9

USING FILES

By definition, CL is a control language. Because control languages rarely need to process files, processing files is usually left to more advanced programming languages. However, CL can process files in a limited way, and it can read and write to display files. Therefore, you can present prompt screens from a CL procedure and have user interaction with complete function key support.

While CL procedures also can read database files, writing database files is not supported. Files can be processed in two different ways. First, they can be processed record by record, the same way RPG programs can. Second, they can be processed as a whole—like other objects.

RECORD-BY-RECORD PROCESSING OF A FILE

CL procedures can process database and display files. Before a file can be processed record by record, however, it must be declared to the CL procedure using the Declare File (DCLF) command. A CL procedure can have up to five DCLF commands.

The DCLF Command

The DCLF command has three parameters. The first parameter, FILE, identifies the file being declared. You must enter the name of the file (which can be qualified with a library name). The second parameter, RCDFMT, lists the record formats that will be known to the CL procedure. It defaults to *ALL to give you a shortcut. If you are going to use a file (such as a display file) that has many record formats, but you will use only one or two in your procedure, it is to your advantage to list only those record formats you will use in the procedure in the RCDFMT parameter. Your program object will become smaller when compiled, and your compile listing will be less cluttered. The third parameter, OPNID, is optional, and defaults to *NONE, if only one file is declared. This value is used to distinguish one file from another. The OPNID must be referenced in all operations that pertain to the file, such as RCVF, SNDRCVF, SNDF, ENDRCV, and WAIT.

If your file was created from DDS, the CL procedure will know all the fields contained in the file because the CL compiler grabs the external-file definition. The CL procedure can use any of the fields as CL variables. For a file whose OPNID is *NONE (the default value), the variable name is the field name prefixed with an ampersand. For example, if a file has a field named TYPE, you can expect to find variable &TYPE in your procedure. This definition of variables is automatic, and you can use &TYPE as you would any other variable. If the OPNID parameter has some other value, each variable name consists of the leading ampersand, the OPNID value (without trailing blanks), an underscore character, and the field name. For example, if a file is declared with OPNID(F1) and has two fields named FILE and LIB, the CL compiler would automatically declare two variables named &F1_FILE and &F1_LIB.

If the file you are reading was created without DDS, the CL procedure has no way of knowing what fields you are using in the file. Therefore, the compiler creates only one variable, that has the same name as the file, and declares it as type (*CHAR). The variable's length is the same as the length of the record. For example, suppose you created FILE2 by running the command shown in Figure 9.1.

```
CRTPF mylib/file2 RCDLEN(256)
```

Figure 9.1: An example of creating a file with the CRTPF command.

When you declare the file in a CL procedure with the DCLF command, remember that you must use variable &FILE2, which will be a 256-byte character string. Use the %SST function to extract the parts you need.

The RCVF, WAIT, and ENDRCV Commands

The Receive File (RCVF) command is what CL procedures use to read the file declared with the DCLF command. RCVF always reads one record. If you process several RCVF commands, the file will be read sequentially. If the file member contains no more records, RCVF issues the escape message CPF0864 (end of file was detected).

Four parameters are defined to RCVF, and all of them are optional. The first, DEV, allows you to specify the name of a display device from which data is read. By using a variable in this parameter, you can retrieve data from different devices by changing the value of the variable. The default value for the DEV parameter is *FILE, which indicates that data is to be received from the display file declared with DCLF.

The second parameter, RCDFMT, is necessary when there are two or more record formats in a file. You may use the default value, *FILE, when there is only one record format in the file.

The OPNID parameter specifies which declared file is to be accessed. This value must match the OPNID parameter of the DCLF command.

The last parameter, WAIT, applies only to display files. WAIT accepts two values—*YES (the default) and *NO. WAIT(*YES) causes the system to suspend execution of the CL procedure until the input operation is complete. WAIT(*NO) causes the system to continue to execute CL commands until it encounters a corresponding WAIT command.

The WAIT command accepts two, both optional, parameters. The first, DEV, may contain a CL variable that contains the name of a display device from which input is expected. A value of *NONE means that no device need be specified.

The second parameter, OPNID, must match the OPNID for the DCLF command and the corresponding RCVF command in order to link WAIT with the appropriate RCVF command.

