOL options are listed in figure 17–6. Most of these are usually set the way you want them, but you may want to override some of them on occasion. In most IBM shops, for example, apostrophes (‘) are used as delimiters instead of quotation marks (“), even though the COBOL standard is quotation marks. If you want to change this setting, you can set the QUOTE option to specify quotation marks or the APOST option to specify apostrophes.

If you want the compiler to write the object module to the data set identified by the SYSLIN DD statement, the OBJECT option needs to be on. If your program uses copy members, the LIB option also needs to be on. Normally, both of these options are on, but if you discover that one has been set to NOOBJECT or NOLIB, you can reset it to OBJECT or LIB. (Another option that can be used for object modules is DECK, but that option is rarely used today, even though you'll still see the NODECK option on compiler listings.)

If you want to change some of the options for the compiler listing, you can set the compiler options in the fourth group in this figure. If, for example, the compiler prints a cross–reference listing that you never use, you can set the NOXREF option. Or, if the compiler isn't printing the cross–reference listing and you want one, you can set the XREF option.

Keep in mind, though, that this figure presents only a few of the translator and compiler options. If you want to review the many options that are provided for an IBM software product, you need to check its reference manual.

When you use an IBM procedure for program development, you can change an option by coding the PARM parameter on the EXEC statement for the procedure. This is illustrated by the example in this figure. Here, three options for the COBOL step within the procedure are changed. Two are turned on, and one is turned off.

When you override any of the options for a procedure step, though, all of the procedure step options are overridden. As a result, you need to include any of the procedure options that you want to keep. To find out what those procedure options are, you can review the procedures, as you'll see in a moment.

You may also want to find out what the default options of the COBOL compiler are. Then, if they're set the way you want them, you don't have to worry about overriding them. To find out what the default options are, you can run a procedure with the PARM parameter set to no options, like this:

```
PARM.COBOL=''
```

This overrides any procedure options so you can review the compiler listing to see what the default options are.

Figure 17–6: The translation and compiler options for CICS and COBOL procedures

Translator options for CICS

Option	Function
QUOTE	Use the ANSI standard quotation mark (").
APOST	Use the apostrophe (').
EDF	The Execution Diagnostic Facility is to apply to the program.
EXCI	The translator is to process EXCI commands.
FLAG	Print the severity of error messages produced by the translator.
LENGTH	Generate a default length if the LENGTH option is omitted from a CICS command.
OOCOBOL	Instructs the translator to accept object–oriented COBOL syntax.
COBOL3	Specifies that the translator is to translate programs compiled by COBOL for OS/390.

Compiler options for VS COBOL II and COBOL for OS/390

Category	Option	Function
Object module	OBJECT	Write object module output to SYSLIN.
	DECK	Write object module output to SYSPUNCH.
Delimiter	QUOTE	Use the ANSI standard quotation mark (").
	APOST	Use the apostrophe (').
Source library	LIB	Allow Copy statements.
Compiler listing	SOURCE	Print source listing.
	OFFSET	Print offset of each Procedure Division verb.
	LIST	

		Print assembler listing of object module.
	MAP	Print offset of each Data Division field.
	XREF	Print sorted cross reference of data and procedure names.
Testing	TEST	Allow interactive debugging.
	FDUMP	Provide formatted dump at abend.

An EXEC statement that turns two compiler options on and one off

```
//STEP1        EXEC PROC=IGYWCLG,PARM.COBOL='APOST,OBJECT,NOXREF'
```

Description

- Translator and compiler options can be set at three levels. First, the default options are set when the software is installed. Second, the IBM procedures can override the default options. Third, your JCL can override the IBM procedure options or the system defaults.
- To turn an option on or off, you code the PARM parameter for a job step. To turn an option on, you code one of the option names shown above. To turn an option off, you precede the name by NO as in NOXREF (QUOTE and APOST are exceptions).
- When you use the PARM parameter to set options, they override all of the options set by the procedure you're using. So if you want to retain any options set in the procedure, you must code them in the PARM parameter too.
- To find out what the default settings are for the compiler options, you can code a PARM parameter with no settings so all the procedure options are overridden. Then, you can see the default settings by reviewing the compiler listing.

