A FORTRAN CODING CONVENTION FOR USE IN THE U.S. GEOLOGICAL SURVEY, WATER RESOURCES DIVISION

U.S. GEOLOGICAL SURVEY
Open-File Report 94-501

A Fortran Coding Convention for Use in the U.S. Geological Survey, Water Resources Division

By Kathleen M. Flynn, John L. Kittle, Jr., and Alan M. Lumb

U.S. GEOLOGICAL SURVEY
Open-File Report 94-501

Reston, Virginia
1994

U.S. DEPARTMENT OF THE INTERIOR

BRUCE BABBITT, Secretary

U.S. GEOLOGICAL SURVEY

Gordon P. Eaton, Director

For additional information write to:

Chief, Hydrologic Analysis Support Section
U.S. Geological Survey, WRD
415 National Center
Reston, VA 22092

Copies of this report can be purchased from:

U.S. Geological Survey
Earth Science Information Center
Open-File Reports Section
Box 25286, MS 517
Denver Federal Center
Denver, CO 80225

CONTENTS

A Fortran Coding Convention for Use in the U.S. Geological Survey, Water Resources Division

By Kathleen M. Flynn[1], John L. Kittle, Jr.[2], *and* Alan M. Lumb[1]

Abstract

A coding convention for computer programs written in Fortran has been established by the Water Resources Division of the U.S. Geological Survey. This convention covers both the implementation of selected Fortran features and a recommended coding style. It is designed to simplify the tasks associated with software support, maintenance, and distribution and is an important element in software quality assurance plans.

The SYStem DOCumentation (SYSDOC) program is also described in this report. The SYSDOC program can be used to produce detailed documentation for any program that follows this coding convention. SYSDOC produces text files that summarize PROGRAMs, SUBROUTINEs, FUNCTIONs, and COMMON blocks and documents all links between them.

INTRODUCTION AND BACKGROUND

As computing power expands and diversifies, the issues related to software development, maintenance, distribution, and support become more complex. Originally, the majority of the computing done by the U.S. Geological Survey, Water Resources Division (WRD), was done on mainframe computers, the first located in Washington, D.C., and the second at its headquarters in Reston, Va. In the early 1980's, much of the computing was moved to a network of minicomputers distributed around the country in district and region offices as well as the headquarters offices in Reston. Today, in addition to the mainframe and minicomputers, there is a large assortment of microcomputers and workstations. In the future, the numbers and assortments of microcomputers and workstations will increase. WRD has gone from maintaining and supporting a single copy of a program on the mainframe to distributing, maintaining, and supporting as many as 100 or more copies of a given program for an array of platforms.

With the expanding computing power, the libraries of programs have grown. Many of the programs are large and complex. Programming is often a team effort, with the programmers frequently located in different states. Often the people involved in the original programming are not available to help with support and maintenance.

A coding convention is particularly needed with a distributed programming, support, and maintenance staff, and a distributed computing system. With a well-defined coding convention, software is easier to read, understand, debug, maintain, distribute, and support than with no coding convention. Software documentation can be computer generated from code that conforms to a fixed convention. Computer-generated documentation can save a substantial amount of time and will be more accurate and up to date than handwritten documentation.

The convention described in this document is intended to simplify the tasks of documenting and supporting software written in Fortran. It should also be helpful in porting software to different types of machines. Unlike some of the newer programming languages, Fortran does not force structured programming (Berns, 1984). This convention encourages structured programming. Some Fortran features that may

[1] U.S. Geological Survey
[2] Consultant

be obsolete or are identified as poor programming practice are discouraged. This convention has not been found to limit the capabilities of Fortran.

Programs that have been written to conform to this convention can be documented using the SYStem DOCumentation (SYSDOC) program. Use of the program is documented in Appendix A. An example of the program documentation generated by SYSDOC is found in Appendix A.

This convention, in combination with SYSDOC and a Fortran static analyzer, provides a sound basis for software quality control. A static analyzer is a software tool that is used to identify problems and errors in code that are often overlooked by a compiler. Static analyzers include, but are not limited to, the Maintainability Analysis Tool (MAT) (Berns, 1985), FTNCHEK (Moniot, 1993), FORCHECK (Leiden University of the Netherlands), FOR-STUDY (Cobalt Blue, Inc., 1993), FOR-STRUCT (Cobalt Blue, Inc., 1992), and plusFORT (Polyhedron Software Limited, 1986-94).

This coding convention has evolved over a number of years beginning with a convention developed for the U.S. Environmental Protection Agency's Hydrological Simulation Program—FORTRAN (Johanson and Kittle, 1983). Several groups, including other Federal agencies, have been using versions of the convention. The final version described here represents a compromise of styles. The main goal has been to produce code that is consistent, well documented without being excessively verbose, readable, and easy to maintain and support. Experience has shown that implementing this convention takes little or no additional time during software development and can save a lot of time in software maintenance, debugging, support, and porting.

In the discussions that follow, the Fortran use and coding style described in this document will be referred to as the *convention* and the Fortran standards will be referred to as the *standard*. The Fortran standard that is associated with this convention is the American National Standards Institute (ANSI) X3.9-1978 FORTRAN standard. Extensions to the standard have been highlighted in the text by a gray background.

FORTRAN REQUIREMENTS, RESTRICTIONS, AND EXTENSIONS

This section describes requirements for and restrictions placed on the use of the ANSI X3.9 - 1978 FORTRAN standard. It also describes recommended extensions to the standard. In addition to the requirements and restrictions and excluding the extensions described in this report, all coding should comply with the standard.

All programmers should have a Fortran language reference manual. The reference manual should be written for the compiler being used and should clearly identify extensions to the standard. Another important tool for the programmer is a structured programming textbook. See the references for some possibilities.

The following section describes the Fortran constructs that are part of this convention. These include recommended constructs, as well as identifying those that should be avoided. In general, coding constructs that should be avoided are those that promote unstructured programming techniques, are not part of the standard, or are obsolete or little used constructs that may yield different results on different computer platforms. Any software features that contain device dependent code, such as operating system dependent input/output operations, should be isolated in separate routines. Table 1 contains a list of Fortran constructs that should be avoided and recommended alternatives to their use. Table 2 contains a list of recommended extensions to the standard. Note that some compilers may not accept all of the language extensions. These features, as well as recommended features, are described in more detail on the following pages.

Table 1. Fortran constructs to be avoided and suggested alternatives for these constructs

Fortran feature	Alternative
alternate RETURN	IF construct
arithmetic IF	IF construct
ASSIGN	structured programming techniques
assigned GO TO	structured programming techniques
BACKSPACE	internal write and read
blank COMMON	labeled COMMON
BLOCK DATA	subroutine that initializes common variables
computed GO TO	IF construct
DO noninteger control	use integer control variables
DIMENSION statement	explicit type declaration
ENTRY	multiple routines or option flags
EQUIVALENCE	
Hollerith	quoted characters
PAUSE	READ statement that waits for input
RETURN (multiple)	condition flags
STOP (multiple)	condition flags
tabs	spaces
, #, and / comments	comments beginning with C in column 1
\, &, !, <, >, ", and _	another character

Table 2. Fortran language extensions that are recommended

Fortran extension	Use
INCLUDE	PARAMETERs, COMMONs, and file names
lowercase	local variable names

Rules are not made to be broken. However, there are exceptions to rules. This coding convention has been developed over time and is based on experiences with porting to different hardware platforms and compilers, as well as experiences with program maintenance and support. The rationale behind each convention is explained below.

At the start of any programming project, these conventions should be reviewed. Any additional requirements, restrictions, or extensions should be clearly identified, described, and justified. Any modifications or amendments to this convention also should be identified, described, and justified.

Alternate RETURN

Alternate RETURNs should not be used.

An alternate RETURN introduces labels into an argument list to allow the called procedure to direct the execution of the caller upon return. The same effect can be achieved with a return code that is tested in an IF construct on RETURN. This avoids an irregularity in the syntax and semantics of argument association.

Arithmetic IF

The arithmetic IF should be avoided.

The multiple branching nature of this statement violates the principles of structured top-down programming and makes the code more difficult to understand and maintain. A preferred alternative is an IF statement or an IF construct.

ASSIGN and assigned GO TO

The ASSIGN statement and the assigned GO TO should not be used.

The multiple branching nature of these statements violates the principles of structured top-down programming and makes the code more difficult to understand and maintain. A preferred alternative is an IF statement or an IF construct.

The ASSIGN statement allows a label to be dynamically assigned to an integer variable and the assigned GO TO statement allows "indirect branching" through this variable. This hinders the readability of the program flow, especially if the integer variable also is used in arithmetic operations. The two totally different usages of the integer variable can be an obscure source of error. The statement should be replaced by ordinary assignment and the computed GO TO.

BACKSPACE

BACKSPACE statements should be avoided.

An alternative is to save the record into a character buffer and then use internal READs. Another option is to write the sequential file to a temporary direct access file, though this second option should be used sparingly due to portability issues (see the section Direct Access files).

BACKSPACE can make the logic of the code difficult to follow, especially when the READ and the BACKSPACE statements are in different parts of the code. Many inconsistencies in compilers and systems occur in file operations.

Blank COMMON

Blank COMMON is not used.

Blank COMMON should never be used for libraries because there is no name associated with it, and it is too easy for it to lose its identity.

BLOCK DATA

BLOCK DATA is not used.

The preferred alternative is to set the data values in a subroutine.

BLOCK DATA should never be used in a library structure as no entry point is generated; therefore, a loader has no way of bringing the initialized data into the executable. Because developed code will often eventually be made into libraries, this construct should not be used in general. The problem stems from the fact that the BLOCK DATA name is optional; therefore, when the entry-point symbol table is generated during compilation, its name is not carried through because it is not required.

Case

All comment lines are identified by an uppercase C in column 1 (a lowercase c is nonstandard). The text of the comment should be in mixed uppercase and lowercase to improve code readability. FORMAT statements may contain mixed uppercase and lowercase to meet output requirements and improve the readability of the output. All other statements are in uppercase.

Although the standard specifies that all code be in uppercase, most compilers allow either case. The use of mixed case may increase readability of the code. One case convention that may improve readability includes the following. All local variables are in lowercase. All Fortran language keywords are in uppercase. All COMMON block variables should have the first letter uppercase and the rest lowercase. All SUBROUTINE and FUNCTION, dummy argument, and PARAMETER variable names should be uppercase. There are many Fortran language processing tools that can be used to automate the case conversion process.

CASE

CASE constructs are not part of the standard and should not be used.

Use an IF statement or an IF construct.

Character set

The Fortran character set consists of the 26 uppercase letters A to Z, the ten digits 0 to 9, and the following special characters:

'	apostrophe	=	equal sign
*	asterisk	(	left parenthesis
	blank	-	minus sign
:	colon	+	plus sign
,	comma	)	right parenthesis
$	currency sign	/	slash
.	decimal point		

Lowercase letters are not part of the standard but may be used in defining variable names as described under Case.

The following characters are not part of the standard and should be used only in quoted character strings or comments:

&	ampersand	<	less than
\	backslash	"	quotation mark
!	exclamation point		tab
>	greater than	_	underscore

The backslash should be avoided as it is sometimes used by compilers as an escape character. Tabs should also be avoided due to problems porting software to various compilers and hardware platforms.

C language

When moving between Fortran and C, remember that the languages handle arrays in different ways. Fortran arrays are column major and C arrays are row major. (Fortran arrays are stored in memory with the leftmost dimension varying faster, C with the rightmost dimension varying faster.)

Comments

All comment lines are identified by an uppercase C in column 1. The text of the comment is in mixed uppercase and lowercase to improve code readability and is indented with the line of code immediately following the comment. Comment lines with no text may be used to separate groups of code but generally should not be used without an accompanying comment line containing text. If the logic of the code is such that the programmer believes it should be spaced by a blank comment, it probably needs a comment describing what the next block of code is all about.