To cancel a wait state, use the End Receive (ENDRCV) command. Like WAIT, ENDRCV has DEV and OPNID parameters. However, the DEV parameter differs in that it requires a literal device name rather than a CL variable.

RCVF and Random Input

In combination with the Override with Database File (OVRDBF) command, RCVF can read a database file randomly by record number, by key, or in reverse sequential order. All you need to do is use OVRDBF for the file being read and indicate the desired record in the POSITION parameter. Then use RCVF to read the record. For instance, suppose you want to read record number 725 of FILE1. You could use the code as shown in Figure 9.2.

```
OVRDBF file1 POSITION(*RRN 725)
RCVF
```

Figure 9.2: Reading a specific record in a database file.

Suppose you need to read the record immediately before the record that has a key value of 'ABC' in a keyed file with three key fields. You would use code as shown in Figure 9.3.

```
OVRDBF file1 POSITION(*KEYB 1 FILE1R 'ABC')
RCVF
```

Figure 9.3: Reading the record prior to key 'ABC'.

*KEYB indicates that the next RCVF will read a record by key. The "B" means that a "key before" type of read is requested. The "1" indicates that you are using only one key field in the file. FILE1R is the name of the record format that has the key. "ABC" is the key value.

RPG programmers can think of OVRDBF as SETGT or SETLL, and RCVF as a READ or READP. In the preceding example, OVRDBF acts like SETLL and RCVF acts like READP.

The technique just presented (using OVRDBF and RCVF) to emulate random reads by key fails in one particular case. If the last key field of the database file is of type character and is not full (this field's last byte is blank), RCVF cannot read the file randomly by key. For example, if the key to the file is 4 bytes long and there is record with a key "ABC " (ABC plus a blank), OVRDBF and RCVF won't be able to find this record using *KEY. Use *KEYBE and repeat RCVF until you find the record you want.

Note: Another technique is to use a character variable that is 1 byte longer than the key field, and place any non-blank character in the last position. Then you can issue an OVRDBF with *KEY (Figure 9.4) and it will work okay.

```
DCL &key_plus_1 *CHAR    5 /* Key plus one byte: The key length is 4 */

CHGVAR &key_plus_1 (&key *CAT 'X')
OVRDBF file1 POSITION(*KEY 0 *N &key_plus_1)
RCVF
```

Figure 9.4: Positioning a file read using a key that is 1 byte longer than the file key.

Note: Variable &KEY is the file key (4 bytes long). You can build variable &KEY_PLUS_1 using the key value plus the letter X so the last byte is not blank. Then use OVRDBF just like before.

Notice the use of 0 and ***n*** in the position parameter. These are shortcuts. If you want to use the full key (all key fields), use zero instead of the actual number of fields. And if there is only one record format, you can use ***n*** instead of its name.

The SNDF and SNDRCVF Commands

The Send File (SNDF) command cannot be used on database files because CL procedures are not allowed to write into them. You can use the SNDF command to write a display file record, which means an output-only operation to present a panel to the interactive user.

If you follow the SNDF command with the RCVF command, you are actually displaying a panel and waiting for user input. You also can use the Send Receive File (SNDRCVF) command, which combines the two operations in a single command. In RPG terms, this command would be an EXFMT operation.

If the panel uses any indicators, they become &INXX logical variables to your CL procedure. For example, the panel may allow F3=Exit and F12=Cancel. You can use the code shown in Figure 9.5 to terminate the procedure.

```
DCLF dspfile

SNDRCVF RCDFMT(options)

IF (&in03 *OR &in12) +
   RETURN
```

Figure 9.5: An example of allowing F3 or F12 to terminate your procedure.

Another way to code the procedure is shown in Figure 9.6.

```
DCLF dspfile

SNDRCVF RCDFMT(options)

IF (&in03 *EQ '1' *OR &in12 *EQ '1') +
   RETURN
```

Figure 9.6: An alternative method for coding F3 or F12 to terminate your procedure.

The *EQ '1' business is not required because both &IN03 and &IN12 are logical variables. You can use whichever method you like. (You also can test if an indicator is off with the *NOT logical operator. An example would be IF (*NOT &IN05) to check whether indicator 05 is off. Using the other coding method, you can compare &IN05 against '0'.)