How to use some of the IBM procedures

To give you a better idea of how the IBM procedures for program development work, the rest of this chapter presents five procedures and the JCL for using them. Three are COBOL procedures, and two are CICS procedures.

The compile–only procedure for OS/390 COBOL

Figure 17–7 presents the IBM compile–only procedure for COBOL for OS/390 along with the JCL for running it. If you remember what you learned about procedures in chapter 9, you should be able to understand what this procedure is doing.

After the PROC statement sets the values for the symbolic parameters, you'll find three comments. The second one tells you that you must always supply a DD statement for the SYSIN data set. That's the one that contains the source program that's going to be compiled.

After the comments, you can see the single job step for this procedure, which is named COBOL. For this step, the EXEC statement runs a program named IGYCRCTL, which is the COBOL compiler. Note that a PARM parameter isn't coded, so no compiler options are set.

In the DD statements that follow, you can see that the SYSLIN DD statement allocates a temporary data set named &&LOADSET. That's the data set that's used for the object module that is created by the compiler. Because this is a temporary data set, though, it isn't saved when the procedure ends. If you want to save it so it can be used as a subprogram, you need to code a SYSLIN DD statement when you run this program to override the procedure's DD statement.

Because this procedure doesn't include a SYSLIB DD statement, it doesn't provide for the use of copy members. If your program uses them, you also need to provide a SYSLIB DD statement in the JCL that runs this procedure.

The job in the middle of this figure shows you how to run the procedure. In this case, the JCLLIB statement identifies the library that the procedure is stored in when the compiler is installed: SYS1.IGY.SIGYPROC. That way, OS/390 can find the procedure when the job is run. Note, however, that procedures like this are often moved to the SYS1.PROCLIB or another installation−specific library that is automatically searched when a procedure is run. If that's the case in your shop, you don't have to code a JCLLIB statement in your job.

In the EXEC statement for this job, you can see that the IGYWC procedure is executed and the APOST and OBJECT options are set. Since the procedure doesn't set any options, these options will override the default options for the compiler in case that's necessary. Then, the SYSIN DD statement identifies the member of the source library that contains the program to be compiled. The SYSLIB DD statement identifies the copy library that's used. And the SYSLIN DD statement overrides the SYSLIN DD statement in the procedure so the object module is saved in an object library.

Figure 17−7: How to use the compile−only procedure for OS/390 COBOL

The IBM compile−only procedure for OS/390 COBOL

```
//IGYWC  PROC  LNGPRFX='SYS1.IGY',SYSLBLK=3200
//*  COMPILE A COBOL PROGRAM
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD …
//*
//COBOL  EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
```

The JCL to invoke this procedure

```
//MM01CL       JOB    (36512),'R MENENDEZ',NOTIFY=MM01
//JOBPROC      JCLLIB ORDER=SYS1.IGY.SIGYPROC
//STEP1        EXEC   PROC=IGYWC,PARM.COBOL='APOST,OBJECT'
//COBOL.SYSIN  DD     DSN=MM01.PAYROLL.SOURCE(PAY4000),DISP=SHR
//COBOL.SYSLIB DD     DSN=MM01.PAYROLL.COPYLIB,DISP=SHR
//COBOL.SYSLIN DD     DSN=MM01.PAYROLL.OBJLIB(PAY4000),DISP=SHR
```

Where the IBM procedures are located

- Most IBM procedures for program development are initially stored in a procedure library other than SYS1.PROCLIB. If so, you need to use the JCLLIB statement to identify the library that contains the procedure that you want to use.

- Often, commonly–used procedures are moved to SYS1.PROCLIB from the libraries they were initially stored in. In that case, you don't need to specify a procedure library for the job.

The compile–and–link procedure for OS/390 COBOL

Figure 17–8 presents the IBM compile–and–link procedure for COBOL for OS/390 along with the JCL for running it. Here, the PROC statement, the comments, and the COBOL step are similar to what you saw in the compile–only procedure. This means that you need to code a SYSIN DD statement for the source program when you run the procedure. You also need to code a SYSLIB DD statement for the copy library if your program uses one.