For consistency, an uppercase C in column 1 is used to signify a comment. An * or # in column 1 or a /* on a statement line are not used. The *, #, and / are symbols that have specific meaning to some text editors and may cause problems when editing files containing these characters. The /* is not part of the Fortran standard.

Comment lines are never inserted into the middle of a statement that is continued on more than one line.

Comment lines are critical to making a program easy to understand. Use good comments liberally. Make sure that comments and code agree; when the code changes, comments also change.

Although the standard specifies that all code be in uppercase, most compilers allow mixed case. See Case for comments on using mixed case.

COMMON

All COMMON blocks are labeled. Blank COMMON is not used. Each variable is explicitly declared by type. The dimensions of arrays are placed in the explicit type declarations, not the COMMON block statement. Large arrays should be dimensioned with a PARAMETER if the code is written to allow different sizes for the arrays.

COMMON blocks should be in INCLUDE files to ensure that the order, size, and type of variables are consistent between routines. More than one COMMON block may be contained in an INCLUDE file.

The standard places restrictions on the order of some data types and on mixing numeric and character data in commons. CHARACTER variables are not contained in the same COMMON block as numeric or logical variables. DOUBLE PRECISION or COMPLEX variables are in separate COMMON blocks or are listed first if included with other numeric variables. For clarity, it is preferred that they be in separate COMMON blocks. Variables should be ordered from the largest variable type to the smallest. The following is the recommended order:

QUAD PRECISION
DOUBLE PRECISION
COMPLEX
REAL
INTEGER
LOGICAL
INTEGER*2
LOGICAL*1

A definition include file should follow the COMMON include file at least the first time the COMMON is found in the code. The definitions will follow the same format as argument definitions.

Comments telling which variables in the COMMON block are input, which are modified, and which are output should be placed in the routine following the COMMON block include file.

Computed GO TO

A computed GO TO may be used to implement the case structure but should generally be avoided.

The multiple branching nature of this statement makes it easy to violate the principles of structured top-down programming and may make the code more difficult to understand and maintain. A preferred alternative is an IF statement or an IF construct.

Concatenation

Use standard library operators and functions such as "//", INDEX, CHAR, and LEN for character concatenation. A library of subprograms is available from the authors for the manipulation of CHARACTER*1 arrays. These subprograms may be easier to use than concatenation of character variables.

Continuation lines

Continuation lines use a consistent symbol in column 6. Any of the 26 letters, 13 special characters, or the digits 1-9, as described above under character set, may be used. The standard does not allow the digit 0. Continuation characters for subprogram arguments are I, M, and O as described under dummy arguments below.

Code on a continuation line is indented at least as far as the code on the previous line. Lines of continued code are never interrupted by a comment. If a statement is so complex that it needs commenting in the middle, it will be difficult to understand and prone to error. It should be broken into smaller, more easily understood statements.

The standard limits the number of continuation lines to 19 per statement. Continuation is commonly found in type declarations, FORMAT, READ, WRITE, PRINT, DATA, CALL, FUNCTION, and SUBROUTINE statements. An attempt should be made to limit continuation in other executable statements to four or five lines. Equations spanning a number of lines may become very difficult to read.

The recommended character is the $. The alpha-numeric and arithmetic operators are not recommended for continuation characters as they can be confused with expressions and statement numbers, making the software harder to read and understand.

DATA statements

Make sure all program variables are initialized prior to use. DATA statements may be used to initialize SAVEd variables that are used as flags for initializing other variables. DATA statements also may be used to set the values for constant variables. Use executable statements to initialize all other variables.

Data types

The standard explicitly permits six types of data. Note that the only one of these that allows for a length, or size, specification is the CHARACTER type. The use of QUAD PRECISION, INTEGER*2, and LOGICAL*1 is discouraged. For consistency, it is recommended that the same order be used for making type declarations.

INTEGER
INTEGER*2
REAL
DOUBLE PRECISION
QUAD PRECISION
COMPLEX
LOGICAL
LOGICAL*1
CHARACTER*n

DIMENSION

The DIMENSION statement is not used.

All variables are declared explicitly by type. The dimensions of variables are included in the explicit type statements only. The dimensions of variables are not included in COMMON statements.

Direct Access files

Direct access files should be used carefully. File opens are separate from the rest of the code and well documented to facilitate maintenance and porting. A comment explicitly describes the required record length.

Inconsistencies in compilers occur in the units used to define the record length of unformatted direct access files. Depending on the compiler, or even the compile options selected, record length units may be bytes, words, half words, or some other unit.

DO loops

DO loops always end with a labeled CONTINUE. Multiple DO loops do not share the same CONTINUE. Control never jumps into a DO loop. Jumping out of a DO loop should be avoided. If control needs to jump out of a DO loop, then use a GO TO structure described below.

REAL and DOUBLE PRECISION DO control variables and DO control expressions are not used. Use INTEGER constants or variables.

DO WHILE, DO UNTIL, and DO END

DO WHILE, DO UNTIL, and DO END are not part of the standard and generally should not be used. Use a GO TO structure described below to implement these features.

Dummy arguments

SUBROUTINE dummy arguments are ordered and listed as Input, Modify, and Output variables, each type beginning on a new continuation line, in the stated order, with an I, M, or O, respectively, in column 6. All dummy arguments in a FUNCTION are Input because FUNCTIONs return a single value. See FUNCTION below for more information.

The practice of using I, M, and O for continuation lines, both in the routines and in the calling routines, has been extremely helpful in debugging, program maintenance, and sharing programs.

ENTRY

ENTRY points are not used. The alternatives include passing an option flag to the routine or separating the routine into multiple routines.

Multiple entries (and the usually accompanying multiple returns) violate the principles of structured top-down programming and make the code more difficult to understand and maintain. A subprogram should be entered at the beginning and exited at the end.

EQUIVALENCE

EQUIVALENCE statements should be avoided. The use of equivalenced variables often reduces program clarity, making maintenance more difficult.

FORMAT

FORMATs are grouped together in numerically ascending order, with the input FORMATs preceding the output FORMATs. FORMATs are consistently numbered.

FORMATs are grouped together and consistently located so they are easy to find and so the logic and structure of the code is easy to read. The ranges 1000 to 1999 or 8000 to 8999 are recommended for input FORMATs and the ranges 2000 to 2999 or 9000 to 9999 are recommended for output FORMATs.

Free format, Fortran

Free format of Fortran code is an extension of the standard that should not be used. The format that should be used for Fortran is:

column	content
1	comment
2-5	statement label
6	continuation
7-72	statement
73-80	blank or revision comment

Free format, input/output

Free format for input and output is recommended, as appropriate, for the application.

FUNCTION

All FUNCTION statements include an explicit type specification. FUNCTIONs return a single value. FUNCTION arguments are input only. For clarity and maintenance, FUNCTIONs do not modify or output dummy arguments and do not use COMMON blocks.

GO TO

GO TO in conjunction with an IF pointing back to a CONTINUE statement is used to implement a structured DO WHILE or DO UNTIL. GO TO statements should be avoided in all other cases.

An IF statement or an IF construct is used in place of a GO TO pointing down in the code.

The use of GO TOs are strictly controlled in structured programming. They should be used only to implement a structured construct, such as DO/WHILE, DO/UNTIL, or CASE, which are not available in the standard or when the elimination of the GO TO will obscure rather than clarify the meaning of the code.

Undisciplined use of the GO TO statement is, perhaps, the most common violation of structured programming principles.

See also the Computed GO TO section.

Grouping of routines

The logical grouping of a program, subroutines, and functions into files will be dependent on a number of things. The type of routine grouping should be decided on at the beginning of the project.

When routines are grouped for a library, they should be ordered by the calling sequence. How the compiler pulls routines into a program should also be considered.

Large new systems being developed by a number of people will require stringent version control and the ability to easily locate a particular routine. In this case, it may be most efficient to store each routine in a separate file.

For other programs, it may be more convenient to group closely related routines together in a single file.

IF constructs

Use IF constructs to implement branching.

IF with .EQ. and .NE.

In general, .EQ. and .NE. should not be used to compare floating point variables. An alternative is to check for a very small absolute difference between the two variables. Machine precision and round off in computations may make equivalent variables different by a very small fraction.

INCLUDE

INCLUDE files are used for PARAMETER statements and COMMON blocks. They may also be useful for blocks of code that are machine dependent, such as file handling. See the SPECIFICATIONS, DOCUMENTATION, AND STYLE section for outlines for some of these INCLUDE files.

INCLUDE is not part of the standard, but the use of INCLUDE files is available on most compilers. It is a fairly simple editing task to replace the INCLUDE statement with the appropriate code for compilers that do not allow INCLUDE. Use of INCLUDE files for COMMON blocks ensures that the order, size, and type of variables are consistent between routines. INCLUDE files can also simplify the task of porting code to multiple machine/compiler types. INCLUDE files also may be helpful for including statements during debugging.

Suggested naming conventions for INCLUDE files are:

p_____.inc - parameter
c_____.inc - common block
d_____.inc - variable definitions
f_____.inc - file name or file handling code
x_____.inc - system dependent block of code
b_____.inc - debugging block of code

______.cmn - common block
______.sys - system dependent block of code
______.dbg - debugging block of code

See also the section Name conventions (file).

Indentation

Indentation is used to denote blocks of code. It is used with DO loops, IF constructs, GO TO implementing DO WHILE and DO UNTIL structures, and error handling for OPEN, READ, WRITE, and error conditions. Code is generally indented two to four spaces—be consistent. The beginning and ending points of the block of code are not indented (DO, CONTINUE, GO TO, IF, ELSE IF, ELSE, and END IF). Comments are indented with the code.

Input

Make sure input does not violate the limits of the program (array dimensions, value range of data types). Terminate input by end-of-file or end-of-record, not by count. Perform validity checks on input and have recovery methods for invalid input. Use free-form input whenever appropriate. Have defaults for input data when appropriate. Input should be self-descriptive, using keywords to allow easy coding and proofreading. Test data should be for the extreme requirements of the code.

Intrinsic functions

Intrinsic functions should be used when possible.

Nonstandard functions should be avoided. These nonstandard functions include AND, OR, XOR, NOT, LS, RS, SHFT, LT, RT, LOC, RND, IRND, INTS, INTL, and double precision and complex functions.

Line numbers

The use of line numbers in columns 73-80 is unnecessary.

The original purpose of the line numbers was to aid in sorting a dropped deck of cards, which is no longer valid. Text editors can provide line numbers for editing purposes when they are needed.

See the Statement labels section for a discussion on numbering statements.

Name conventions (file)

File names may be anything that is valid on the platform being used. A consistent naming convention is important. Files that are used on more than one platform should have the same name on all platforms to reduce confusion.

Some projects may need to develop a detailed naming convention for file names to convey pertinent information to the user.

There are a few system-specific limitations on file names. Very few systems permit blanks within a file name. The PC environment generally limits a file name to eight characters followed by a period and a three-character suffix, with some suffixes having special meaning to the system. Special characters such as <, >, /, and \ should be avoided as they may have special meaning on some systems.

Name conventions (symbolic)

All program, function, subroutine, and variable names should be as descriptive as possible (mean something) within the limit of six characters. The standard limits the length of a name to six characters, and those characters are the letters A through Z and the digits 0 through 9. Function and variable names are explicitly defined by type. Although each name is explicitly defined, a first-letter, type naming convention of I through N for INTEGER, D for DOUBLE PRECISION, C or A for CHARACTER, and the remaining alphabet for REAL may enhance readability. Descriptive names should always take precedence over a type naming convention.

The standard defines a symbolic name as having from one to six letters or digits, the first being a letter.

COMMON block variable names should be at least two characters in length.

