Structured Programming in a Sperry FORTRAN Plex Environment

Sandra L. Fitch, Shell Oil Company
William R. Lee, Shell Oil Company

Background

Shell Oil Company's Information and Computer Services (I&CS) organization is involved in solving Shell's business and technical problems through the use of computer technology. I&CS has two locations - the Information Center in Houston, Texas and the Credit Card Center in Tulsa, Oklahoma. The computing and data processing needs for Shell, other than the credit card division, are served by the Information Center in Houston. The services performed there are basically grouped into two major categories - Commercial Processing and Technical Processing.

The primary function of Technical Processing, a department of I&CS, is to convert large quantities of raw seismic data into a form to be interpreted by geophysicists in the Exploration divisions. Technical Operations has several Sperry multi-processor 3100/80's and 90's, two 100's, and one Array Processor per machine. These mainframes are connected by a high bandwidth local area network (developed by Shell) which allows direct process to process communication across mainframes. We are presently using S2k Release 2.90 and 2.95c, and 3.1, Exec Level 395A and Level 9R1 ASCII Fortran (with local modifications). Our programming is primarily written in ASCII FORTRAN, with certain applications coded in MASM. We use PLEX imbedded only in FORTRAN programs.

Geophysical Processing Systems (GPS) is a group within Technical Processing that develops and maintains systems used by Technical Operations to ensure the efficient management of Geophysical jobs at the Information Center. Most of our process control systems are large online systems that run in a real-time environment. The systems are all multibanked, multi-tasked, and use one or more System 2000 data bases. These systems consist of suites of programs (both independent tasks and subroutines); the number of program units in one of these systems may reach as many as 30 to 40 modules. Most of these program units use PLEX to manipulate attached S2k data bases.

Introduction

In a modern data processing shop, the need to write well structured, readable code is a critical requirement for a number of reasons - clarity, readability, maintenance, and efficiency, to name a few. In the GPS group we had converted to ANSI '77 FORTRAN from FORTRAN V and were making every effort to produce new code and to upgrade existing code to use structured techniques.

Additionally in our shop, modular program development was being stressed as an important technique in order to achieve well structured code. We tried to develop our systems so that unrelated processes or all purpose routines could be isolated into a single module, preferably an external subroutine or task. The ability to develop, test, and maintain each module independent of another module in the system was an essential tool for project management, development, and testing.

We found, however, the requirement to produce structured code and to develop code in separate modules quite difficult when using the FORTRAN PLEX precompiler to produce FORTRAN source code. With some research and experimentation, we were able to devise a method to produce structured FORTRAN Flex source output. We also devised a technique to support modular program development using Flex program units.

This paper will address both topics - techniques used to achieve structured FORTRAN PLEX code and a method of pre-compiling FORTRAN PLEX programs with multiple program units that will later be compiled independently.

Techniques used to achieve Structured Programming in PLEX

In the GPS group we have converted to ASCII FORTRAN and have made major strides in producing structured FORTRAN. In attempting to establish good programming
standards we tried to adhere to certain rules and guidelines which include:

1. Begin non-comment lines in Column 8.

2. Make use of the BLOCK-IF and IF-THEN-ELSE constructs. Indent the IF-block and the ELSE-block three spaces when using these constructs.

3. Indent the body of a DO loop three spaces.

4. Use meaningful variable names.

5. Write readable code. This includes using proper spacing, indentation, and internal documentation.

These are just a few of the guidelines we follow in our group. None of the enumerated rules appeared to be too restrictive and after a few iterations, these practices became habits in our project development efforts. That was, however, before we began to write PLEX!

Consider an example of FORTRAN PLEX code before (EXAMPLE 1) and after (EXAMPLE 2) precompilation using the R2.95 precompiler. How well have our programming guidelines been followed in this PLEX example?

