An Automated FORTRAN Documenter

T. Erickson
Tracking Systems and Applications Section

We have written a set of programs designed to help R&D programmers document their FORTRAN programs more effectively. The central program reads FORTRAN source code and asks the programmer questions about things it has not heard of before. It inserts the answers to these questions as comments into the FORTRAN code. The comments, as well as extensive cross-reference information, are also written to an unformatted file. Other programs read this file to produce printed information or to act as an interactive document.

I. Introduction

Documentation is a continuing problem in software development: programmers don't like to produce it, much of it is never used, it takes up shelf space, time, and nervous energy. Yet we can't get along without it. We too easily forget how programs are put together and how to run them. After a week away from a program, undocumented variable names and sub-routine calls may seem like hieroglyphics even to their well-intentioned author.

One solution has been to insist that programmers adhere to strict documentation standards in spite of the time required and the paper produced. Every routine must be flowcharted, every variable explained. This has been applied to implement programming with some success, but it has never gained a foothold with programmers doing research and development (R&D).

Programs produced in an R&D environment still tend, as a rule, to be documented poorly. We cannot possibly calculate the time lost in program development due to forgetting, or the time lost when maintenance or implementation programmers can't figure out what the original author meant. Just why R&D programmers are so resistant to standardized programming practices is not clear. It may be their varied backgrounds: they can't be fit into the same molds. Maybe it's their academic histories: the programming techniques needed for thesis research seldom require documentation for other users or maintenance programmers. Perhaps it's natural stubbornness.
reacting to a "waste of time," or the eagerness to get on to the next problem "now that this routine is working." As to flow-charting routines before coding, the R&D programmer often does not know what algorithm is going to solve a problem best until the program runs.

Whatever the reasons for poor R&D documentation, something must be done. RNDOC (an R&D Documenter) and its family are interactive tools especially designed to help R&D programmers document their programs effectively and painlessly. It is designed first to help the programmer remember things that need remembering. Later versions will pay more attention to the needs of other programmers and the users of the programs.

II. How RNDOC Works

RNDOC is the name of the central program in this set of tools. It actually reads the FORTRAN code being documented; the other tools require output from RNDOC as well. It's being developed in FORTRAN 77 on a VAX 11/780 under the VMS operating system. It can be used to document programs written in FORTRAN and Structured FORTRAN. We will discuss its portability to other computers and other languages later.

RNDOC is basically a simple parser that decomposes lines of code into FORTRAN symbols and constants. We should briefly discuss the structures RNDOC can currently recognize in FORTRAN code. The first structure RNDOC recognizes is the whole program, whose name it gets from the user or from a PROGRAM statement. The program consists of one or more modules (functions, subroutines, and the main program), which in turn reside in one or more regions on the disk called files. Files may contain more than one module. For each module it encounters, RNDOC records the file it is in, so that it (and the user) will always know where to find the code. This is very simple, but even this saves time for a forgetful programmer. Within modules, RNDOC identifies arguments, common areas, I/O units, declared variables, and undeclared variables. For example, in the FORTRAN 77 code fragment

```
SUBROUTINE DISPLAY (lun)
character*6 symb
integer lunlim, x(10)
common/DISCOM/ lunlim, x
DO i = lun, lunlim
   write (i, 100) x, symb (x(10))
END DO
```

RNDOC will recognize the beginning of module DISPLAY, its argument lun, the common area DISCOM, the declared variables lunlim and x, which are also recognized as an array, the undeclared variables i and lun, the I/O unit lun, and the function symb. It will also record obvious relationships between these names: DISCOM is found in DISPLAY, symb is called by DISPLAY, lunlim and x reside in DISCOM, i is local to DISPLAY, and so forth. To carry the example further, if RNDOC has not yet heard of symb, it will ask the user for a comment. Now RNDOC knows that the program uses the module symb, though it will not know what file symb is in until it comes across

```
Character*6 FUNCTION Symb(j)
```

The user can set RNDOC to prompt for comments at various levels; RNDOC always requires comments for modules and arguments, but the user can choose whether to comment every variable. Furthermore, the user can "table" or postpone comments, or get a display of the section of code where a symbol was found.

At the end of a module, when RNDOC encounters a FORTRAN "END" statement, the user may quit, leaving RNDOC, or continue to other modules in the file. At the end of the file, the user can choose to write a new copy of the code with all new comments in place. Figure 1 shows schematically how RNDOC works.

