NOTE: this tech note is obsolete ...
but can be xeroxed for reference material...

Vectorization and Conversion of Fortran Programs for the CRAY-1 (CFT) Compiler

by

Lee Higbie

2240207
PREFACE

This technical note presents a number of techniques for promoting vectorization in FORTRAN programs to be run on the Cray Research CRAY-1 Computer System. Because the CRAY-1 FORTRAN Compiler (CFT) is continually being refined, this note will be updated periodically. With each update, I hope to increase the number of examples and expand it in a few other ways to improve its usefulness.

I welcome any material that might be included in future editions, such as more examples and coding techniques. I especially solicit help with Appendix B where many errors of both omission and comission undoubtedly lie. In particular, the various table positions that are blank indicate that I don't know the proper entry. Special thanks are due to the several people who sent me suggestions and examples. In particular, a great deal of help for this revision was provided by Dick Hendrickson.

LCH
CONTENTS

PREFACE ........................................ iii
1 INTRODUCTION .............................. 1-1
2 FINDING THE CENTRAL PORTION OF THE PROGRAM ...... 2-1
3 GETTING AROUND OVERLY MODULAR OR STRUCTURED PROGRAMS .... 3-1
4 RECURSION AND DIRECTING THE COMPILER TO VECTORIZE ........ 4-1
5 IRREGULAR ADDRESSING ..................... 5-1
6 MISCELLANEOUS TECHNIQUES ................. 6-1
7 REMOVING IF STATEMENTS AND USING BUILT-IN FUNCTIONS .... 7-1

A $SCILIB SUBROUTINES ......................... A-1
B FORTRAN DIALECTICAL DIFFERENCES .......... B-1
C ABORT MESSAGES AND TRACEBACK ............. C-1
D COMPILER DIRECTIVES ....................... D-1
E CHARACTER SETS ............................ E-1
SECTION 1

INTRODUCTION

This note describes techniques for helping to vectorize codes written for the CRAY-1 Computer and the CRAY-1 FORTRAN, CFT, Compiler System. Since the CFT Compiler is continually being refined, some of these techniques will become unnecessary. It is primarily intended to aid programmers vectorizing existing codes but should aid many programmers who are writing new codes to generate vectorizable loops.

Before going further, a caveat is in order. This note presents techniques for enhancing the vectorizability of codes only; it addresses few other methods of increasing program speed. Algorithm selection is generally far more important than coding techniques. For example, the FFT and various good sorting algorithms are approximately N/logN times as fast as the typical simplistic algorithms to perform the same tasks (for N input data), whereas the vectorization usually increases speed by a factor of 3 to 6. Thus, for typical dataset sizes, these best algorithms or other nearly optimal ones, are orders of magnitude faster than poor algorithms. No fancy coding techniques can overcome the use of ill chosen algorithms in such cases. Indeed, a good algorithm poorly coded is usually preferable to a poor one optimally coded.

Also, this note does not address to any great extent good programming practices which for CFT include (1) using few loops with long code blocks in preference to many short code loops; (2) judicious use of typing of variables; (3) long loops inside short loops rather than vice versa; and (4) if you are trying to get the last little bit from a vectorized loop, inserting extra parentheses starting at the end of an expression so that operations occur in an order that increases chaining. The techniques that are described are presented in six groups comprising the remaining sections of this note.

Sections 2 and 3 present the central issues that must be resolved before any useful work can be done, namely (1) finding the time consuming portions of the program and (2) circumventing overly modular or structured programming techniques. These tasks are so fundamental that compiler improvements are unlikely to be of much aid in the foreseeable future without programmer help in these areas.

Section 4 discusses recursion, feedback, or vector dependency and introduces a simple but very useful and powerful technique; using directives which allow the programmer to indicate to the compiler that an individual block of code is logically vectorizable. Frequently, although the programmer knows from the physics of a situation that the code is vectorizable, coding in a form that allows the compiler to see this may not be convenient. Directives provide a means of pointing out vectorizable code to the compiler when this happens.
Section 5 shows one way to partially vectorize codes with irregular addressing—another anathema of vectorization, but one that the compiler will work around before too long.

Section 6 is a pot pourri of "tricks" to improve vectorization that do not readily fit into any of the earlier discussions.

The final section describes removing IF statements, a syntactic construct that the compiler will soon handle, at least in some cases.

Thus, Section 5 and 7 should be of less interest to programmers who are not in a big hurry to get the highest speed from their codes.

As a general note, CFT vectorizes innermost DO-loops only; it does not vectorize IF loops. Table 1 lists typical syntactic elements that may inhibit vectorization. Except for I/O statements, which often can be moved outside of loops after the debug phase, I discuss the more difficult of these constructs and how the programmer can remove these so that CFT will vectorize loops. Although not vectorized in the usual sense, unformatted I/O statements which involve arrays are processed with vector techniques.