EXAMPLE 1 - Excerpt of FORTRAN PLEX before precompilation

```
S2KDU5=2
S2KDU1=9T
CALL SETPRM(rptflx, T S2KDUM, 1 S2KDUM, 1 S2KDUM, 1 S2KDUM, T S2KDUM, 1 S2KDUM, T S2KDUM, 1 S2KDUM, 1 S2KDUM)
S2KOU2= 1
S2KOU1= 60
CALL OPENPL(rptflx,S2KDUM)
IF ( DAMAGE .NE. 0 ) THEN
  S2KOU5=0
  S2KDU2= T
  S2KDUT= 97
  CALL CLOSEPL(rptflx,S2KOU)
  STOP
ENDIF
S2KDU5=1
S2KDU1=10
S2KDU2= 61
S2KDL= 1
CALL LOCATE(SPDP ,S2KDUM) IF ( RHODE .EQ. 0 .AND. DATASN .GT. 0 ) THEN
S2KDU1= 93
S2KDUM= 72
S2KDL= 41
CALL GETPLG(SPDP ,S2KDUM)
END OF EXAMPLE 1
```

Upon inspection of EXAMPLE 1, one immediately sees that this code fails to meet three of the five listed guidelines. In fact the statements that fail are the PLEX statements. The requirement to begin all PLEX statements in column 7 poses a major block to any attempt to do proper indentation or to follow our in-house standard of FORTRAN beginning in column 8 or greater.

Let's now examine the source output after the same code has been precompiled.

EXAMPLE 2 - Excerpt of FORTRAN PLEX after precompilation

```
S2KDU5=2
S2KDU1=9T
CALL SETPROM(rptflx, 1 S2KDUM, 1 S2KDUM, 1 S2KDUM, 1 S2KDUM, 1 S2KDUM, 1 S2KDUM, 1 S2KDUM, 1 S2KDUM)
S2KOU2= 1
S2KOU1= 60
CALL OPENPL(rptflx,S2KDUM)
IF ( DAMAGE .NE. 0 ) THEN
  S2KOU5=0
  S2KDU2= T
  S2KDUT= 97
  CALL CLOSEPL(rptflx,S2KOU)
  STOP
ENDIF
S2KDU5=1
S2KDU1=10
S2KDU2= 61
S2KDL= 1
CALL LOCATE(SPDP ,S2KDUM) IF ( RHODE .EQ. 0 .AND. DATASN .GT. 0 ) THEN
S2KDU1= 93
S2KDUM= 72
S2KDL= 41
CALL GETPLG(SPDP ,S2KDUM)
END OF EXAMPLE 2
```

Now take a look at the output of the precompilation in EXAMPLE 2. The first thought that comes to mind is, 'What happened to MY code?'. The source output of the precompiler that is now to be compiled with the FORTRAN compiler shows little resemblance to the original source input. An examination of EXAMPLE 2 shows it to be nearly impossible to determine what the original PLEX commands were and what this code is suppose to accomplish. So again, we find that this code fails to meet any of the guidelines we listed earlier. In addition to failing any semblance of structured code, this code also proves to be unreadable and extremely difficult to interpret.

We had to find some way to produce well structured, readable, maintainable FORTRAN PLEX output, so we decided to develop our own techniques of producing structured code despite the constraints of the precompiler.

The solution was actually rather simple and involved two steps. The first of these was obvious.

- Every PLEX command would be preceded by a comment denoted by the characters 'CPL' in columns 1-3 and would be an echo of the actual PLEX command. The 'CPL' command would, however, be properly indented.
The second step involved using two EDIT directives to the compiler and placing these directives in strategic places in relation to the PLEX commands. These EDIT commands are documented in the SPERRY FORTRAN 911 Programmer Reference. The two directives were:

- **STOP EDIT SOURCE** which causes the compiler to terminate both the source and the object code listing, if it has been initiated by a preceding EDIT statement or by a processor call option.

- **START EDIT SOURCE** which causes the compiler to initiate the source listing, if it has been suppressed by a preceding EDIT statement or a processor call option.