Symbolic names longer than six characters may be more descriptive. Some projects may need to develop a detailed naming convention to improve readability and to convey pertinent information to the reader. The first six characters should be unique as some compilers truncate names to six characters. Note that there are tradeoffs between using long, descriptive variable names and the readability of even relatively simple equations.

PARAMETER

PARAMETER statements should be used for important constants and symbolic data, such as pi, logical unit numbers, and array dimensions. These statements ensure that a constant is not inadvertently changed and enhance portability of the code as they allow modification of device specific information in a single statement. For example, if arrays are dimensioned using a PARAMETER value, then the code could be easily made smaller in terms of memory requirements for use on a small problem or when machine memory is limited.

PARAMETERs should be defined in INCLUDE files.

PAUSE

PAUSE should be avoided.

Execution of a PAUSE statement requires operator or system-specific intervention to resume execution. In most cases, the same functionality can be achieved as effectively and in a more portable way with the use of an appropriate READ statement that awaits some input data.

RETURN

There is a single RETURN in each subprogram. Multiple RETURNs are not used.

Multiple RETURNs cause nonstructured code, make documentation and code maintenance more difficult, and are not necessary. A subprogram should be entered at the beginning and exited at the end.

Statement labels (numbers)

Statement labels are used only on FORMAT and CONTINUE statements, never on executable statements. Statement labels start in column 2 and increase as you go down in the code. FORMAT statements are 1000 or greater. The numbers 1 to 999 are used for CONTINUE statements. Be consistent in numbering statements.

Statement numbers that are not in numerically ascending order make code difficult to understand and maintain. One-digit statement labels in column 1 may be difficult to find when reading code that includes comment lines.

One numbering method is to have CONTINUE statements begin at a base of 10 and be incremented by a value that is dependent on the current indentation level (that is the number of open DO or IF blocks). The increment value of un-indented code should be 100; therefore, the first un-indented label is 100 CONTINUE. The label increment for code at indentation level 2 should be 50, for level 3 - 20, for level 4 - 10, for level 5 and above - 1. For example, if the last label is 445 and the indentation level is 3, the label will be 460 (the next higher multiple of 20). Use of this labeling style may increase readability of the code. Fortran language processing tools can be used to automate this process.

STOP

There is a single STOP, located at the end of the main program. Well-structured code with IF constructs and flag variables does not need additional STOP statements.

Multiple STOPs cause nonstructured code, make documentation and code maintenance more difficult, and are not necessary.

Structure

Routines have readable flow from top to bottom. The code follows structured programming principles. PROGRAMs, SUBROUTINEs, and FUNCTIONs, and code blocks in general, are entered at the beginning (top) and exited at the end (bottom). ENTRY points are not used. There is a single STOP in a program, located at the end of the main routine, immediately before the END statement. SUBROUTINEs and FUNCTIONs never contain a STOP and contain a single RETURN, located immediately before the END statement.

See the section STRUCTURED PROGRAMMING TECHNIQUES.

Tabs

Tabs are not used.

Tabs are not part of the standard. They make program porting more difficult.

Uppercase/ lowercase

See Case.

STRUCTURED PROGRAMMING TECHNIQUES

All programmers should have a Fortran language reference manual. The reference manual should be written for the compiler being used and should clearly identify all extensions to the standard. Another important tool for the programmer is a structured programming textbook. Structured programming is based on two premises (General Electric, 1986):

1. that programs must be designed and written in a manner that is understandable and maintainable; and
2. that reliable software can be created by refining a problem (and its solution) into manageable elements.

This coding convention has been developed over time. It is based on the experiences of many scientists and programmers developing, maintaining, and supporting software for different hardware platforms using varied Fortran compilers. This convention, when followed, is designed to structure code in recognizable patterns with each block of code (control structure) having a single entry and a single exit, thereby supporting the functionality of top-down design. Structured programs are better than unstructured in three ways: increased reliability, easier verification, and easier modification.

Recognizable patterns in the code greatly simplify software development and maintenance as the meaning of the code is more easily ascertained and automatic tools can be used to aid in the development process. Therefore, both manual and automatic validation of software is enabled.

Validation of unstructured code is complicated as each statement must be treated as a separate event. That is, in order to understand the meaning of the code, a statement-by-statement examination must be made (the programmer must simulate a computer). Also, automated procedures cannot be used and debugging becomes very tedious and prone to errors.

Program readability is enhanced using this convention as program structure is developed in a predictable manner in recognizable blocks of code. Each block of code performs a unique operation with a single entry and a single exit. The size of a single routine should be limited to one or two printed pages for the program logic. Through this modularization of the code, individual subfunctions can be easily identified and understood.

This coding convention, in addition to defining recommended coding constructs, defines a coding style (indentation, commenting conventions, and internal documentation, and so forth) designed to provide maximum clarity and readability. For example, indentation enhances clarity by showing the logical structure of the code.

SPECIFICATIONS, DOCUMENTATION, AND STYLE

This section describes the parts of the convention related to specification statements, in-line documentation, and overall style and readability. Fortran code that is written using this style of specifications and documentation can be processed by the SYSDOC program described in Appendix A, producing documentation in the format shown in Appendix A. Character strings containing keyword identifiers are used by SYSDOC to identify the various elements of the specifications and documentation. Table 3 contains an ordered list of the character strings SYSDOC expects to find by default. When a specification is not needed for a routine, then the character string identifier should be omitted. The purpose, history, and end specifications identifiers are always required. An outline for a Fortran subroutine is shown in figure 1. Outlines for functions and programs are very similar to the subroutine outline. The remainder of this section contains brief descriptions and examples for each of the documentation elements.

Table 3. Character strings used to identify key elements in the documentation

Character string	Required
+ + + PURPOSE + + +	yes
+ + + HISTORY + + +	yes
+ + + KEYWORDS + + +	
+ + + DUMMY ARGUMENTS + + +	
+ + + ARGUMENT DEFINITIONS + + +	
+ + + PARAMETERS + + +	
+ + + PARAMETER DEFINITIONS + + +	
+ + + COMMON BLOCKS + + +	
+ + + COMMON DEFINITIONS + + +	
+ + + SAVES + + +	
+ + + SAVE DEFINITIONS + + +	
+ + + LOCAL VARIABLES + + +	
+ + + LOCAL DEFINITIONS + + +	
+ + + EQUIVALENCES + + +	
+ + + EQUIVALENCE DEFINITIONS + + +	
+ + + FUNCTIONS + + +	
+ + + INTRINSICS + + +	
+ + + EXTERNALS + + +	
+ + + DATA INITIALIZATIONS + + +	
+ + + INPUT FORMATS + + +	
+ + + OUTPUT FORMATS + + +	
+ + + STATEMENT FUNCTIONS + + +	
+ + + END SPECIFICATIONS + + +	yes

Figure 1. Fortran subroutine outline.

```
C
C     nnnnnn
C
      SUBROUTINE   aaaaaa
     I                   ( aaaaaa,...
     M                     bbbbbb,...
     O                     cccccc,...)
C
C     + + + PURPOSE + + +
C     This space is used to define the purpose and function of this
C     routine.  It should be mixed uppercase and lowercase because it
C     will be reproduced in the system documentation exactly as it
C     appears here.
C
C     + + + HISTORY + + +
C     Name 08/06/95 short description of change.
C
C     + + + KEYWORDS + + +
C     aaaaaam, bbbbbm...
C
C     + + + DUMMY ARGUMENTS + + +
      INTEGER
      REAL
      DOUBLE PRECISION
      CHARACTER*n
      LOGICAL
C
C     + + + ARGUMENT DEFINITIONS + + +
C     aaaaaa - definition of first subroutine argument, in mixed
C              uppercase and lowercase.  This definition will be
C              reproduced in the system documentation exactly as
C              it is entered here.
C        .
C     cccccc - definitions of last subroutine argument
C
C     + + + PARAMETERS + + +
      INCLUDE 'Pxxxxx.INC'
C
C     + + + PARAMETER DEFINITIONS + + +
C     pppppp - parameter definitions
C
C     + + + COMMON BLOCKS + + +
      INCLUDE 'Cxxxxx.INC'
C     I:  AAA, BBB, ...
C     M:  CCC, DDD, ...
C     O:  EEE, FFF, ...
C
C     + + + COMMON DEFINITIONS + + +
      INCLUDE 'Dxxxxx.INC'
C
C     + + + SAVES + + +
      INTEGER
      REAL
      DOUBLE PRECISION
      CHARACTER*n
      LOGICAL
      SAVE
C
C     + + + SAVE DEFINITIONS + + +
C     SSSSS1 - saved variable definitions are included here
C
C     + + + LOCAL VARIABLES + + +
      INTEGER
      REAL
      DOUBLE PRECISION
      CHARACTER*n
      LOGICAL
C
```

Figure 1. Fortran subroutine outline--Continued.

```
C     + + + LOCAL DEFINITIONS + + +
C     AAAAA1 - local variable definitions are included here
C
C     + + + EQUIVALENCES + + +
      INTEGER
      REAL
      DOUBLE PRECISION
      CHARACTER*n
      LOGICAL
      EQUIVALENCE ( ______, ______ )
C
C     + + + EQUIVALENCE DEFINITIONS + + +
C     AAAAA1 - equivalenced variable definitions are included here
C
C     + + + FUNCTIONS + + +
      INTEGER
      REAL
      DOUBLE PRECISION
      CHARACTER*n
      LOGICAL
C
C     + + + INTRINSICS + + +
      INTRINSIC
C
C     + + + EXTERNALS + + +
      EXTERNAL
C
C     + + + DATA INITIALIZATIONS + + +
      DATA  AAAA1, AAAA2, AAAA3
     $      /   1,     2,     3 /
C
C     + + + INPUT FORMATS + + +
 1nnn FORMAT (     )
C
C     + + + OUTPUT FORMATS + + +
 2nnn FORMAT (     )
C
C     + + + STATEMENT FUNCTIONS + + +
C     description of statement function
      NAME ( arguments ) = expression
C
C     + + + END SPECIFICATIONS + + +
C
C     Code goes here.  Should generally be less than
C     150 statements, exclusive of comment lines.
C
      RETURN
      END
```

PROGRAMS, FUNCTIONS, SUBROUNTINES

Three comment lines precede PROGRAM, FUNCTION, and SUBROUTINE statements. They may be blank comment lines or may include a programmer-defined identification system for subprograms. All function statements are preceded by a type specification.

```
C
C     Version 1.0
C
      PROGRAM   SYSDOC
```

and

```
C
C
C
      SUBROUTINE   VUSE
```

and

```
C
C     9410.1
C
      INTEGER   FUNCTION   CRINTE
```

ARGUMENTS

SUBROUTINE dummy arguments are listed as Input, Modify, and Output variables, each type beginning on a new continuation line, in the stated order with an I, M, or O, respectively, in column 6. This practice has been extremely helpful in debugging, program maintenance, and sharing subprograms.

FUNCTION arguments are input only. FUNCTIONs return a single value. For clarity and standard compliance, FUNCTIONs do not have Modify or Output arguments, nor common blocks. If a routine needs a Modify or Output argument or contains a common block, it should be written as a subroutine, not a function.