You can also code a SYSLIN DD statement that overrides the temporary data set that's used to save the object module produced by the COBOL step. You only need to do that, though, if you want to save the object module after the job has finished. Otherwise, the temporary data set is used as input to the LKED step, which creates the load module.

In the LKED step, however, you can see that the SYSLMOD DD statement for the load module says that it should be saved in a temporary data set named &&GOSET(GO). (You have to substitute the values set by the PROC statement into the symbolic parameters to get this data set name.) Because you want to be able to use this load module after the job has finished, though, that isn't what you want. As a result, you need to override the SYSLMOD DD statement in the JCL for running the procedure.

If your program calls static subprograms, you also need to concatenate your subprogram libraries with the library allocated by the SYSLIB DD statement in the LKED step. (A *static subprogram* is one that's been compiled into an object module and is link edited with the program into a single load module.) You can't just override the SYSLIB statement in the procedure because it identifies the library that contains the system object modules that need to be linked with your COBOL program.

One last point to note is that the LKED step isn't run if 8 is less than the return code for the COBOL step. This means that if the COBOL compiler finds serious errors in the source program, the LKED step isn't run. In that case, you need to correct the errors and try the procedure again.

Here again, the JCL example after the procedure shows you how to run it. To start, the JCLLIB statement identifies the library that the IGYWCL procedure is stored in. Then, for the COBOL step, the SYSIN DD statement identifies the source program member in the source library and the SYSLIB DD statement identifies the copy library.

For the LKED step, the SYSLMOD DD statement identifies the load library that the load module should be stored in. Then, the SYSLIB DD statement concatenates a user object library to the object library that's identified by the procedure (it's SYS1.CEE.SCEELKED when you replace the symbolic parameter with the value set in the PROC statement). That way, the linkage editor can find the system object modules as well as the user object modules.

Figure 17–8: How to use the compile–and–link procedure for OS/390 COBOL

The IBM compile–and–link procedure for OS/390 COBOL

```
//IGYWCL PROC  LNGPRFX='SYS1.IGY',SYSLBLK=3200,
//             LIBPRFX='SYS1.CEE',
//             PGMLIB='&&GOSET',GOPGM=GO
//*  COMPILE AND LINK EDIT A COBOL PROGRAM
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD …
//*
//COBOL   EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED    EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&PGMLIB(&GOPGM),
//             SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
```

The JCL to invoke this procedure

```
//MM01CL       JOB    (36512),'R MENENDEZ',NOTIFY=MM01
//JOBPROC      JCLLIB ORDER=SYS1.IGY.SIGYPROC
//STEP1        EXEC   PROC=IGYWCL,PARM.COBOL='APOST,OBJECT'
//COBOL.SYSIN  DD     DSN=MM01.PAYROLL.SOURCE(PAY4000),DISP=SHR
//COBOL.SYSLIB DD     DSN=MM01.PAYROLL.COPYLIB,DISP=SHR
//*——————————————————————————————————*
//LKED.SYSLMOD DD     DSN=MM01.PAYROLL.LOADLIB(PAY4000),DISP=SHR
//LKED.SYSLIB  DD
//             DD     DSN=MM01.PAYROLL.OBJLIB,DISP=SHR
```

The compile–link–and–go procedure for OS/390 COBOL

If you understand the first two procedures, the compile–link–and–go procedure in figure 17–9 should be easy for you because the COBOL and LKED steps are similar. As a result, you can focus on the GO step.

The GO step starts with an EXEC statement that executes a program identified by a backward reference to the SYSLMOD data set in the LKED step. But that's the load module that was created by the LKED step. This means that the GO step runs the load module for your source program.

Why is a backward reference needed? So this procedure works even if you override the SYSLMOD DD statement when you run the procedure. In fact, that's what the JCL example in this figure does. As a result, the backward reference actually identifies the load module named PAY4000 in the load library named MM01.PAYROLL.LOADLIB.