Similarly, you can turn indicators on or off before or after you present the panel. For example, you can highlight fields that have conditioning indicators in the DDS of the display file for such display attributes as high intensity or underline. Remember that any indicators used in the DDS of the display file become known to the CL procedure as logical variables &IN01 to &IN99. The command shown in Figure 9.7 turns on indicator 21.

```
    CHGVAR &in21 '1'
```

Figure 9.7: Using CHGVAR command to turn on indicator 21.

The CLOSE Command

The CLOSE command closes a file that was defined with the DCLF command. Once the file is closed, it is available for processing again. The first RCVF after a close of that same file re-opens the file. CLOSE accepts one parameter—the open identifier. For an example, see Figure 9.8.

```
PGM

DCLF   FILE(TempList)  OPNID(Temp)
... omitted lines ...
DOWHILE '1'
   RCVF   OPNID(Temp)
   MONMSG   CPF0864  EXEC(LEAVE)   /* end of file */
... omitted lines ...
ENDDO
CLOSE   OPNID(Temp)
/* the file may be re-opened by issuing another RCVF */
```

Figure 9.8: As of V6R1, you may close a file manually.

PROCESSING A FILE AS A WHOLE

CL procedures also can process files as another type of object. In this case, the entire file is processed as a whole and you are not limited to one file per CL procedure. Furthermore, you don't use the DCLF command to declare the file.

Creating and Deleting Files

CL procedures can be used to create and delete files by coding the appropriate commands in the CL procedure. For example, you can create a physical file with the Create Physical File (CRTPF) command and later delete it with the Delete File (DLTF) command.

Remember that you don't need DDS to create a physical file if you are willing to give up the benefits of external file definition. Keep in mind that creating and deleting files is a somewhat resource-intensive process that should be kept to a minimum. It is more efficient to create the files once and leave them in the libraries.

Processing Database File Members

Database files can be processed at the member level with the following commands:

- Add Physical File Member (ADDPFM) and Add Logical File Member (ADDLFM).
- Change Physical File Member (CHGPFM) and Change Logical File Member (CHGLFM).
- Remove Member (RMVM).
- Clear Physical File Member (CLRPFM).
- Reorganize Physical File Member (RGZPFM).

For example, suppose you have created a work file (WORKFILE) for a job. This work file is not created and deleted each time the procedure runs. To improve performance, the work file is left in the library between job runs.

Because it exists all the time, the job must ensure that there is no data from previous runs left over. The easiest way to make sure is to run the CLRPFM command to erase all data that exists. If no data exists, CLRPFM will do nothing.

This technique works when the procedure is run by only one person at a time. An example would be a program that is always submitted to the same single-threaded job queue. Because the job queue is single-threaded, it runs only one job at once. But what if the programmer cannot guarantee that two or more people won't attempt to run the same program at the same time?

One solution is to use different *members* and assign each job a separate member. This member must be added (ADDPFM command) at the beginning of the job and removed (RMVM command) at the end. Each member must have a different name. To assign unique names, you can use the six-digit job number (as shown in Figure 9.8).

```
DCL &jobnbr     *CHAR    6
DCL &mbr        *CHAR   10

RTVJOBA NBR(&jobnbr)
CHGVAR &mbr ('JOB' *CAT &jobnbr)
ADDPFM FILE(...) MBR(&mbr)

/* Processing */

RMVM FILE(...) MBR(&mbr)
```

Figure 9.8: Assigning a unique name to a member using the six-digit job number.

This method for using temporary files is not foolproof; use it with care. The program is provided only as an example of using ADDPFM and RMVM. One of the problems with this method is that the job might be canceled before it has a chance to run RMVM, and that leaves unwanted data in the database file.

The OVRXXXF and DLTOVR Commands

Files can be overridden for a number of reasons. Database files, in particular, can be overridden for three important reasons:

- They are not in any library of the library list. Rather than change the library list, the programmer can issue an Override with Database File (OVRDBF) command (Figure 9.9), which indicates that file FILE1 must be obtained from MYLIB.

```
OVRDBF file1 TOFILE(mylib/file1)
```

Figure 9.9: Telling the procedure to obtain a file that is not in the library list.

- The HLL program to be called refers to the file by a different name. Again, the OVRDBF solves this problem. In Figure 9.10, the HLL program uses file SOURCE, but the CL programmer wants to process QRPGSRC in library QGPL.

```
OVRDBF source TOFILE(QGPL/QRGPSRC)
CALL ...
```

Figure 9.10: Telling a program to use a file of another name.