III. An Example

Imagine that we have written a FORTRAN program to calculate primes. It looks like this:

```
PROGRAM Prime
   implicit integer (a-z)
   dimension primes(1000)
   call getnum(n, 'How many primes do you want to calculate?')
   primes(1) = 2
   DO i=2,n
      call FindNextPrime(i, primes)
   END DO

In file PRIME.FOR:
```

```
write(6,100) n,primes(n)
format('prime number ',i5,' is ',i6)
call getnum(m,'How many primes do you want to type out?')
DO i=1,m
    write(6,100)i,primes(i)
END DO
call exit
end

SUBROUTINE getnum(n,string)
character(*) string
write (6,100) string
format('*','a',x)
read (5,'(i10)') n
return
end

In file PRIMESUB.FOR:

SUBROUTINE FindNextPrime(i,Primes)
implicit integer (a-z)
dimension primes(i)
logical*1 IsTestPrime
test = primes(i-1)
DO WHILE (.true.)
    test = test + 1
    IF (IsTestPrime(test,primes,i)) THEN
        primes(i) = test
        return
    ENDIF
END DO
END

LOGICAL*1 FUNCTION IsTestPrime(test,primes,i)
implicit integer (a-z)
dimension primes(i)
DO j=1,i-1
    IF (mod(test,primes(j)) .eq. 0) THEN
        IsTestPrime = .false.
        return
    ELSE IF (primes(j)**2 .gt. test) THEN
        IsTestPrime = .true.
        return
    ENDIF
END DO
IsTestPrime = .true.
return
END

This program has been deliberately broken into subroutines for the purposes of this example. Let's see how parts of an RNDOC session would look at the terminal. In the following example, boldface text is typed by the user. Annotations appear in italics in the right hand column. On the VAX, control-Z is the end-of-file character, and will be indicated by [^Z].

$ run rncdc
enter name of program: PRIME
do you want to skip old comments? yes
Do you want to document variables other than arguments?
no

CREATING A NEW DOCUMENTATION FILE: prime.rnd
-----------------------------------------------
---------------- CURRENT STATUS ----------------
---------------- UNFOUND MODULES: --------------
none
-----------------------------------------------

Enter filename (ctrl-z to quit, SYS for system): prime.for

FOUND a reference to your main program, PRIME:

PROGRAM PRIME

Type in your comment (ctrl-z ends):

This is a program to calculate the first "n" primes, where "n" is a parameter typed in by the user. [^Z].

OKAY. Thanks.

--------------- ENTERING MODULE PRIME ------------

FOUND a reference to the routine GETNUM:
call getnum (n, 'How many primes do you want to calculate?')

Type in your comment (ctrl-z ends):

This routine gets an integer from the keyboard, prompting the user with the string argument. [^Z]

OKAY. Thanks.

END encountered in module PRIME.

Do you want to quit looking at PRIME, FOR? no

--------------- ENTERING MODULE GETNUM ------------

Reconciling variables in module GETNUM:
We need a comment for variable "n" in module GETNUM.

Type in your comment (ctrl-z ends):

OUTPUT: the number returned as data to the calling program. [^Z]

OKAY. Thanks.

We need a comment for variable "string" in module GETNUM.

Type in your comment (ctrl-z ends):

INPUT: the string used to prompt the terminal for the number. [^Z]
OKAY. Thanks.

END encountered in module GENTEST.

Do you want to quit looking for PRIME.FOR? no

END OF FILE found in PRIME.FOR.

Do you want to write a new copy of it? Yes

----------------------------------- CURRENT STATUS -----------------------------------
----------------------------------- UNFOUND MODULES:  -----------------------------------
  FINDNEXTPRIME
  EXIT
  2 modules in all

enter filename (ctrl-z to quit, SYS for system):

primesub.for

----------------------------------- ENTERING MODULE FINDNEXTPRIME -----------------------------------

FOUND a reference to some FUNCTION named INTTESTPRIME:

    IF (isTestPrime(test,primes,i)) THEN

Type in your comment (ctrl-z ends):


END OF FILE found in PRIMESUB.FOR.

Do you want to write a new copy of it? Yes

----------------------------------- CURRENT STATUS -----------------------------------
----------------------------------- UNFOUND MODULES:  -----------------------------------
  EXIT
  MOD
  2 modules in all

enter filename (ctrl-z to quit, SYS for system):

SYS

Is EXIT a system routine? yes
Is MOD a system routine? yes

----------------------------------- CURRENT STATUS -----------------------------------
----------------------------------- UNFOUND MODULES:  -----------------------------------
  none

enter filename (ctrl-z to quit, SYS for system): [^Z]

and we are done.
Here is what the rewritten code looks like:

```
PROGRAM Prime
C.
C. MODULE PRIME
C. This is a program to calculate the first “n” primes, where
C. “n” is a parameter typed in by the user.
C.
C. ARGUMENTS
C. --none--
C.
C. MODULES CALLED
C. EXIT This routine closes all files and stops execution.
C.
C. FINDNEXTPRIME This routine updates a list of the first (i-1) primes by
C. putting the i-th prime in position i.
C.
C. GETNUM This routine gets an integer from the keyboard,
C. prompting the user with the string argument.
C.
C. COMMON BLOCKS
C. --none--
C.
C.  processed by RNDOC 20-OCT-81 10:28:21 for program PRIME
C. implicit integer (a-z)
C. dimension primes(1000)
C. call getnum(n,’How many primes do you want to calculate?’)
C. primes(1) = 2
C. DO i=2,n
C. call FindNextPrime(i,primes)
C. END DO
C. write(6,100) n,primes(n)
C. format(‘prime number,’i5,’ is ‘,i6)
C. call getnum(m,’How many primes do you want to type out?’)
C. DO i=1,m
C. write(6,100)i,primes(i)
C. END DO
C. call exit
C. end
C.
C. SUBROUTINE getnum(n,string)
C.
C. MODULE GETNUM
C. This routine gets an integer from the keyboard, prompting
C. the user with the string argument.
C.
C. ARGUMENTS
C. --none--
C.
C. string input:
C. the number returned as data to the
C. calling program.
C.
C. OUTPUT
C. the string used to prompt the terminal
C. for the number.
C.
C. ENTRY POINTS
C. --none--
C.
C. COMMON BLOCKS
C. --none--
C.
C. processed by RNDOC 20-OCT-81 10:28:21 for program PRIME
C. character(*) string
C. write (6,100) string
C. format(‘#’,a,x)
C. read (5,’(i10)’) n
C. return
C. end
```

53
The other file, PRIMESUB.FOR, has been similarly commented. The unformatted documentation file, PRIME.RND, is now also on disk, and contains all the comments, module locations, and cross-reference information.

IV. Other Tools

We are designing and writing other tools to work with RNDOC to make R&D documentation easier. Generally, they read RNDOC's unformatted output and allow for code display if it is needed. They are interactive, and their displays are designed for a CRT. For the most part, these tools act as interactive documents without producing printout. Their functions include the following:

1. Module list: for each module in a program, lists the name of the file where it resides, its comment, the modules it calls, the modules that call it, its arguments, and the common areas it hosts.

2. Call tree: graphically displays which modules call which for all the modules in a program.

3. Comment editor: lets you change a particular comment without rerunning RNDOC.

4. Variable cross-reference: displays all lines of code in a program where a particular variable occurs.

5. Index searches: lets you look for a string or a word in all the comments and symbol names of a program in order to find some forgotten bit of information.

Here is an example of the use of such tools: suppose you have seen an output error in a large program. Instead of looking at the output routine to rediscover the name of the variable that is at least a symptom of the problem, you remember that it has something to do with "delay." You do an index search on "delay," and discover five variables whose comments contain that word. You probably recognize which one is the one you want from the names and comments. Having settled on ZDLY as the variable you are after, a variable cross-reference search will show all the lines in all modules of the program where ZDLY appears. You can use a general-purpose editor to make the appropriate changes to the code, and see if your solution worked. If it did you make changes to affected comments using the comment-editing tool.

V. Portability

Portability has also influenced the design of RNDOC. It is the main reason to read code and not listings — even though listings have a lot of useful information in them — as listing format changes from compiler to compiler.

RNDOC should be easy to run under any virtual memory operating system that supports FORTRAN 77. For smaller systems without virtual memory, the job will be harder but not impossible. Basically, parameters governing the size of arrays should be reduced, and the most space-consuming arrays changed to functions reading scratch direct-access files. RNDOC's response time will grow accordingly. As to host language requirements, RNDOC and its family rely heavily on the FORTRAN "character" data type, so versions of FORTRAN which do not support it will not support these tools.

Structured FORTRAN code can be processed by RNDOC, through each version of Structured FORTRAN must be tested thoroughly to ensure that RNDOC understands its particular methods of describing IF-structures and procedure calls. RNDOC's techniques could be used on other languages, such as Pascal; that would require major changes in the parser, and besides, excellent off-the-shelf Pascal Development Systems make such a program unnecessary.

VI. Current Work and Future Plans

The RNDOC project is currently working to improve the processing of I/O references, to improve the portability to systems smaller than the VAX, and to ensure upward compatibility of current comments and unformatted files with future versions of RNDOC.

Future versions of this software will include run-time flow information and recognition of structured blocks of code (DO-or IF-blocks). We are also continuing to try to find out just what documentation R&D programmers and their maintenance and implementation heirs need to program more effectively.
Fig. 1. Block diagram of RNDODC and expansion of one block (simplified)