In the COND parameter for the EXEC statement, you can see that the program won't be run if 8 is less than the return code of the COBOL step, which means that the source program had serious errors. The program also won't be run if 4 is less than the return code for the LKED step, which usually means that the linkage editor couldn't link edit the user subprograms with the main program.

In the JCL for running this procedure, you can see that the DD statements for the COBOL and LKED steps are the same as they were for the last procedure. In the DD statements for the GO step, though, you need to allocate all of the data sets used by the program.

In this example, the DD statements allocate data sets for the PAYTRAN and PAYRPT ddnames that are defined by the program. The SYSOUT DD statement also allocates a SYSOUT data set, which will receive the output from any Display statements that are used in the program. In the GO step of the procedure, you can see a SYSUDUMP DD statement, which means that a storage dump will be printed if the program has an abnormal termination.

Figure 17–9: How to use the compile–link–and–go procedure for OS/390 COBOL

The IBM compile–link–and–go procedure for OS/390 COBOL

```
//IGYWCLG PROC LNGPRFX='SYS1.IGY',SYSLBLK=3200,
//             LIBPRFX='SYS1.CEE',GOPGM=GO
//*  COMPILE, LINK EDIT AND RUN A COBOL PROGRAM
//*  CALLER MUST SUPPLY //COBOL.SYSIN DD …
//*
//COBOL   EXEC PGM=IGYCRCTL,REGION=2048K
//STEPLIB  DD  DSNAME=&LNGPRFX..SIGYCOMP,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=&SYSLBLK)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//LKED    EXEC PGM=HEWL,COND=(8,LT,COBOL),REGION=1024K
//SYSLIB   DD  DSNAME=&LIBPRFX..SCEELKED,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&&GOSET(&GOPGM),SPACE=(TRK,(10,10,1)),
//             UNIT=SYSDA,DISP=(MOD,PASS)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
//GO      EXEC PGM=*.LKED.SYSLMOD,COND=((8,LT,COBOL),(4,LT,LKED)),
//             REGION=2048K
//STEPLIB  DD  DSNAME=&LIBPRFX..SCEERUN,
//             DISP=SHR
//SYSPRINT DD  SYSOUT=*
//CEEDUMP  DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

The JCL to invoke this procedure

```
//MM01CL        JOB     (36512),'R MENENDEZ',NOTIFY=MM01
//JOBPROC       JCLLIB ORDER=SYS1.IGY.SIGYPROC
//STEP1         EXEC    PROC=IGYWCLG,PARM.COBOL='APOST,OBJECT'
//COBOL.SYSIN   DD      DSN=MM01.PAYROLL.SOURCE(PAY4000),DISP=SHR
//COBOL.SYSLIB  DD      DSN=MM01.PAYROLL.COPYLIB,DISP=SHR
//*———————————————————————————————————*
//LKED.SYSLMOD  DD      DSN=MM01.PAYROLL.LOADLIB(PAY4000),DISP=SHR
//LKED.SYSLIB   DD
//              DD      DSN=MM01.PAYROLL.OBJLIB,DISP=SHR
//*———————————————————————————————————*
//GO.SYSOUT     DD      SYSOUT=A
//GO.PAYTRAN    DD      DSN=MM01.PAYROLL.TRANS,DISP=SHR
//GO.PAYRPT     DD      SYSOUT=A
```

The translate–compile–and–link procedure for CICS/COBOL

If you're a CICS/COBOL programmer, you may be interested in the procedure in figure 17–10. It's the translate–compile–and–link procedure for developing CICS load modules. As you can see, this procedure is named DFHYITVL, and it consists of three steps: TRN for the CICS translation, COB for the COBOL compilation, and LKED for the link–editing step. Note, however, that some of the lines have been omitted in this figure as indicated by two vertical dots so the procedure would fit on one page.

The JCL example after this procedure shows how you can use it. In the EXEC statement that invokes this procedure, you need to code a PROGLIB parameter that specifies the load library in which the load module that's created by the link−edit step will be stored. This overrides the value given by the PROC statement in the procedure, which is substituted for a symbolic parameter in the LKED step (not shown in this figure).

If you want to code a PARM parameter that sets options for the TRN and COB steps, you need to make sure that you don't override the options that are set by the procedure. In this case, the COBOL3 option is set in the TRN step and eight options are set in the COB step. As a result, you need to include these options in your PARM parameter if you want to set other options.

For the TRN step of your JCL, you need to code a SYSIN DD statement that identifies the source program. And for the COB step, if your program uses copy members, you need to code a SYSLIB DD statement that identifies the libraries that contain the copy members used by the program. In this case, though, you can't just override the SYSLIB statement in the procedure because its concatenated libraries contain members required by CICS. Instead, you have to code a DD statement with no parameters as a placeholder for the first library in the procedure. Then, you code a second DD statement to concatenate the copy library that you need, overriding the second library in the procedure (if you substitute the symbolic variable values, you'll find that the first and second DD statements in the procedure identify the same library, so it's safe to override the second one).

If your program uses static subprograms, you also need to concatenate any subprogram libraries that your program needs with the ones that are allocated in the procedure. You can do that by coding a SYSLIB DD statement for the LKED step as shown in the JCL example. Here, two DD statements with no parameters are required because the procedure allocates two object libraries, and you shouldn't override either one. In most cases, though, you can omit the LKED.SYSLIB statements because most CICS programs don't use static subprograms.

Finally, in the LKED step, you need to code a SYSIN DD statement that gives a name to the load module that is going to be stored in the load library that you identified with the PROGLIB parameter of the EXEC statement. The SYSIN data set is used to provide additional information to the linkage editor. In this example, it tells the linkage editor that the load module should be named CUSTMNT1. The R in parentheses after the module name tells the linkage editor to replace the module if it already exists.

Figure 17−10: How to use the translate−compile−and−link procedure for CICS/COBOL

The IBM translate−compile−and−link procedure for CICS/COBOL