- The CL programmer wants the HLL program being called to process a particular member of the database file—not the first one.

```
OVRDBF source TOFILE(QGPL/QRGPSRC)
CALL ...
```

Figure 9.11: Telling a program to use member WORKMBR of database file FILE1.

In this case, the program being called will process member WORKMBR. Left to itself, the system would have processed the first member by default.

Printer and display files also can be overridden with the Override with Printer File (OVRPRTF) and Override with Display File (OVRDSPF) commands. For example, suppose you created file INVPRT to print invoices on form INVOICE, which has a form size of 25 lines by 80 columns. Now you need to print invoice images on regular stock paper. Run the OVRPRTF command in Figure 9.12 before you call the HLL program that prints them.

```
OVRPRTF invprt FORMTYPE(*STD) PAGESIZE(66 132) OVRFLW(60)
```

Figure 9.12: Using the OVRPRTF command to change form attributes for the job.

When you use any of the override commands (OVRXXXF), remember the following rules:

- The override applies only to the current and following levels in the call stack. If you CALL a program that performs an override, the override will no longer be effective when control returns to your program.
- The system will activate only one override at each call level. If you override a database file with the TOFILE parameter now, and a few statements later perform another override with the MBR parameter, the file will be overridden only in member. If you suspect the file has been overridden and want to start over, run the Delete Overrides (DLTOVR) command first to delete any overrides currently in effect before you issue new ones.

Note: You can always use the Display Overrides (DSPOVR) command to check out what overrides a file might have at any moment.

Sorting with OPNQRYF

The Open Query File (OPNQRYF) command probably is the best method to sort files for later use. With OPNQRYF, you can specify what file or files to use, perform calculations on the fields from the files used, and select and sort the records based on the original fields or the calculated results. Because a detailed discussion of OPNQRYF fills a book by itself, OPNQRYF is described briefly here. For additional information, see MC Press' *Open Query File Magic!* by Ted Holt.

OPNQRYF uses the external file definition; the record layout DDS is provided when the file is created. You can reference the fields by their names (instead of by their absolute beginning and ending positions). Doing so gives you flexibility because you don't have to worry about changing the parameters of OPNQRYF if your file changes at some future date. The steps to use OPNQRYF for one file are as follows:

- Override the database file with SHARE(*YES) using the Override with Database File (OVRDBF) command. This command makes the HLL program that reads the query file use a shared open data path (ODP) for the file in question, rather than creating a new ODP when it opens the file.
- Run OPNQRYF. OPNQRYF creates an open data path over the data file(s) being queried.
- Run your HLL program to make use of the sorted file. Your HLL program must reference the file by its original name and can be coded as keyed or sequential access (it makes no difference). Again, because you created an open data path, the HLL program will use that data path and not open another. The HLL uses the sorted file (not the original file).
- Close the file with the Close File (CLOF) command.
- Delete the override with the Delete Override (DLTOVR) command.

The following CL program illustrates the entire procedure. An inventory master file (INVMST) is defined in Figure 9.13.

```
....+....1....+....2....+....3....+....4....+....5....+....6....+....7
     A         R IMREC
     A           IMITEM      15A       COLHDG('Item Number')
     A           IMTYPE       1A       COLHDG('Item' 'Type')
     A           IMCLAS       2A       COLHDG('Item' 'Class')
     A           IMCOST       7P  2     COLHDG('Standard' 'Cost')
     A                                 EDTCDE(1)
     A           IMPRCE       9P  2     COLHDG('Sell' 'Price')
     A                                 EDTCDE(1)
     A           IMDESC      30A       COLHDG('Item Description')
     A           IMDRAW      15A      COLHDG('Engineering' 'Drawing')
     A           IMLVL        3P  0    COLHDG('Level' 'Number')
     A                                 EDTCDE(3)
     A           K IMITEM
```

Figure 9.13: Inventory master file (INVMST).