PROGRAMs have no arguments.

```
      SUBROUTINE   VUSE
     I                 ( CMNFG, VNAM, UPDCNT, UPDYTP, UPDNAM,
     I                   FCOUT,
     M                   BUFF
     O                   NUMUSE, IMOFG )
```

and

```
      INTEGER   FUNCTION   CRINTE
     I                           ( ERRINT, LEN, STR )
```

PURPOSE

A paragraph in mixed case (for readability) describes the purpose and function of the routine. This information should be complete, as it will be reproduced in the system documentation. Any nonstandard features should be described.

```
C
C     + + + PURPOSE + + +
C     This routine converts a character string to its integer
C     equivalent.  It returns the value of ERRINT for an
C     invalid string.  The integer is expected to be right
C     justified in the string.  The integer may be negative.
```

HISTORY

This space is used to document the history of the routine. The name of the author or responsible person or unit is included here, as well as dates and justifications for modifications, fixes, and other changes.

```
C
C     + + + HISTORY + + +
C     KMFlynn  09/30/90  tested and accepted
C     KMFlynn  02/10/92  change made in code to handle bug
C                        found in compiler for handling quoted
C                        backslash
```

KEYWORDS

Keywords are helpful in program maintenance and in indexing the subprograms. Keywords are separated by commas and contained in columns 7 through 72. Generally, the number of keywords will be fairly small. If there are a lot of keywords, the routine may be doing too many things and may be difficult to maintain.

```
C
C     + + + KEYWORDS + + +
C     Character conversion, Integer Conversion,
C     Numeric conversion
```

DUMMY ARGUMENTS

All arguments are explicitly declared by type in the order INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and CHARACTER. Any array dimensions are included in the explicit type declaration. There are no implicit type declarations or dimension statements.

The *n size declaration is not for numeric or logical variables; it is an extension to the standard that should be avoided.

```
C
C     + + + DUMMY ARGUMENTS + + +
      INTEGER     CMNFG, UPDCNT, UPDTYP(UPDCNT), FCOUT, NUMUSE
      CHARACTER*1 VNAM(8), BUFF(100), UPDNAM(6,UPDCNT), IMOFG
```

ARGUMENT DEFINITIONS

A definition is included for each dummy argument. The definitions appear in the same order that the arguments are passed to the routine. Each definition begins on a new line. The Argument name appears in uppercase on the first line of the definition beginning in column 7. A hyphen (-) is in column 14. The definition is in mixed case in columns 16-72, using as many lines as are needed to define the argument. The definition will be reproduced in the documentation exactly as it appears in the code.

A hyphen (-) is in column 14 or one space after the name, whichever is larger. The definition is in mixed case beginning one space after the hyphen, not exceeding column 72, and left justified to this point. Typically the definitions will be in columns 16-72.

```
C
C     + + + ARGUMENT DEFINITIONS + + +
C     DATE   - starting date and time
C              (1) year       (4) hour
C              (2) month      (5) minute
C              (3) day        (6) second
C     TSTEP  - time step, in TUNIT units
C     TUNIT  - time units
C              1 - second      4 - day
C              2 - minute      5 - month
C              3 - hour        6 - year
C     NVAL   - number of data values
C     VALUE  - array containing NVAL data values
```

PARAMETERS and PARAMETER DEFINITIONS

PARAMETER statements are used mainly to define the size and limits of arrays and the unit numbers for input and output. A PARAMETER constant is declared in an explicit type declaration before it is defined, and it is defined before it is used. The definition of a PARAMETER constant follows the PARAMETER statement, using the same form as was used for dummy arguments.

It is often useful to place PARAMETER statements in INCLUDE files to ensure that a parameter is used consistently between routines and to more easily modify the values when needed for adjusting array limits and for porting the code. An outline for a PARAMETER include file is shown in figure 2. See INCLUDE in the FORTRAN REQUIREMENTS, RESTRICTIONS, AND EXTENSIONS section for suggestions on file naming conventions. Definitions for PARAMETERs may be included in the PARAMETER INCLUDE file or in a separate definition INCLUDE file.

```
C
C     + + + PARAMETERS + + +
      INCLUDE 'PSORT.INC'
```

Where the file PSORT.INC contains:

```
      INTEGER   LENREC, LENARY
      PARAMETER ( LENREC=120, LENARY=1000 )
C
C     + + + PARAMETER DEFINITIONS + + +
C     LENREC - maximum record length that will be sorted
C     LENARY - maximum number of records that can be sorted
```

Figure 2. PARAMETER INCLUDE file outline.

```
      INTEGER   nnnnnn
      PARAMETER   (nnnnnn=      )
C
C     + + + PARAMETER DEFINITIONS + + +
C     nnnnnn - definition of parameter goes here
```

COMMON BLOCKS and COMMON DEFINITIONS

All COMMON blocks are named. There is no blank COMMON. Character and numeric data are not mixed in the same COMMON block. Each COMMON statement is followed by explicit type declarations for each member of that COMMON. Any arrays in the COMMON are dimensioned in the explicit type declarations.

COMMON blocks are placed in INCLUDE files. An INCLUDE file may contain more than one COMMON block. Definitions of the variables in COMMON may be placed in the COMMON include file or in a definition include file. These definitions are in the same format as argument definitions. See INCLUDE in the FORTRAN REQUIREMENTS, RESTRICTIONS, AND EXTENSIONS section for suggestions on file naming conventions. Outlines for these files are shown in figures 3 and 4. Each COMMON block is followed by comment lines indicating which variables from the INCLUDE file are Input to the routine, Modified by the routine, and Output by the routine.

```
C
C     + + + COMMON BLOCKS + + +
      INCLUDE 'CPLOT.INC'
C     I:  KOUNT, KOLOR, LINE
C     M:  LABX, LABY, TITLE
C     O:  XAXIS, YAXIS
```

Where file CPLOT.INC contains:

```
C     + + + PARAMETERS + + +
      INTEGER   MAX
      PARAMETER ( MAX = 100 )
C     + + + PARAMETER DEFINITION + + +
C     MAX - maximum number of points
C
      COMMON / PLOTN / KOUNT, KOLOR, LINE
                       XVAL, YVAL, XMIN, XMAX, YMIN, YMAX,
     $
      INTEGER   KOUNT, KOLOR, LINE
      REAL      XVAL(MAX), YVAL(MAX), XMIN, XMAX, YMIN, YMAX
C
      COMMON / PLOTC / LABX, LABY, TITLE
      CHARACTER*40 LABX, LABY
      CHARACTER*60 TITLE(3)
C
C     + + + COMMON DEFINITIONS + + +
C      KOUNT  - number of points to be plotted
C      KOLOR  - code for line color
C               1 - black
C               2 - red
C               3 - blue
C      LINE   - code for line type
C               1 - solid
C               2 - dash
C               3 - dot
C      XVAL   - array of values for x-axis
C      YVAL   - array of values for y-axis
C      XMIN   - minimum value for x-axis
C      XMAX   - maximum value for x-axis
C      YMIN   - minimum value for y-axis
C      YMAX   - maximum value for y-axis
C      LABX   - label for x-axis
C      LABY   - label for y-axis
C      TITLE  - title for plot, 3 lines
```

Figure 3. COMMON block INCLUDE file outline.

```
C     + + + PARAMETERS + + +
      INTEGER   nnnnnn
      PARAMETER   (nnnnnn=      )
C
C     + + + PARAMETER DEFINITIONS + + +
C     nnnnnn - definition of parameter goes here
C
C     Description of how this common is used
      COMMON / CCCCCC / iiiii1, rrrrr1
C
      INTEGER   iiiii1
      REAL      rrrrr1
C
C     Description of how this common is used
      COMMON / CCCCCC / dddddd
C
      DOUBLE PRECISION dddddd
C
C     Description of how this common is used
      COMMON / CCCCCC / ccccc1, ccccc2
C
      CHARACTER*N  ccccc1, ccccc2
```

Figure 4. Definition INCLUDE file outline.

```
C
C     + + + COMMON DEFINITIONS + + +
C     iiiii1 - definition of common member
C     rrrrr1 - definition of common member
C     ddddd1 - definition of common member
C     ccccc1 - definition of common member
```

SAVE and SAVE DEFINITIONS

The save statement is used to allow local variables within a subprogram to retain their value between calls to the subprogram. SAVE should not be indiscriminately used. Variables to be saved are explicitly declared by type before they are included in the SAVE statement. Define all saved variables as shown. Any variable that needs to be saved needs a good explanation of how it is used.

```
C
C     + + + SAVES + + +
      REAL      TABLE(20,5)
      LOGICAL   FIRST
      SAVE      TABLE, FIRST
C
C     + + + SAVE DEFINITIONS + + +
C     TABLE  - table used to compute discharge
C              (1) - depth, in feet
C              (2) - area, in acres
C              (3) - volume, in acre-feet
C              (4) - outflow 1, in cubic feet per second
C              (5) - outflow 2, in cubic feet per second
C     FIRST  - indicator for status of TABLE
C              TRUE - populate the TABLE array
C              FALSE - use current values in TABLE array
```

LOCAL VARIABLES

All local variables are explicitly declared as INTEGER, REAL, DOUBLE PRECISION, CHARACTER, or LOGICAL. Local variables should be defined to assist subsequent programmers in maintaining the code. The six characters allowed by the standard for variable names generally do not permit meaningful names; however, trivial variables, such as loop indexes, need not be defined. In addition to having precise definitions grouped at the beginning of the routine, it is helpful to have comments within the code explaining the function of the variables.

Local variables may be coded in lowercase. Because six characters may not be meaningful, variable names may be longer, but the first six characters must be unique.

```
C
C     + + + LOCAL VARIABLES + + +
      INTEGER   I, J, KOUNT, DATE(6), TSTEP, TUNIT
      REAL      DISCH
C
C     + + + LOCAL DEFINITIONS + + +
C     KOUNT  - number of data values
C     DATE   - array containing starting date and time
C              (1) - year (4 digits)       (4) - hour
C              (2) - month                 (5) - minute
C              (3) - day                   (6) - second
C     TSTEP  - time step, in TUNIT units
C     TUNIT  - time unit for time step TSTEP
C              1 - second         5 - day
C              2 - minute         6 - month
C              3 - hour           7 - year
C     DISCH  - array containing KOUNT discharge values
C              beginning at date/time DATE, with time step
               TSTEP,TUNIT
```

EQUIVALENCES

EQUIVALENCE statements should be avoided. They can make program maintenance very difficult. All variables in an equivalence statement are explicitly declared by type. (NOTE: This is an example of using EQUIVALENCE, it is not a recommended use.)

```
C
C     + + + LOCAL VARIABLES + + +
      CHARACTER*80  TITLE
      CHARACTER*40  NAME
C
C     + + + LOCAL DEFINITIONS + + +
C     TITLE  - title for printout
C     NAME   - name of site being analyzed
C
C     + + + EQUIVALENCES + + +
      CHARACTER*1  TEXT(120)
      EQUIVALENCE ( TEXT(1), TITLE ) (TEXT(81), NAME )
C
C     + + + EQUIVALENCE DEFINITIONS + + +
C     TEXT   - descriptive information printed out
C              (1-80) contains a title
C              (81-120) contains site name
```

FUNCTIONS

All external functions and statement functions used within a routine are explicitly declared by type as INTEGER, REAL, DOUBLE PRECISION, CHARACTER, or LOGICAL.

```
C
C     + + + FUNCTIONS + + +
      INTEGER     LENSTR
      REAL        CHRDEC
      CHARACTER*1 DIGCHR
      REAL   YLINE, XLINE
```

INTRINSICS

All intrinsic functions, such as MOD, ALOG, REAL, DBLE, and so forth, are declared.

```
C
C     + + + INTRINSICS + + +
      INTRINSIC   ABS, MOD
```

EXTERNALS

All external subprograms, including system subprograms and graphics subprograms, are declared.

It is recommended that continuation lines not be used to declare EXTERNALs. One compiler has been found that cannot handle the continuation. System utilities, such as grep in UNIX, can be used to easily identify all EXTERNAL routines used by a collection of code. In addition, it may also be helpful to group externals by library or functionality.

```
C
C     + + + EXTERNALS + + +
      EXTERNAL   LENSTR, CHRDEC, DIGCHR, CHRDEL, CTRSTR
      EXTERNAL   GSCHUP, GTX, GQTXX
```

DATA INITIALIZATIONS

Data initializations are grouped together. It is a violation of the standard to initialize data in a type declaration statement.