```
//DFHYITVL PROC SUFFIX=1$,            Suffix for translator module
//       INDEX='CICSTS13.CICS', Qualifier(s) for CICS libraries
//       PROGLIB='CICSTS13.CICS.SDFHLOAD', Name of o/p library
//       DSCTLIB='CICSTS13.CICS.SDFHCOB',  Private macro/dsect
//       AD370HLQ='SYS1',             Qualifier(s) for AD/Cycle compiler
//       LE370HLQ='SYS1',             Qualifier(s) for LE/370 libraries
//       OUTC=A,                      Class for print output
//       REG=4M,                      Region size for all steps
//       LNKPARM='LIST,XREF',         Link edit parameters
//       STUB='DFHEILID',             Lked INC. fr DFHELII
//       LIB='SDFHC370',              Library
//       WORK=SYSDA                   Unit for work datasets
//TRN    EXEC PGM=DFHECP&SUFFIX,PARM='COBOL3',REGION=&REG
//STEPLIB  DD DSN=&INDEX..SDFHLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=&OUTC
//SYSPUNCH DD DSN=&&SYSCIN,DISP=(,PASS),
//            UNIT=&WORK,DCB=BLKSIZE=400,
//            SPACE=(400,(400,100))
//COB    EXEC PGM=IGYCRCTL,REGION=&REG,
//       PARM='NODYNAM,LIB,OBJECT,RENT,RES,APOST,MAP,XREF'
//STEPLIB  DD DSN=&AD370HLQ..SIGYCOMP,DISP=SHR
//SYSLIB   DD DSN=&DSCTLIB,DISP=SHR
//         DD DSN=&INDEX..SDFHCOB,DISP=SHR
//         DD DSN=&INDEX..SDFHMAC,DISP=SHR
//         DD DSN=&INDEX..SDFHSAMP,DISP=SHR
```

```
//SYSPRINT DD SYSOUT=&OUTC
//SYSIN    DD DSN=&&SYSCIN,DISP=(OLD,DELETE)
//SYSLIN   DD DSN=&&LOADSET,DISP=(MOD,PASS),
//            UNIT=&WORK,SPACE=(80,(250,100))
.
.
//LKED   EXEC PGM=IEWL,REGION=&REG,
//            PARM='&LNKPARM',COND=(5,LT,COB)
//SYSLIB   DD DSN=&INDEX..SDFHLOAD,DISP=SHR
//         DD DSN=&LE370HLQ..SCEELKED,DISP=SHR
.
.
//SYSLIN   DD DSN=&&COPYLINK,DISP=(OLD,DELETE)
//         DD DSN=&&LOADSET,DISP=(OLD,DELETE)
//         DD DDNAME=SYSIN
```

The JCL to invoke this procedure