We found that by using the 'CPL' PLEX command in combination with the STOP EDIT SOURCE and START EDIT SOURCE directives, the generated listing looked very much like the structured code we wanted. An example of how to use these is seen below:

```
EXAMPLE 3  PLEX command surrounded by comment and EDIT directives

CPL get (2) volume next.
*pI get (3) volume next.
END OF EXAMPLE 3
```

Now let's rewrite the code in EXAMPLE 1 to use this technique and then precompile and compile it.

```
EXAMPLE 4  Except from a PLEX program using structured techniques

CPL start s2k(2).
*pI start s2k(2).
CPL open rptflx.
*pI open rptflx.
if ( damage .ne. o ) then
  CPL close rptflx.
  *PI close rptflx.
  stop
and
CPL locate(1) spd where spdd exists.
*pI locate(1) spd where spdd exists.
if ( rtnede .eq. 0 .and. datasn .gt. 0 ) then
  CPL order by (1) spd.
  *PI order by (1) spd.

END OF EXAMPLE 4
```

EXAMPLE 5 shows the source output of the FORTRAN compiler after the code in EXAMPLE 4 (which uses our 2 step technique) has first been precompiled and then compiled. The code is now indented properly, easy to read, understandable, and maintainable.

```
EXAMPLE 5  Except from PLEX using special directives after compilation

CPL PASSWD 'RPTTST'
CPL START S2K(2).
CPL OPEN RPTTST.
if ( damage .ne. 0 ) then
  CPL close RPTTST.
END IF
CPL LOCATE(1) SPD WHERE SPDD EXISTS.
if ( RTNCDE .EQ. 0 .AND. DATASN .GT. 0 ) then
  CPL ORDER BY (1) SPD.
  SAVDSN = DATASN
  KNTSPD = 0
  DO 300 I = 1, SAVDSN
  ORDER by (1) spd.
  *PI GET(1) SPD NEXT.
END OF EXAMPLE 5
```

After inspecting EXAMPLE 5, one finds that by using the 'CPL' commands to echo the PLEX command and the compiler EDIT directives, structured code is possible -- EVEN USING FORTRAN PLEX!

The analysts in our group were quite pleased with the result of this technique since it produced semi-structured code. Two drawbacks, however, were (1) additional keystrokes required for the CPL echo line and the edit directives, and (2) due to keystroke errors the CPL command might not match the actual PLEX command. So we decided to go one step further - we developed our own PRE-PRECOMPILER to do the extra work for us. Our pre-precompiler

- Requires only the CPL commands which can span lines and begin in any column.
• Expands the CPt command into the genuine Plex command.

• Surrounds the #PL command with the STOP EDIT SOURCE and START EDIT SOURCE directives.

The output of the pre-precompiler is then fed to the S2K FORTRAN precompiler, and then compiled, and mapped. The result is still the structured code we desire without the extra keystrokes!

Most of our analysts today use the pre-precompiler to produce their PLEX source code; others, however, choose to include the additional comments and directives themselves. In any case, the result is a readable, structured source listing of a FORTRAN Plex program.

PRE-COMPIILING PLEX PROGRAMS WITH MULTIPLE PROGRAM UNITS

We had devised a method of producing structured FORTRAN PLEX code, but a workable means of performing modular program development of PLEX programs still did not exist. The next step was to tackle this problem. A viable solution would provide:

• A simple means of making a global change to the logical view of the database (the schemas)

• A method of precompiling and compiling PLEX programs units independently

• A means of making non-PLEX modifications to a PLEX program without being required to precompile the entire program again

First we decided to keep the schemas in a separate element that could be @add'ed to the main program so that it would be expanded in the main module during precompilation. We also took this same schemas element and created a PROC of the precompiled schemas using the @PDP processor so that it could be 'included' in external subroutines and tasks. An example of the runstream used to create the 'Schemas PROC' and the use of the separate Schemas element and PROC is seen below.