Now suppose you need to write an HLL program—to list the inventory master file by item number and class code—that includes only the records that have "1" in the item type field. You can code the CL program as shown in Figure 9.14.

```
OVRDBF invmst SHARE(*YES)

   OPNQRYF invmst QRYSLT('imtype *EQ ''1''') +
           KEYFLD((imclas) (imitem))
   CALL ...
   CLOF invmst
   DLTOVR invmst
```

Figure 9.14: Using OPNQRYF to pre-select the items to be included in a report.

The OPNQRYF command creates the shared data path. OPNQRYF sorts the file by item by class (which means that the records will be sorted by item number for each item class) and selects only those records that have an item type equal to "1". Your HLL program is executed and the report prints. The CLOF command closes the open data path and DLTOVR removes the override. Remember that your HLL program must reference the file as INVMST.

Note: When OPNQRYF is processing very large files and/or selecting more than 80 percent of the records in a file member, you can improve performance by allowing OPNQRYF to use a sort routine instead of access paths. To permit the use of a sort, specify ALWCPYDTA(*OPTIMIZE). This does not guarantee that OPNQRYF will use a sort routine. The query engine may choose to use an access path instead.

CAPTURING OUTPUT USING QTEMP

QTEMP is a temporary library that exists only between the beginning and the end of a job. The system creates a QTEMP library (a different QTEMP library) for each job when the job begins, and it deletes the job's QTEMP library when the job ends. Each job has a different QTEMP. If you create an object in your job's QTEMP, no other job in the system can use that object.

QTEMP is the ideal place to put all the objects you create that are needed for the duration of the job only. For example, some commands or programs could create work files that contain data extracted from several files, which will be used to print a report. The work file has no further use after printing the report. Work files are by no means the only example and *FILE is not the only type of object you can put in QTEMP.

Because QTEMP is deleted when the job ends (either normally or abnormally), all objects placed in QTEMP will be lost at that time. If you create many objects in QTEMP in your interactive session, the SIGNOFF command will take a while to sign you off and present the sign-on screen again. During this time, the system will be busy deleting the many objects in QTEMP and then QTEMP itself.

Using Permanent Work Files

Frequently, you will need to create work files for temporary use. For example, you might have to design a report that gathers information from several files and performs many selections before it prints the report. Resist the temptation to create a "flat" file without DDS to describe the fields. It might be faster, but

you lose the capability to use external file definitions in your procedures. Also, program maintenance becomes more difficult. Get into the habit of creating permanent work files in your production libraries. You should put work files in the same library where you create the programs that use them. For example, you can create the file MYLIB/WORKFILE after you write the DDS for the file.

Each time you need to use the permanent file, create a duplicate in QTEMP (as in Figure 9.17). The copy in QTEMP will automatically share the complete record definition you provided with the DDS. In order to use the file, all you need to do is override it so the QTEMP copy is used instead of the original.

```
PGM

   CRTDUPOBJ workfile mylib *file qtemp

   OVRDBF workfile TOFILE(qtemp/workfile)

   /* Rest of the CL procedure goes here.  All references to WORKFILE */
   /* automatically apply to the copy in QTEMP, not to the original.*/

ENDPGM
```

Figure 9.17: Example of using permanent work files.

To simplify the task of creating these copies in QTEMP, appendix A includes the Create Work File (CRTWRKF) command. Figure 9.18 shows an example of using CRTWRKF instead of CRTDUPOBJ.

```
PGM

   CRTWRKF MODEL(mylib/workfile)

   OVRDBF workfile TOFILE(qtemp/workfile)

  /* Rest of the CL procedure goes here.  All references to WORKFILE */
  /* automatically apply to the copy in QTEMP, not to the original.*/

ENDPGM
```

Figure 9.18: Example of using the CRTWRKF utility.

Because it resides in QTEMP, the system automatically deletes the file when the job ends.

Using Outfiles

i5/OS commands that begin with the verb Display (DSP) are meant to produce output to the display station or to the printer. For example, the Display Program (DSPPGM) command shows information about a particular program. Its OUTPUT parameter determines where the information is presented. If the user selects * (the default value for the OUTPUT parameter), output goes to the display (if the command runs interactively) or to the printer (if the command runs in batch). If the user selects *PRINT, the output goes to the printer.

Other commands have a third option named *OUTFILE. In this case, output is directed to a database file you specify. For example, consider the command shown in Figure 9.19.

```
DSPOBJD OBJ(qgpl/*ALL) OBJTYPE(*ALL) OUTPUT(*OUTFILE) +
        OUTFILE(qtemp/objects) OUTMBR(*FIRST *REPLACE)
```

Figure 9.19: Using an OUTFILE with DSPOBJD.