```
//MM01CMPL    JOB  (36512),'R.MENENDEZ',NOTIFY=MM01
//CICSCMP     EXEC DFHYITVL,
//           PROGLIB='MM01.CICS.LOADLIB'
//TRN.SYSIN   DD   DSN=MM01.CICS.SOURCE(CUSTMNT1),DISP=SHR
//COB.SYSLIB  DD
//            DD   DSN=MM01.CICS.COPYLIB,DISP=SHR
//LKED.SYSLIB DD
//            DD
//            DD   DSN=MM01.CICS.OBJLIB.DISP=SHR
//LKED.SYSIN  DD   *
   NAME CUSTMNT1(R)
/*
```

In practice, most shops have JCL jobs that you can use for running the translate–compile–and–link procedure. Then, you just substitute the names for the load library, the source program, the copy library, the subprogram library, and the load module whenever you run the procedure.

The procedure for assembling a BMS mapset

If you're a CICS/COBOL programmer and you create your own mapsets, you may be interested in the procedure in figure 17–11. It creates a physical map and a symbolic map from a BMS mapset that is written in assembler language. Here again, most shops will have the JCL that you need for running this procedure already set up. As a result, you just have to replace the names in the EXEC and DD statements with names that are appropriate for the mapset you want to assemble.

If you look at the procedure, you can see that it is named DFHMAPS and it consists of four job steps. The COPY step runs the IEBGENER utility that's presented in chapter 18 to create a temporary data set from the source code for the mapset. In this step, you can see a comment that says you need to provide a SYSUT1 DD statement to identify the source code for the mapset.

The temporary data set then becomes input to the ASMMAP step that runs the assembler to create an object module for the physical map. It is followed by a LINKMAP step that runs the linkage editor to create the load module for the physical map. Last, the ASMDSECT step also uses the temporary data set created by the COPY step to create the symbolic map, or DSECT.

In the PROC statement for this procedure, you can see that the MAPLIB parameter gives the name of a library for the physical map, and the DSCTLIB parameter gives the name of a library for the DSECT. So if you want to store the maps in other libraries, you need to code these parameters in the EXEC statement for running the procedure. Similarly, the MAPNAME parameter in the PROC statement doesn't give a value. Its comment says that you must supply the mapset name in this parameter when you run the procedure.

The JCL example after the procedure shows you how to run it. In the EXEC statement for invoking the procedure, you code the MAPLIB parameter to identify the library for the physical map, the DSCTLIB parameter to identify the library for the symbolic map, and the MAPNAME parameter to supply the name that's used for those maps. Then, you code a SYSUT1 DD statement for the COPY step to identify the source member for the mapset. As you can see, the name of the source member and the name given in the MAPNAME parameter should be the same.

Figure 17−11: How to use the procedure for assembling a BMS mapset

An OS/390 procedure for preparing a BMS mapset