EXAMPLE 7 FORTRAN processor call to save the expanded PLEX source code

```
@FTN, sdir symprd.schemas/exp, relprd.schemas
```

This method allows one to handle the schemas external to each PLEX module. Next we devised a means to save the 'expanded' PLEX source code output from the precompiler so that we could work with the 'expanded' modules when we needed to make changes to non-PLEX code or simply wanted to add temporary diagnostic check prints to code during development and testing.

• First we added a FORTRAN processor call as a comment to the original source PLEX element. This call would save the source output of the precompiler in a separate element which we decided to identify with a standard version name of 'EXP'. An example is seen below -

EXAMPLE 8 Separate schema element and PROC in a program w/ multiple mod

```
@add symprd.schemas
@ed.U 3.
d.1
@pdp. ifl symprd.schemas/exp
slhchemas proc
@d. d.3.
d.4.
```

• Pre-Compiling PLEX Programs with Multiple Program Units

We had devised a method of producing structured FORTRAN PLEX code, but a workable means of performing modular program development of PLEX programs still did not exist. The next step was to tackle this problem. A viable solution would provide:

• A simple means of making a global change to the logical view of the database (the schemas)

• A method of precompiling and compiling PLEX programs units independently

• A means of making non-PLEX modifications to a PLEX program without being required to precompile the entire program again

First we decided to keep the schemas in a separate element that could be @add'ed to the main program so that it would be expanded in the main module during precompilation. We also took this same schemas element and created a PROC of the precompiled schemas using the @PDP processor so that it could be 'included' in external subroutines and tasks. An example of the runstream used to create the 'Schemas PROC' and the use of the separate Schema element and PROC is seen below.

394
**EXAMPLE 8 Runstream used to build a PLEX program**

```plaintext
@use s2k, s2k195
@xqt,q s2k.precornfor
@add symprd.mainprd
@add symprd.subsrt
@add symprd.subrte
@add symprd.subrt2
@tladd symprd.subrt3
end procedure.
@ed,@u 3.
```

This build element will cause all of the separate elements to be pre-compiled with one execution of the precompiler, a requirement of System 2000 PLEX programs. It then will modify the precompiler's output to cause each of the program modules to be compiled independently. The MAP for this program system, "bldprd.mainprg/map", must be prepared by using the standard "USR-FTN-MAP" modified to INCLUDE the elements that were created in this BUILD process.

The building of a PLEX program using this method will result in the expanded PLEX source code being saved in its own element. Modifications can then be made to any of the expanded modules independently. One then needs only to re-compile the module in question and re-map the program. We found this method extremely helpful for several reasons.

- Our large PLEX systems sometimes take as long as 20 - 30 minutes to precompile and compile. If only a few minor non-PLEX changes needed to be made to the code, those changes were made to the "expanded" code, recompiled, and remapped in a fraction of the time required to rebuild the entire system. Since our systems are online and vital to the operational environment at Shell, the ability to change or modify these systems quickly is essential.

- One helpful tool in debugging is the printing of diagnostic messages throughout the code in question. Adding these temporary check prints to the 'expanded' code proved to be another important reason to build our PLEX programs in the manner described.

- In the source listing, the subroutine calls that were the output of the PLEX precompiler were being suppressed by the EDIT directives (STOP EDIT SOURCE and START EDIT SOURCE). By saving the output of the precompiler in the 'EXP' element, we still had a listing of the expanded PLEX commands should we need them for diagnostics purposes.

**NOTE**

- A word of warning concerning modifications and additions to the expanded PLEX code - All permanent changes that are made to the expanded code should also be made to the original.

**Conclusion**

In this paper we have noted deficiencies in the SYSTEM 2000's FORTRAN PLEX precompiler's ability to support structured, modular program development. We have then outlined two methods that can be used to circumvent the problems with the precompiler. These techniques have 'eased the pain' somewhat, but are still only 'work-arounds'. The new S2K product will hopefully support the software engineering techniques that are so vital to the science of our field. Perhaps the methods described in this paper will aid in your FORTRAN PLEX software development and maintenance until an improved product is available.