Definitions for all variables in DATA statements are included with the definitions for local variables. Use a format for the data statement that is easily read and uncluttered. Group associated data. When array elements have specific meaning, it may be useful to include a descriptive comment if it fits in 1 or 2 lines.

```
C
C     + + + DATA INITIALIZATIONS + + +
      DATA    SMALL,     LIMITS
     $      / 0.00001,  20,5,10 /
C                   id  name  lat  lng
      DATA  INDEX /   2,   45,   8,   9  /,
     $       TYPE /  'C',  'C'  'R', 'R' /,
     $      LENTH /  16,   48,   1,   1  /
```

INPUT FORMATS

Input FORMAT statements use statement labels greater than 999. The ranges 1000 to 1999 and 8000 to 8999 are recommended. Statement numbers are usually incremented by 10. They are grouped together in numerically ascending order.

```
C
C     + + + INPUT FORMATS + + +
 1000 FORMAT ( 5F10.3 )
 1010 FORMAT ( I10, 4F10.2 )
```

OUTPUT FORMATS

Output FORMAT statements use statement labels greater than 999 and greater than input FORMAT statements. The ranges 2000 to 2999 and 9000 to 9999 are recommended. Statement numbers are usually incremented by 10. They are grouped together in numerically ascending order and follow the INPUT formats.

Code FORMATs so they are easy to read. Do not use Hollerith data or the H field descriptor. It may be helpful to start a continuation line in the FORMAT statement for each new line in the output. Using indentation and lining up text that will be lined up in the output may make the FORMAT statement easier to read and modify.

```
C
C     + + + OUTPUT FORMATS + + +
 2000 FORMAT (/, 1X, 'Summary of measured data on ', I4,2I3,
     $        /, 1X, '    precipitation gage ', A7,  '=', F5.2,
     $        /, 1X, '                  gage ', A7,  '=', F5.2,
     $        /, 1X, '                  basin average =', F5.2,
     $        /, 1X, '        total runoff, in inches =', F5.2 )
 2010 FORMAT (//,1X, '***',
     $        /, 1X, '*** Warning:  total runoff exceeds basin ',
     $               'average precipitation by', F5.2, ' inches.',
     $        /, 1X, '***', / )
```

STATEMENT FUNCTIONS

Statement functions are grouped together, with each function preceded by a comment line(s) describing the purpose of the function. The explicit type declaration for statement functions occurs earlier with the declarations for all external functions. All arguments of the statement function are explicitly declared by type under local variables.

```
C
C     + + + STATEMENT FUNCTIONS + + +
C     compute Y given X and the slope and intercept
      YLINE ( SLOPE, X, INTRCP ) = SLOPE * X + INTRCP
C
C     compute X given Y and the slope and intercept
      XLINE ( SLOPE, Y, INTRCP ) = (Y - INTRCP) / SLOPE
```

END SPECIFICATIONS and LOGIC

The END SPECIFICATIONS line signals the end of the declaration, definition and specification statements, and the beginning of the logic of the code. The code should be well structured. It should generally be limited to two printed pages, including comments. Indentation is used for DO loops, GO TOs, and IF constructs. Two to four spaces are used for indentation. Comment lines should be used liberally and are indented with the code. See the FORTRAN REQUIREMENTS, RESTRICTIONS, and EXTENSIONS section. See also the section STRUCTURED PROGRAMMING TECHNIQUES.

```
C
C     + + + END SPECIFICATIONS + + +
C
      .
      .
      .
      (program logic)
      .
      .
      .
C
      RETURN
      END
```

SELECTED REFERENCES

American National Standard for Information Systems Programming Language Fortran, June, 1989, Draft S8, Version 112, S8(X3.9-198x), American National Standards Institute, Inc., New York, N.Y.

American National Standard Programming Language FORTRAN, 1978, ANSI X3.9-1978, American National Standards Institute, Inc., New York, N.Y.

American National Standard Programming Language Fortran 90, 1991, ANSI X3.198-199x, American National Standards Institute, Inc., New York, N.Y., 369 p.

Barnwell, T.O., Jr., and Kittle, J.L., Jr., 1984, Hydrological Simulation Program - FORTRAN, Development, Maintenance and Applications: *in* Proceedings, Third International Conference on Urban Storm Drainage, Chalmers Institute of Technology, Goteborg, Sweden.

Berns, G.M., 1984, New Life for Fortran: Datamation, September 1, p. 166-174.

_____ 1985, Maintainability Analysis Tool, MAT, User's Guide for MAT Version 10: Science Applications International Corporation, Arlington, Va., 46 p.

Burns, Evelyn, 1985, Fortran 77 Reference Guide, Fourth Edition (Updated for Revision 20.2 by Jerry Onrstein, 1986), Prime Computer, Inc., Natick, Mass.

Cobalt Blue, Inc., FOR-STRUCT Your Fortran Structuring Solution, 1992, Roswell, Ga., 147 p.

Cobalt Blue, Inc., FOR-STUDY FORTRAN /Static Analyzer, 1993, Roswell, Ga., 90 p.

Data General Customer Documentation: Green Hills Software Fortran Language Reference Manual, Data General Corporation, Westboro, Mass., 1992, 293 p.

Etter, D.M., 1987, Structured Fortran 77 for Engineers and Scientists, The Benjamin/Cumming Publishing Company, Inc., Menlo Park, Calif., 519 p.

General Electric, Corporate Information Systems, Bridgeport, Conn., 1986 "Software Engineering Handbook", McGraw-Hill Series in Software Engineering and Technology, 224 p.

Johanson, R.C., and Kittle, J.L., Jr., 1983, Design, Programming and Maintenance of HSPF: Journal of Technical Topics in Civil Engineering, v. 109, no. 1, April, p. 41-57.

Katzan, Harry, Jr., 1978, Fortran77: Van Nordstrand Reinhold Company, New York, 203 p.

Kernighan, B.W., and Plauger, P.J., 1976, Software Tools: Addison-Wesley Publishing Company, Reading, Mass., 338 p.

_____ 1978, The Elements of Programming Style: McGraw-Hill, New York, 168 p.

Lahey Computer Systems, Inc., "Lahey Fortran Language System Reference Manual", Revision B, November, 1990, Incline Village, Nev., 260 p.

_____ "Programmer's Reference", Revision B, November, 1990, Incline Village, Nev., 126 p.

Leiden University of the Netherlands, FORCHECK--Fortran Verifier Programming Aid and Software Engineering Tool: Forcheck is a registered trademark by Leiden University of the Netherlands and solely distributed in the USA by Computing & Systems Consultants of Raleigh, N.C.

Moniot, Robert, 1993, FTNCHEK Version 2.7, Fordham University, 47 p.

Polyhedron Software Limited, Copyright 1986-1994, plusFORT--Adding Value to FORTRAN.

Yourdon, E., 1975, Techniques of Program Structure and Design, Prentice-Hall.

APPENDIX A. SYSTEM DOCUMENTATION (SYSDOC) PROGRAM

The SYStem DOCumentation (SYSDOC) program can be used to produce software documentation directly from source code. The program reads source code that conforms to the standard and this convention and generates reports describing the processed software. The SYSDOC processing options and the optional input file sysdoc.opt are described in table A.1. The input, intermediate, and output files for SYSDOC are described in table A.2. The format of the required input file [prefix].inp is described in table A.3.

Table A.1. Description of the SYSDOC processing options and the optional input file sysdoc.opt

Record type	Columns	Description
1		Include as many record type 1 as needed. Records may appear in any order. Only the options that are being modified are required.
	1-6	Name of option, identified below.
	8-15	Value for option.
2		Comments. May appear mixed in any order with record type 1.
	1	Identifier indicating this is a comment. SYSDOC recognizes an asterisk (*) and a pound sign (#).
	2-80	Comment.

Name	Default	Description and valid ranges
The following 2 options are related to file management.		
prefix	sysdoc	Prefix to be used for input and output files _.inp, _.out, _.com, _.int, _.unk, _.ina, and _.ins.
clean	1	Flag indicating disposition of intermediate files. 0 - keep files 1 - delete files (default)
The following 10 options effect the formatting of the reports. The width of the lines in the report depends on the values for marglf and margrt. The report width is equal to marglf + 94 + margrt. In characters, it is at least 96 and at most 134 characters wide.		
pgform	1	Format page with page ejects, blank line padding, and page numbers. 0 - no (note: no index will be generated) 1 - yes (note: each routine will begin at tope of page)
marglf	10	Blank spaces for left margin, 1 - 20.
margrt	18	Blank spaces for right margin, 1 - 20.
margtp	2	Blank lines before heading for a routine, 1 - pgline.
pgline	80	Number of lines per page 20 - 1000. (note: not used when pgform = 0)
pgmain	1	Starting page number for main report, 1 - 9999. (note: not used when pgform = 0)
pgintr	1	Starting page number for intrinsics report, 1 - 9999. (note: not used when pgform = 0)
pgcomn	1	Starting page number for common block report, 1 - 9999. (note: not used when pgform = 0)
pgunkn	1	Starting page number for unknown routines report, 1 - 9999. (note: not used when pgform = 0)
index	0	Generate the file containing the indexes. 0 - no, do not generate the file (default) 1 - yes, generate the file (note: index is not generated if pgform = 0)

Table A.1. Description of the SYSDOC processing options and the optional input file sysdoc.opt—Continued

Name	Default	Description and valid ranges
The following 3 options determine the processing options for SYSDOC. By default, the program will generate reports from source code input.		
stxref	1	Flag indicating status of intermediate files SDcall.t SDnam.t SDarg.t SDcuse.t SDadef.t. 0 - use files generated in previous run 1 - generate the files (default)
stargf	1	Flag indicating the status of intermediate SDfarg.t. 0 - use file generated in a previous run 1 - generate the file (default)
merge	1	Merge the intermediate, unformatted files together to generate the formatted main, common blocks, intrinsics, and unknown routines reports ([prefix].out, [prefix].com, [prefix].int, and [prefix].unk). 0 - no, do not generate reports 1 - yes, generate reports (default)
		Miscellaneous options.
debug	0	Debugging output level. 0 - no debugging messages displayed 1 - temporary files not deleted 2 - minimum debugging messages 3 - maximum debugging messages
intlen	4	Default integer length. 2 - short 4 - long (default)

Table A.2. Input, intermediate, and output files for SYSDOC

Status	File names	Description
input	sysdoc.opt	Contains changes to the default report processing and formatting options. Described in table A.1. Optional.
The following files are read or written when stxref = 1. The intermediate files are deleted by the program when clean = 1.		
input	[prefix].inp	Contains a list of the source code and (or) cross-reference files to be documented. Described in table A.3. Required. The default prefix is sysdoc; it may be modified in sysdoc.opt.
input	[name].___	All source code files identified in [prefix].inp. The file suffix is usually f, for, or f77. A [name].XRF file will be generated for each of these files.
output/input	[name].XRF	All cross-reference files generated from the source code files identified in [prefix].inp and all cross-reference files identified in [prefix].inp.
intermediate	SDadef.t	Contains definitions for all dummy arguments. Listed by routine in the order they were encountered.
intermediate	SDarg.t	Contains type, size, and Input/Modify/Output status for all dummy arguments. Listed by routine in the order the routines were processed.
intermediate	SDcall.t	Contains the names of the external and intrinsic routines called by each routine. Listed in the order the calling routines were processed.
intermediate	SDcuse.t	Contains information on common block usage. Includes Input/Modify/Output/Argument status for how common variables are used. Listed in the order the routines were processed.
intermediate	SDdoc.t	Contains the purpose information. Listed in the order the routines were processed.
intermediate	SDnam.t	Contains a list of the routines processed. Listed in the order the routines were encountered. Includes the routine name, the name of the file containing the routine, the position in the file, and the type of routine.
intermediate	SDsadef.t	Sorted version of SDadef.t. Sorted alphabetically by dummy argument name.
intermediate	SDsarg.t	Sorted version of SDarg.t. Sorted alphabetically by argument name.
intermediate	SDscalld.t	Sorted version of SDcall.t. Sorted alphabetically by the name of the calling routine.
intermediate	SDscallu.t	Sorted version of SDcall.t. Sorted by the name of the called routine.