```
//DFHMAPS PROC INDEX='CICSTS13.CICS', FOR SDFHMAC
//             MAPLIB='CICSTS13.CICS.SDFHLOAD', TARGET FOR MAP
//             DSCTLIB='CICSTS13.CICS.SDFHMAC', TARGET FOR DSECT
//             MAPNAME=,                        NAME OF MAPSET - REQUIRED
//             A=,                              A=A FOR ALIGNED MAP
//             RMODE=24,                        24/ANY
//             ASMBLR=ASMA90,                   ASSEMBLER PROGRAM NAME
//             REG=2048K,                       REGION FOR ASSEMBLY
//             OUTC=A,                          PRINT SYSOUT CLASS
//             WORK=SYSDA                       WORK FILE UNIT
//COPY     EXEC PGM=IEBGENER
//SYSPRINT DD SYSOUT=&OUTC
//SYSUT2   DD DSN=&&TEMPM,UNIT=&WORK,DISP=(,PASS),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=400),
//            SPACE=(400,(50,50))
//SYSIN    DD DUMMY
//* SYSUT1 DD * NEEDED FOR THE MAP SOURCE
//ASMMAP   EXEC PGM=&ASMBLR,REGION=&REG,
//* NOLOAD CHANGED TO NOOBJECT
//  PARM='SYSPARM(&A.MAP),DECK,NOOBJECT'
//SYSPRINT DD SYSOUT=&OUTC
//SYSLIB   DD DSN=&INDEX..SDFHMAC,DISP=SHR
//         DD DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1   DD UNIT=&WORK,SPACE=(CYL,(5,5))
//SYSUT2   DD UNIT=&WORK,SPACE=(CYL,(5,5))
//SYSUT3   DD UNIT=&WORK,SPACE=(CYL,(5,5))
//SYSPUNCH DD DSN=&&MAP,DISP=(,PASS),UNIT=&WORK,
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=400),
//            SPACE=(400,(50,50))
//SYSIN    DD DSN=&&TEMPM,DISP=(OLD,PASS)
//LINKMAP  EXEC PGM=IEWL,PARM='LIST,LET,XREF,RMODE(&RMODE)'
//SYSPRINT DD SYSOUT=&OUTC
//SYSLMOD  DD DSN=&MAPLIB(&MAPNAME),DISP=SHR
//SYSUT1   DD UNIT=&WORK,SPACE=(1024,(20,20))
//SYSLIN   DD DSN=&&MAP,DISP=(OLD,DELETE)
//* NOLOAD CHANGED TO NOOBJECT
//ASMDSECT EXEC PGM=&ASMBLR,REGION=&REG,
//  PARM='SYSPARM(&A.DSECT),DECK,NOOBJECT'
//SYSPRINT DD SYSOUT=&OUTC
//SYSLIB   DD DSN=&INDEX..SDFHMAC,DISP=SHR
//         DD DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1   DD UNIT=&WORK,SPACE=(CYL,(5,5))
//SYSUT2   DD UNIT=&WORK,SPACE=(CYL,(5,5))
//SYSUT3   DD UNIT=&WORK,SPACE=(CYL,(5,5))
//SYSPUNCH DD DSN=&DSCTLIB(&MAPNAME),DISP=OLD
//SYSIN    DD DSN=&&TEMPM,DISP=(OLD,DELETE)
```

The JCL to invoke this procedure

```
//MM01MAPS JOB  (36512),'R.MENENDEZ',NOTIFY=MM01
//MAPASM   EXEC DFHMAPS,
//             MAPLIB='MM01.CICS.LOADLIB',    TARGET LOADLIB FOR MAP
//             DSCTLIB='MM01.CICS.COPYLIB',   TARGET COPYLIB FOR DSECT
//             MAPNAME=ORDSET1                NAME OF MAPSET (REQUIRED)
//COPY.SYSUT1 DD DSN=MM01.CICS.SOURCE(ORDSET1),DISP=SHR   MAPSET SOURCE
/*
```

Perspective

If this chapter has succeeded, you should now be able to use the IBM procedures for compiling and testing COBOL and CICS/COBOL programs. However, you should also be able to apply the skills that you've learned to any procedure for any language. To do that, you need to use the appropriate IBM manuals to find out what the step names, data set requirements, and options are. You may also want to print the code for the procedure that you're going to use. Once you've done that, though, the principles are the same.

After you set up the JCL for running a procedure, you can use it whenever you need to run the procedure again. Each time you use it, you just substitute the library and member names that the procedure requires. The trick, of course, is setting up the JCL the first time, which you should now be able to do.

Terms

compiler
source program
object module
object library
copy member
copy library
compiler listing
source statement library
linkage editor
link editor
load module
assembler
assemble
compile–link–and–go procedure
loader
compile–load–and–go procedure
compile–and–go procedure
CICS translator
compile–only procedure
compile–and–link procedure
static subprogram