The system will run the Display Object Description (DSPOBJD) command but, instead of displaying or printing the information gathered, the information goes to the first member of file QTEMP/OBJECTS and replaces whatever records might have been there. If file QTEMP/OBJECTS doesn't exist, it is automatically created.

The command executed (DSPOBJD) contains help text for all parameters. If you press the Help key while the cursor is on the OUTFILE parameter, the help text will tell you the name of the QSYS file that is used as a "model" for the file created.

In the case of the DSPOBJD command, the model outfile is QSYS/QADSPOBJ. You can use the Display File Field Description (DSPFFD) command to get the layout of QADSPOBJ's record. The layout will show you what information you are getting in the outfile, and where it is within the record.

Outfiles can be processed in CL procedures. For example, suppose you have approximately 200 user profiles on your system. For administrative purposes, you must change every profile that belongs to group profile GRP_A so it belongs to group profile GRP_B. If you performed this task manually, it would be an extremely tiresome, time-consuming, and error-prone job. Instead, you can write a CL procedure (let's name it CHGGRPPRF) as shown in Figure 9.20.

```
PGM (&oldgrpprf &newgrpprf)

   DCL &newgrpprf  *CHAR   10
   DCL &oldgrpprf  *CHAR   10

   DCLF qadspupb

   DSPUSRPRF *ALL TYPE(*BASIC) OUTPUT(*OUTFILE) +
             OUTFILE(qtemp/qadspupb)
   OVRDBF qadspupb TOFILE(qtemp/qadspupb)

loop:
   RCVF
   MONMSG cpf0864 EXEC(RETURN)
   IF (&upgrpf *EQ &oldgrpprf) DO
      CHGUSRPRF USRPRF(&upuprf) GRPPRF(&newgrpprf)
   ENDDO

   GOTO loop

ENDPGM
```

Figure 9.20: A CL procedure to automatically change the group profile in every user profile.

Now, execute the program and supply the old and new group profiles. All user profiles that had the old user profile will automatically be changed to the new group profile as shown in Figure 9.21.

```
CALL chggrpprf ('GRP_A' 'GRP_B')
```

Figure 9.21: Calling the program shown in Figure 9.19 and supplying old and new group profile names.

If you encounter this sort of situation often, you might consider creating a command so that you don't have to remember what parameters to pass and in what order. The program works by running the Display User Profile (DSPUSRPRF) command to an outfile. Because the outfile has the same name as the QSYS model (QADSPUPB), OVRDBF runs to make sure that the program uses the file in QTEMP instead of the empty QSYS model file.

The RCVF command is processed and reads a record from QTEMP/QADSPUPB. If the end of file is detected, CPF0864 is issued (which is trapped by the MONMSG command, resulting in an end of the program). Otherwise, the program compares the group profile name in the user profile just read (field &UPGRPF from the outfile) against &OLDGRPPRF. If they are equal, it changes the user profile (&UPUPRF) to the new group profile. Variables &UPGRPF and &UPUPRF come from the external definition of the outfile.

*Capturing OUTPUT(*PRINT)*

For those commands that don't offer the *OUTFILE option in the OUTPUT parameter, you can still process the output in a CL procedure if you direct the output to *PRINT and then read the printed report in the CL procedure. Here's what you need to do:

- Override the printer file to HOLD(*YES). You need to know the name of the printer file that is used by the command whose output you want to process. By holding the file, you ensure that it is not printed.
- Execute the command that has the output you want to process. Specify OUTPUT(*PRINT).
- Copy the spooled output to a physical file. The physical file must already exist without an external file definition. Use the Copy Spooled File (CPYSPLF) command to copy the spooled output to the physical file.
- Now you can process the physical file one record at a time. Remember that the file will contain the report headings, column headings, and footers

of the normal printed output. These records will have to be recognized and ignored by the procedure.

- Finally, delete the spooled file with the Delete Spooled File (DLTSPLF) command and remove the override to the printer file with the Delete Override (DLTOVR) command.
- This process is automated by using the Convert Print to Physical File (CVTPRTF) command provided in appendix A.