Table A.2. Input, intermediate, and output files for SYSDOC—Continued

Status	File names	Description
intermediate	SDscuse.t	Sorted version of SDcuse.t. Sorted alphabetically by the routine name.
intermediate	SDscusex.t	Sorted version of SDcuse.t. Sorted alphabetically by the name of the common block and the common block variable names.
intermediate	SDsdoc.t	Sorted version of SDdoc.t. Sorted alphabetically by routine name.
intermediate	SDsnam.t	Sorted version of SDnam.t. Sorted alphabetically by routine name.
The following 2 intermediate files are created when stargf = 1. The intermediate files above are required to generate these files. These files are deleted by the program when clean = 1.		
intermediate	SDfarg.t	Combination of files SDarg.t and SDadef.t. This file is deleted when SDsfarg.t is created.
intermediate	SDsfarg.t	Sorted version of SDfarg.t. Sorted alphabetically by routine name.
The following 6 output files are created when merge = 1. The intermediate files above generated when stxref = 1 and stargf = 1 are required to generate these files. The default prefix is sysdoc; it can be modified in sysdoc.opt.		
output	[prefix].out	Formatted report for all processed routines. See figure A.2.
output	[prefix].com	Formatted report on common block usage. See figure A.3.
output	[prefix].int	Formatted report on usage of intrinsic routines. See figure A.4.
output	[prefix].unk	Formatted report on usage of unknown routines. See figure A.5.
output	[prefix].ina	Combined index of routines processed. Includes page numbers for [prefix].out, [prefix].com, [prefix].int, and [prefix].unk. Listed alphabetically by name. Not generated when pgform = 0.
output	[prefix].ins	Sorted version of [prefix].ina. Sorted by report. Not generated when pgform = 0.

Table A.3. Format of the required input file [prefix].inp

Record type	Columns	Description
1		Include as many record type 1 as needed. Records may appear in any order.
	1-64	Name of file containing source code or SYSDOC generated cross reference. May be any name that is valid on the computer system being used. May include a complete path name if necessary. The length of the file name may be restricted to less than 64 characters on some systems. Each file may contain one or more subroutines and (or) functions. A main program is not required but may be included. The characters >, \, and / are recognized as delimiters for directory names. The suffix XRF is used for cross-reference files generated in a previous run. Source code files are usually identified by the suffix f, for, or f77. Any combination of source code and cross-reference files may be entered.

The example input files shown in figure A.1 were used to generate the example reports in figures A.2, A.3, and A.4. The sysdoc.opt file indicates that the prefix used for the input file and the output files is test1. The reports will be formatted with one blank space for the left and right margins (marglf = 1 and margrt = 1). The reports will have no form feeds or fixed page spacing (pgform = 0). There will be two blank lines before the start of each routine (default for margtp is 2). All intermediate files will be saved (clean = 0). The test1.inp file indicates that source code will be read from two files, utilcc.f and utilcn.f. The files utilcc.f and utilcn.f are distributed with the program. The routines contained in these two files are for manipulation of character arrays and are a subset of a larger library of routines available from the authors.

Figure A.1. Example input files sysdoc.opt and test1.inp.

File name	**Contents**
`sysdoc.opt`	`* no page formatting,`
	`* 1 space on margins, save files`
	`prefix test1`
	`pgform  0`
	`marglf  1`
	`margrt  1`
	`clean   0`
`test1.inp`	`utilcc.f`
	`utilcn.f`

Figure A.2 contains the test1.out routine report generated when SYSDOC was run using the input files shown in figure A.1. This report contains documentation for each of the functions, subroutines, and main programs processed by SYSDOC. The report is arranged in alphabetical order by routine name. For each routine processed, the report may contain:

routine name - as a header, left and right justified
identification - routine type, order number in the source file, and the name of the source file
description - the purpose of the routine
list of arguments - order number, name, type and size, status (input, modify, or output), and description
common usage - common blocks used, variables used from common, and status (input, modify, output, or passed as an argument)
called routines - list of any called routines, both intrinsics and externals
calling routines - lists any routines in the report that call this routine and the names of the code groups that contain the calling routines

Note that the documentation in figure A.2 is essentially as SYSDOC generated it. The only formatting done using FrameMaker on a UNIX workstation was to select font type and size for the text (Courier 8 point bold) and the routine names (Helvetica 12 point bold, with a box drawn around it) and to add page breaks.

Figure A.2. Example report of documented routines.

```
ADCOMA                                                              ADCOMA

 This SUBROUTINE is number 8 in file utilcn.

 This routine places a comma(s) in the real number in the string
 every 3 digits.  String may include sign, decimal point, and
 decimal digits.  If there is not enough room in the string, the
 string is returned as it was entered.

 ARGUMENTS:
                 declaration
  order   name   type  size   status  explanation
  -----  ------  -----------  ------  ---------------------------------------------------------
    1    LEN      I*4            I    size of character array STR
    2    STR      C*1 (V)        M    character array of size LEN

 CALLS:

  routine
  -------
  COPYC      LENSTR      LFTSTR      RHTSTR      STRFND      ZIPC

 CALLED BY:

  unknown

CARVAR                                                              CARVAR

 This SUBROUTINE is number 12 in file utilcc.

 Places the contents of the character*1 array CARY of size LENA
 into the character string CVAR of length LEN.  If the length of
 the array is greater than the length of the string (LENA > LENV),
 CVAR will contain the first LENV characters from CARY.  If the
 array is smaller than the string (LENA < LENV), then CVAR will
 be padded at the end with blanks.

 ARGUMENTS:
                 declaration
  order   name   type  size   status  explanation
  -----  ------  -----------  ------  ---------------------------------------------------------
    1    LENA     I*4            I    size of input character*1 array CARY
    2    CARY     C*1 (V)        I    input character array of size LENA
    3    LENV     I*4            I    available length for output character variable CVAR
    4    CVAR     C*V            O    output character variable of length LENV

 CALLS:

  none

 CALLED BY:

  unknown
```

CHDECE

This SUBROUTINE is number 9 in file utilcn.

Convert the character array STR to its real equivalent. STR is expected to contain a right-justified number and may include a sign, decimal point, exponent, and decimal digits (+, -, ., D, E, and 0-9). If the character array contains an invalid character or cannot be converted, ERRFLG will be returned with a value of 1; RVAL is set to -R0MAX in this case. (Note: R0MAX, the largest representable number, is determined in subroutine NUMINI. It is machine dependent.)

ARGUMENTS:

```
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    size of character array STR
   2    STR      C*1 (V)        I    character array containing real value
   3    RVAL     R              O    real value
   4    ERRFLG   I*4            O    flag indicating success of conversion
                                     0 - successful
                                     1 - unsuccessful
```

COMMON USAGE:

```
 block    name   status
 ------  ------  ------
 RCONST  R0MAX      I
```

CALLS:

```
 routine
 -------
 CHRDEC      NUMINI
```

CALLED BY:

```
 unknown
```

CHINTE CHINTE

This SUBROUTINE is number 10 in file utilcn.

Convert the character array STR to its integer equivalent. STR is expected to contain a right-justified integer value and may include a sign and decimal digits (+, -, and 0-9). If the character array contains an invalid character or cannot be converted, ERRFLG will be returned with a value of 1; IVAL is set to 0 in this case.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	LEN	I*4	I	size of character array STR
2	STR	C*1 (V)	I	character array of size LEN containing an integer value
3	IVAL	I*4	O	integer value
4	ERRFLG	I*4	O	flag indicating success of conversion 0 - successful 1 - unsuccessful

CALLS:

routine
CHRDIG

CALLED BY:

unknown

CHKSTR CHKSTR

This INTEGER FUNCTION is number 13 in file utilcc.

Searches the columns of the array STR2 for a match to the array STR1. If a match is found, CHKSTR returns the column number of the first column in STR2 containing STR1. If no match is found, CHKSTR returns a zero.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	LEN	I*4	I	size of character array STR1 and number of rows in STR2
2	NSTR	I*4	I	number of strings to be checked, number of columns in STR
3	STR1	C*1 (V)	I	character array of size LEN to searched
4	STR2	C*1 (V,V)	I	character array of size LEN,NSTR to be searched

CALLS:

none

CALLED BY:

unknown

```
This SUBROUTINE is number 14 in file utilcc.

Copy LEN characters from array STR1 to array STR2.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    size of character arrays STR1 and STR2
   2    STR1     C*1 (V)        I    input character array of size LEN
   3    STR2     C*1 (V)        O    output character array of size LEN

CALLS:

 none

CALLED BY:

 group     routine
 --------  -------
 utilcc    DATLST
 utilcn    DECCHX
```

This REAL FUNCTION is number 11 in file utilcn.

Convert the character array STR to its real equivalent. STR is expected to contain a right-justified number and may include a sign, decimal point, exponent, and decimal digits (+, -, ., D, E, and 0-9). If the character array contains an invalid character or cannot be converted, CHRDEC will return -R0MAX. (Note: R0MAX, the largest representable number, is determined in subroutine NUMINI. It is machine dependent.) Leading blanks are ignored. Trailing blanks are treated as zero.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	LEN	I*4	I	size of array STR
2	STR	C*1 (V)	I	character array of size LEN containing a real value

COMMON USAGE:

block	name	status
RCONST	R0MAX	I

CALLS:

routine	
CHRDIG	NUMINI

CALLED BY:

group	routine
utilcn	CHDECE

CHRDEL

This SUBROUTINE is number 15 in file utilcc.

Deletes the character in array position POS in STRING and then shifts the rest of the array left one position. The last character in the STRING is set to blank. If POS is greater than LEN, no action is taken.

ARGUMENTS:

order	name	declaration type	declaration size	status	explanation
1	LEN	I*4		I	size of character array STRING
2	POS	I*4		I	array position of the character to be deleted from STRING
3	STRING	C*1	(V)	M	character array of size LEN

CALLS:

none

CALLED BY:

group	routine
utilcn	DECCHX

CHRDIG

This INTEGER FUNCTION is number 13 in file utilcn.

CHRDIG returns the integer equivalent of a single character. The expected characters are '0' - '9'. If CHR contains a character other than expected, CHRDIG returns a -1.

ARGUMENTS:

order	name	declaration type	declaration size	status	explanation
1	CHR	C*1		I	a single character

CALLS:

none

CALLED BY:

group	routine				
utilcn	DECCHX	CHINTE	CHRDEC	CHRDPR	CRINTE

This DOUBLE PRECISION FUNCTION is number 12 in file utilcn.

Convert the character array STR to its double precision equivalent. STR is expected to contain a right-justified number and may include a sign, decimal point, exponent, and decimal digits (+, -, ., D, E, and 0-9). If the character array contains an invalid character or cannot be converted, CHRDPR will return -D0MAX. (Note: D0MAX, the largest representable double precision number, is determined in subroutine NUMINI. It is machine dependent.) Leading blanks are ignored. Trailing blanks are treated as zero.

ARGUMENTS:

		declaration		
order	name	type size	status	explanation
1	LEN	I*4	I	size of array STR
2	STR	C*1 (V)	I	character array of length LEN containing a real value

COMMON USAGE:

block	name	status
DCONST	D0MAX	I

CALLS:

routine	
CHRDIG	NUMINI

CALLED BY:

unknown

CHRINS CHRINS

This SUBROUTINE is number 16 in file utilcc.

Inserts the character CHAR into array position COL in the character array STRING. First, array positions COL thru LEN-1 are shifted forward one space. Then CHAR is placed in array position COL. The original STRING(LEN) value is deleted in the process.
WARNING: CHRINS does not check that COL is a valid position.

ARGUMENTS:

		declaration		
order	name	type size	status	explanation
1	LEN	I*4	I	size of array STRING
2	COL	I*4	I	position in STRING that CHAR is to be inserted
3	CHAR	C*1	I	character to be inserted in STRING
4	STRING	C*1 (V)	M	character array of size LEN

CALLS:

none

CALLED BY:

group	routine
utilcn	DECCHX

CHRINT CHRINT

This INTEGER FUNCTION is number 14 in file utilcn.

CHRINT returns the integer equivalent of the character array STR. STR is expected to contain a right-justified integer value and may include a sign and digits (+, -, and 0-9). CHRINT returns a zero if the character array contains an invalid character or cannot be converted. Leading blanks are ignored and trailing blanks are treated as a zero.

ARGUMENTS:

		declaration		
order	name	type size	status	explanation
1	LEN	I*4	I	size of character array STR
2	STR	C*1 (V)	I	character array of size LEN containing an integer value

CALLS:

routine
CRINTE

CALLED BY:

unknown

CKNBLK CKNBLK

This INTEGER FUNCTION is number 17 in file utilcc.

Checks the character array CBUF for the occurrence of blanks.
CKNBLK returns a zero if the array contains all blanks. CKNBLK
returns a 1 if there are any non-blank characters in CBUF.

ARGUMENTS:

```
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    size of character array CBUF
   2    CBUF     C*1 (V)        I    character array of size LEN
```

CALLS:

none

CALLED BY:

unknown

CKNBLV CKNBLV

This INTEGER FUNCTION is number 10 in file utilcc.

Looks for the first non-blank character in CBUF. CKNBLV returns
the position of the first non-blank character. If CBUF contains
all blanks, CKNBLV returns a zero.

ARGUMENTS:

```
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    size of character array CBUF
   2    CBUF     C*1 (V)        I    character array of size LEN
```

CALLS:

none

CALLED BY:

unknown

COPYC

This SUBROUTINE is number 18 in file utilcc.

Copies the contents of character array ZIP to character array X.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	LEN	I*4	I	size of character arrays XIP and X
2	ZIP	C*1 (V)	I	input character array of size LEN
3	X	C*1 (V)	O	output character array of size LEN

CALLS:

none

CALLED BY:

group	routine	
utilcn	ADCOMA	INTCHR

CRINTE

This INTEGER FUNCTION is number 15 in file utilcn.

CRINTE returns the integer equivalent of the character array STR. STR is expected to contain a right-justified integer value and may include a sign and digits (+, -, and 0-9). CRINTE returns ERRINT if the character array contains an invalid character or cannot be converted. Leading blanks are ignored and trailing blanks are treated as a zero.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	ERRINT	I*4	I	value to be returned when the integer value cannot be determined
2	LEN	I*4	I	size of character array STR
3	STR	C*1 (V)	I	character array of size LEN containing an integer value

CALLS:

routine
CHRDIG

CALLED BY:

group	routine	
utilcn	CHRINT	CRINTX

CRINTX

```
This INTEGER FUNCTION is number 16 in file utilcn.

CRINTX returns the integer equivalent of the character array STR.
STR is expected to contain a right-justified integer value and may
include a sign and digits (+, -, and 0-9).  CRINTX returns -999
if the character array contains an invalid character or cannot be
converted.  Leading blanks are ignored and trailing blanks are
treated as a zero.
Convert a character array to its integer equivalent.
The integer is expected to be right justified in STR.
For an invalid integer, -999 is returned.  Valid characters
are '0' - '9', '+', and '-' .  Leading blanks are ignored.
Trailing blanks are treated as 0.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4           I    size of character array STR
   2    STR      C*1 (V)       I    character array of size LEN

CALLS:

 routine
 -------
 CRINTE

CALLED BY:

 unknown
```

CTRSTR

```
This SUBROUTINE is number 19 in file utilcc.

Centers the characters within the character array TITLE.  Embedded
blanks are preserved.  Leading and trailing blanks are balanced.
TITLE is restricted to a maximum size of 132.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4           I    size of character array TITLE, 1 <= LEN <= 132
   2    TITLE    C*1 (V)       M    character array of size LEN

CALLS:

 none

CALLED BY:

 unknown
```

This SUBROUTINE is number 20 in file utilcc.

Places the contents of the character variable CVAR of expected length LENV into the character*1 array CARY of length LENV. If the length of the variable is greater than the length of the array (LENV > LENA), CARY will contain the first LENA characters of CVAR. If the variable is shorter than the array (LENV < LENA), then CVAR will be padded at the end with blanks.

ARGUMENTS:

```
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LENV     I*4            I    length of input character variable
   2    CVAR     C*V            I    input character variable of length LENV
   3    LENA     I*4            I    size of output character array
   4    CARY     C*1 (V)        O    output character array of size LENA
```

CALLS:

none

CALLED BY:

unknown

DATCHR

```
This SUBROUTINE is number 21 in file utilcc.

Places a year, month, and day date into the character array DBUFF.
The month is represented as a number.  If MO is not a valid month
(1-12), the month and day will not be included in DBUFF.  If DY is
not a valid day (1-31), the day will not be included in DBUFF.
(Note that no check is made to verify that DY is a valid day for
the month MO.) OLEN is the actual number of characters used to
represent the date, the remainder of DBUFF is padded with blanks.
Examples:
           YR   MO   DY   OLEN  DBUFF
          1994  12   31    10   1994/12/31
           92    1    1     6   92/1/1
          1992  11    0     7   1992/11

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  --------------------------------------------------------
   1    YR       I*4            I    year
   2    MO       I*4            I    month
   3    DY       I*4            I    day
   4    OLEN     I*4            O    number of array positions used for date
   5    DBUFF    C*1 (10)       O    output character array

CALLS:

 routine
 -------
 INTCHR     ZIPC

CALLED BY:

 unknown
```

DATLST

```
This SUBROUTINE is number 22 in file utilcc.

Converts an integer date (year, month, day, hour, minute, and
second) to a character representation, with the month represented
as a 3-character abbreviation.  If the hour, minute, and second are
all zero, they are not included.  If the date array contains an
invalid date or time, the character representation will contain all
blanks.  The program does check that the day is valid for the month.
A 2-digit year is assumed to occur in the first century.  Examples:
<-- DATE(1-6) -->   <----- DSTRNG ------>   LEN
1986,2,14,10,30,0   1986 FEB. 14 10:30:00    21
92,12,31,0,0,0      92 DEC. 31               10

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    DATE     I*4 (6)        I    date (year, month, day, hour, minute, second)
   2    DSTRNG   C*1 (21)       O    output character array
   3    LEN      I*4            O    actual number of characters output to character array
   4    ERRCOD   I*4            O    flag indicating valid date
                                     0 - valid date
                                     1 - invalid year
                                     2 - invalid month
                                     3 - invalid day
                                     4 - invalid hour
                                     5 - invalid minute
                                     6 - invalid second
                                     If the year or month is invalid, the remaining date
                                     elements are not checked.  If the year and month are
                                     valid, ERRCOD is set for the smallest invalid date element;
                                     i.e.,  if day and hour are both invalid, ERRDOC =  4.

CALLS:

 routine
 -------
 CHRCHR     DAYMON     INTCHR     ZIPC

CALLED BY:

 unknown
```

DECCHR DECCHR

This SUBROUTINE is number 1 in file utilcn.

Converts a real number to a character array. The number is left or right justified in the array based on the value of JUST. If the number will not fit in the array, exponential notation is used and the number is right justified in the array. For left-justified numbers, STR is padded with trailing blanks. For right-justified numbers, STR is padded with leading blanks.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	REAIN	R	I	real value to be converted to a character array
2	LENGTH	I*4	I	available size for output character array STR
3	JUST	I*4	M	output justification 0 - right justified 1 - left justified will be forced to 0 if an exponent is required
4	JLEN	I*4	O	actual number of characters placed in string, includes any leading blanks if number is right justified
5	STR	C*1 (V)	O	output character array of size LENGTH

COMMON USAGE:

block	name	status
ICONST	RPREC	I
RCONST	R0MIN	I
RCONST	RP1MIN	A

CALLS:

```
 routine
 -------
 ABS         AINT        ALOG10      ANINT       IABS        INDEX       INT         INTCHR
 LEN         NINT        NUMINI      REAL        RWDIGS
```

CALLED BY:

unknown

DECCHX DECCHX

This SUBROUTINE is number 5 in file utilcn.

Converts a real number to a character array. The number is right justified in the array. The number will be represented with SIGDIG significant digits and DECPLA decimal places, if there is room in STR. If there is not room, the number will be represented using exponential notation.

ARGUMENTS:

order	name	declaration type	declaration size	status	explanation
1	REAIN	R		I	real value to be converted to a character array
2	LEN	I*4		I	available size for output character array
3	SIGDIG	I*4		I	significant digits for output
4	DECPLA	I*4		I	number of decimal places for output 0 - no decimal places <0 - force exponential output
5	STR	C*1	(V)	O	output character array of size LEN

CALLS:

```
 routine
 -------
 ABS         CHRCHR      CHRDEL      CHRDIG      CHRINS      DIGCHR      INTCHR
```

CALLED BY:

unknown

DIGCHR DIGCHR

This CHARACTER FUNCTION is number 6 in file utilcn.

DIGCHR returns the character equivalent of a single digit. If the integer provided is not within the valid range 0 thru 9, DIGCHR will return a zero.

ARGUMENTS:

order	name	declaration type	declaration size	status	explanation
1	DIG	I*4		I	digit to be converted to a character

CALLS:

none

CALLED BY:

```
 group     routine
 --------  -------
 utilcn    DECCHX      INTCHR
```

DPRCHR

This SUBROUTINE is number 2 in file utilcn.

Converts a double precision number to a character array. The number is left or right justified in the array based on the value of JUST. If the number will not fit in the array, exponential notation is used and the number is right justified in the array. For left-justified numbers, STR is padded with trailing blanks.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	DPRIN	D	I	double precision value to be converted to a character array
2	LENGTH	I*4	I	available size for output character array STR
3	JUST	I*4	M	output justification 0 - right justified 1 - left justified will be forced to 0 if an exponent is required
4	JLEN	I*4	O	actual number of characters placed in string, includes any leading blanks if number is right justified
5	STR	C*1 (V)	O	output character array of size LENGTH

COMMON USAGE:

block	name	status
DCONST	D0MIN	I
DCONST	DP1MIN	A
ICONST	DPREC	I

CALLS:

```
 routine
 -------
 DABS        DBLE        DINT        DLOG10      DNINT       DWDIGS      IABS        IDINT
 IDNINT      INDEX       INTCHR      LEN         NUMINI
```

CALLED BY:

unknown

```
This INTEGER FUNCTION is number 4 in file utilcn.

DWDIGS returns the number of whole digits in the double precision
value.  The value is expected to be greater than 0, no check is
made to verify that this is true.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    DVAL     D              I    double precision value

CALLS:

 routine
 -------
 DLOG10      IDINT

CALLED BY:

 group     routine
 --------  -------
 utilcn    DPRCHR
```

INTCHR INTCHR

This SUBROUTINE is number 7 in file utilcn.

Converts an integer number to a character array. The number is left or right justified in the array based on the value of JUST. The maximum number of digits for the integer is 9. For numbers that are left justified, STRNG is padded with trailing blanks. For numbers that are right justified, STRNG is padded with leading blanks.

ARGUMENTS:

```
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    INTIN    I*4            I    integer value to be converted to a character array
   2    LENA     I*4            I    available size for output character array
   3    JUST     I*4            I    output justification
                                     0 - right justified in the field
                                     1 - left justified in the field
   4    JLEN     I*4            O    actual number of characters output to character array
   5    STRNG    C*1 (V)        O    output character array of size LENA
```

CALLS:

```
 routine
 -------
 ALOG10      COPYC       DIGCHR      INT         MOD         REAL        ZIPC
```

CALLED BY:

```
 group     routine
 --------  -------
 utilcc    DATCHR      DATLST
 utilcn    DECCHR      DECCHX      DPRCHR
```

LENSTR LENSTR

This INTEGER FUNCTION is number 1 in file utilcc.

LENSTR returns the position of the last non-blank character in the array STR. If STR contains all blanks, LENSTR returns a zero.

ARGUMENTS:

```
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    size of character array STR
   2    STR      C*1 (V)        I    character array of size LEN
```

CALLS:

```
 none
```

CALLED BY:

```
 group     routine
 --------  -------
 utilcn    ADCOMA
```

LFTSTR LFTSTR

```
This SUBROUTINE is number 2 in file utilcc.

Left justifies the characters within the character array TITLE.
Imbedded blanks are preserved.  Title is restricted to a maximum
size of 132.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  --------------------------------------------------------
   1    LEN      I*4            I    size of character array TITLE, 1 <= LEN <= 132
   2    TITLE    C*1 (V)        M    character array of size LEN

CALLS:

 none

CALLED BY:

 group     routine
 --------  -------
 utilcn    ADCOMA
```

NUMINI NUMINI

```
This SUBROUTINE is number 17 in file utilcn.

Initialize machine dependent floating point constants.

ARGUMENTS:

 none


COMMON USAGE:

 block    name   status
 ------  ------  ------
 DCONST  D0MAX      O
 DCONST  D0MIN      O
 DCONST  DP1MIN     M

 ICONST  DPREC      M
 ICONST  RPREC      O

 RCONST  R0MAX      O
 RCONST  R0MIN      O
 RCONST  RP1MIN     M

CALLS:

 routine
 -------
 INT         LOG10

CALLED BY:

 group     routine
 --------  -------
 utilcn    CHDECE      DECCHR      DPRCHR      CHRDEC      CHRDPR
```

QUPCAS QUPCAS

```
This SUBROUTINE is number 9 in file utilcc.

Converts all lowercase letters in STRING to uppercase.  All
uppercase letters are left in uppercase.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    length of character array STRING
   2    STRING   C*1 (V)        M    character array of length LEN

CALLS:

 routine
 -------
 CHAR        ICHAR       MOD

CALLED BY:

 unknown
```

RHTSTR RHTSTR

```
This SUBROUTINE is number 11 in file utilcc.

Right justifies the characters within the character array TITLE.
Embedded blanks are preserved.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    size of character array TITLE, 1 <= LEN <= 132
   2    TITLE    C*1 (V)        M    character array of size LEN

CALLS:

 none

CALLED BY:

 group     routine
 --------  -------
 utilcn    ADCOMA
```

RWDIGS RWDIGS

```
This INTEGER FUNCTION is number 3 in file utilcn.

RWDIGS returns the number of whole digits in the real value.  The
real value is expected to be greater than 0, no check is made to
verify that this is true.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    RVAL     R              I    real value

CALLS:

 routine
 -------
 ALOG10     INT

CALLED BY:

 group     routine
 --------  -------
 utilcn    DECCHR
```

STRFND STRFND

```
This INTEGER FUNCTION is number 3 in file utilcc.

STRFND returns the position in the array STR where the array FSTR
begins.  If FSTR is not contained in STR, STRFND returns a zero.
If FLEN is greater than LEN, STRFND returns a zero.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    LEN      I*4            I    size of character array being searched
   2    STR      C*1 (V)        I    character array of size LEN to be searched
   3    FLEN     I*4            I    size of character array to search for
   4    FSTR     C*1 (V)        I    character array of size FLEN to be searched for

CALLS:

 none

CALLED BY:

 group     routine
 --------  -------
 utilcn    ADCOMA
```

STRLNX STRLNX

This INTEGER FUNCTION is number 4 in file utilcc.

STRLNX returns the number of characters in the array BUFF. Leading and trailing blanks are not included in the count, embedded blanks are include in the count. If there are no non-blank characters in the array, STRLNX returns a 1.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	NPTS	I*4	I	size of the character array BUFF
2	BUFF	C*1 (V)	I	character array of size NPTS

CALLS:

none

CALLED BY:

unknown

ZIPC ZIPC

This SUBROUTINE is number 5 in file utilcc.

Fill the character array X of size LEN with the given value ZIP.

ARGUMENTS:

order	name	declaration type size	status	explanation
1	LEN	I*4	I	size of character array ZIP
2	ZIP	C*1	I	character to fill array X
3	X	C*1 (V)	O	character array of size LEN to be filled with ZIP

CALLS:

none

CALLED BY:

group	routine	
utilcc	DATCHR	DATLST
utilcn	ADCOMA	INTCHR

ZLJUST ZLJUST

```
This SUBROUTINE is number 6 in file utilcc.

Left justifies the characters in STRING.  Leading blanks are
removed and the characters are shifted to the left.  STRING is
padded with trailing blanks.  Embedded blanks are preserved.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    STRING   C*V            M    character variable

CALLS:

 routine
 -------
 LEN

CALLED BY:

 unknown
```

ZLNTXT ZLNTXT

```
This INTEGER FUNCTION is number 7 in file utilcc.

Determine the length of STRING, excluding trailing blanks and
nulls.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    STRING   C*V            I    character variable

CALLS:

 routine
 -------
 CHAR        LEN

CALLED BY:

 unknown
```

```
This SUBROUTINE is number 8 in file utilcc.

Left justify the characters in STRING.  All leading and embedded
blanks are removed.  String is padded with trailing blanks.

ARGUMENTS:
                declaration
 order   name   type  size   status  explanation
 -----  ------  -----------  ------  ---------------------------------------------------------
   1    STRING   C*V            M    character variable

CALLS:

 routine
 -------
 LEN

CALLED BY:

 unknown
```

Figure A.3 contains the test1.com common report generated when SYSDOC was run using the files shown in figure A.1. This report contains documentation for each of the common blocks used by the documented routines. The report is arranged in alphabetical order by common block name. For each common block documented, the report contains:

common name - as a header, left and right justified

common usage table - alphabetically by variable name, lists routines that use the variable, the code group that contains the routine, the order number of the routine in the code group, and the status of the variable in the routine. The possible status are O - set, I - used but not changed, M - used and then changed, and A - passed as an argument to another routine.

Note that the documentation in figure A.3 is essentially as SYSDOC generated it. The only formatting done using FrameMaker on a UNIX workstation was to select font type and size for the text (Courier 8 point bold) and the routine names (Helvetica 12 point bold, with a box drawn around it) and to add page breaks.

Figure A.3. Example report of common block usage.

```
DCONST                                                    DCONST

This COMMON BLOCK is used by:

 name   routine    group   number  status
------  -------  --------  ------  ------

 D0MAX   CHRDPR   utilcn      12      I
 D0MAX   NUMINI   utilcn      17      O

 D0MIN   DPRCHR   utilcn       2      I
 D0MIN   NUMINI   utilcn      17      O

 DP1MIN  DPRCHR   utilcn       2      A
 DP1MIN  NUMINI   utilcn      17      M
```

```
ICONST                                                    ICONST

This COMMON BLOCK is used by:

 name   routine    group   number  status
------  -------  --------  ------  ------

 DPREC   DPRCHR   utilcn       2      I
 DPREC   NUMINI   utilcn      17      M

 RPREC   DECCHR   utilcn       1      I
 RPREC   NUMINI   utilcn      17      O
```

RCONST RCONST

```
This COMMON BLOCK is used by:

 name   routine    group   number  status
------  -------  --------  ------  ------

 R0MAX   CHDECE   utilcn       9      I
 R0MAX   CHRDEC   utilcn      11      I
 R0MAX   NUMINI   utilcn      17      O

 R0MIN   DECCHR   utilcn       1      I
 R0MIN   NUMINI   utilcn      17      O

 RP1MIN  DECCHR   utilcn       1      A
 RP1MIN  NUMINI   utilcn      17      M
```

Figure A.4 contains the test1.int intrinsic report generated when SYSDOC was run using the files shown in figure A.1. This report contains documentation for each of the intrinsic routines used by the documented routines. The report is arranged in alphabetical order by intrinsic name. For each intrinsic documented, the report contains:

intrinsic name - as a header, left and right justified

calling routines - an alphabetical listing by code group and calling routine of all the documented routines that call the intrinsic

Note that the documentation in figure A.4 is essentially as SYSDOC generated it. The only formatting done using FrameMaker on a UNIX workstation was to select font type and size for the text (Courier 8 point bold) and the routine names (Helvetica 12 point bold, with a box drawn around it) and to add page breaks.

Figure A.4. Example report of intrinsic usage.

```
ABS                                                            ABS

 This called routine is a SYSTEM INTRINSIC.

 CALLED BY:

  group     routine
  --------  -------
  utilcn    DECCHR      DECCHX

AINT                                                          AINT

 This called routine is a SYSTEM INTRINSIC.

 CALLED BY:

  group     routine
  --------  -------
  utilcn    DECCHR

ALOG10                                                      ALOG10

 This called routine is a SYSTEM INTRINSIC.

 CALLED BY:

  group     routine
  --------  -------
  utilcn    DECCHR      INTCHR      RWDIGS
```

ANINT ANINT

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DECCHR
```

CHAR CHAR

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcc    QUPCAS      ZLNTXT
```

DABS DABS

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DPRCHR
```

DBLE DBLE

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DPRCHR
```

DINT DINT

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DPRCHR
```

DLOG10 DLOG10

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DPRCHR      DWDIGS
```

DNINT DNINT

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DPRCHR
```

IABS IABS

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DECCHR      DPRCHR
```

ICHAR ICHAR

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcc    QUPCAS
```

IDINT IDINT

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DPRCHR      DWDIGS
```

IDNINT IDNINT

This called routine is a SYSTEM INTRINSIC.

CALLED BY:

```
 group     routine
 --------  -------
 utilcn    DPRCHR
```

INDEX INDEX

This called routine is a SYSTEM INTRINSIC.

CALLED BY:

```
 group     routine
 --------  -------
 utilcn    DECCHR      DPRCHR
```

INT INT

This called routine is a SYSTEM INTRINSIC.

CALLED BY:

```
 group     routine
 --------  -------
 utilcn    DECCHR      INTCHR      RWDIGS      NUMINI
```

LEN LEN

This called routine is a SYSTEM INTRINSIC.

CALLED BY:

```
 group     routine
 --------  -------
 utilcc    ZLJUST      ZLNTXT      ZTRIM
 utilcn    DECCHR      DPRCHR
```

LOG10 LOG10

This called routine is a SYSTEM INTRINSIC.

CALLED BY:

```
 group     routine
 --------  -------
 utilcn    NUMINI
```

MOD MOD

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcc    QUPCAS
 utilcn    INTCHR
```

NINT NINT

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DECCHR
```

REAL REAL

```
This called routine is a SYSTEM INTRINSIC.

CALLED BY:

 group     routine
 --------  -------
 utilcn    DECCHR      INTCHR
```

Figure A.5 contains the test1.unk unknowns report generated when SYSDOC was run using the files shown in figure A.1. This report contains documentation for each of the unknown routines called by the documented routines. An unknown routine is a routine that is not a known intrinsic and was not included with the documented routines. The DAYMON routine is contained in the library that utilcc.f and utilcn.f are a subset of. For each unknown documented, the report contains:

routine name - as a header, left and right justified

calling routines - an alphabetical listing by code group and calling routine of all the documented routines that call the unknown routine

Note that the documentation in figure A.5 is essentially as SYSDOC generated it. The only formatting done using FrameMaker on a UNIX workstation was to select font type and size for the text (Courier 8 point bold) and the routine names (Helvetica 12 point bold, with a box drawn around it) and to add page breaks.

Figure A.5. Example report of unknown routines.

DAYMON **DAYMON**

```
The location of this called routine is UNKNOWN.

CALLED BY:

 group     routine
 --------  -------
 utilcc    DATLST
```