CFT 1.06, The July 1979 release of CFT, vectorizes loops with constructs in the easy group (table 1) and allows scalar temporaries and user-provided (but CAL) functions from the second group. However, it inhibits vectorization for a loop containing other constructs in the second group or constructs in the third or fourth group. The third group includes constructs that are theoretically vectorizable but present challenges to the compiler writers. The items in the fourth group present a theoretical impossibility so that the only real hope for vectorizing loops containing them in the near future is to break the loop into several loops with the "impossible" construct in a separate (scalar) loop. If loops can be recast so that the inner DO loops include only items in the first categories, the chance of vectorization is enhanced and such loops that do not vectorize with CFT 1.06 are more likely to vectorize in the near future.
Table 1. FORTRAN inner DO-loop constructs

<table>
<thead>
<tr>
<th>Difficulty</th>
<th>Syntactic constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy</td>
<td>- Long or complicated loops</td>
</tr>
<tr>
<td></td>
<td>- Non unit incrementing of subscripts</td>
</tr>
<tr>
<td></td>
<td>- Expressions in subscript</td>
</tr>
<tr>
<td></td>
<td>- Intrinsic function references</td>
</tr>
<tr>
<td>Straightforward</td>
<td>- Scalar temporary variables</td>
</tr>
<tr>
<td></td>
<td>- Function calls to</td>
</tr>
<tr>
<td></td>
<td>programmer-supplied functions</td>
</tr>
<tr>
<td></td>
<td>- Inner products</td>
</tr>
<tr>
<td></td>
<td>- Logical IF statements</td>
</tr>
<tr>
<td></td>
<td>- Transfer out of a loop (search</td>
</tr>
<tr>
<td></td>
<td>loop)</td>
</tr>
<tr>
<td></td>
<td>- Reduction operations</td>
</tr>
<tr>
<td>Difficult</td>
<td>- Linear recursion</td>
</tr>
<tr>
<td></td>
<td>- IF statements</td>
</tr>
<tr>
<td></td>
<td>- Some I/O</td>
</tr>
<tr>
<td></td>
<td>- Complicated subscript expressions</td>
</tr>
<tr>
<td>&quot;Impossible&quot;</td>
<td>- Nonlinear indexing</td>
</tr>
<tr>
<td></td>
<td>- Complicated branching within a loop</td>
</tr>
<tr>
<td></td>
<td>- Ambiguous subscripting</td>
</tr>
<tr>
<td></td>
<td>- Transfers into a loop</td>
</tr>
<tr>
<td></td>
<td>- Subroutine calls</td>
</tr>
<tr>
<td></td>
<td>- Nonlinear recursion</td>
</tr>
<tr>
<td></td>
<td>- Some I/O</td>
</tr>
</tbody>
</table>
SECTION 2

FINDING THE CENTRAL PORTION OF THE PROGRAM

Before spending your time vectorizing parts of the program that do not significantly affect the run time, first analyze the program to determine where it spends its time. This information may be readily available if the code is simple or if there is someone available who is familiar with it. Suppose, however, that this is the first time you've been faced with this task and that you have never seen the program before.

A typical, well behaved or "nice" program has a structure similar to that illustrated at right. With any luck, you will be able to find a similar pattern in your program and will be able to concentrate on the inner points of the program where your efforts will significantly impact the program's run time.

If you question the worth of this, look at a typical program and you are likely to see many simple vector loops that have been there all along. The trouble is, they initialize the grid and are not used for any of the computations! Since a problem with a grid of 100 points on a side has about 400 boundary points and about 10,000 interior points, working on the interior points and ignoring the boundary points -- let alone the initialization -- is clearly worthwhile. Even entirely removing the code for the boundary points leaves more than 96% of the points in the grid.

If you cannot discern the general structure, a worthwhile procedure is to use the flow analysis option in CFT to get a complete list of the subroutine calling tree and the time spent in each of the called routines. These figures tell you which routines consume the largest fractions of the time, thus it tells which routines are worth looking at. Refer to CFT Reference Manual, section 5, for a description of the Flowtrace option.
The flowtrace option adds a substantial overhead to every subroutine call and its output is lost if a job fails or executes a CALL EXIT.* Thus, if you have a program with many small subroutines, it is worthwhile to flowtrace a small case, at least for starters. You might also put a test such as the following in your code to stop the job after a reasonable length of time:

IF(SECOND().GT.50)STOP**

To use the flowtrace option (CFT Manual section 5.4.5), put ON=F on the CFT statement:

CFT,ON=F,... .

At the end of the run, you will get a table listing the time, percent of total time, number of times entered, and average time for each routine that is called as well as what routines called it and what routines it called. Only calls to FORTRAN programs that are compiled by the CFT,ON=F... statement are monitored; $FTLIB, $SCILIB, $SYSLIB, and CAL routines are not monitored nor are the FORTRAN routines compiled separately without flowtrace enabled. Because of the great difference in execution speed of vectorized code compared to non-vectorized code, use of flowtrace is recommended even if you are familiar with a program. The timing analysis of flowtrace is frequently surprising.

*EXIT might be the name of one of your routines. Thus, the system cannot automatically assume that EXIT terminates a program. If you use EXIT to terminate your program, you can still use flowtrace by inserting the following subroutine in your deck.

SUBROUTINE EXIT
STOP 'EXIT'
END

**This is one of many examples of non-standard FORTRAN employed in this note. CFT accepts all FORTRAN shown in the examples here.
SECTION 3
GETTING AROUND OVERLY MODULAR OR STRUCTURED PROGRAMS

Assume your code looks like this:

```
DO 31  I = 2,99
DO 31  J = 2,99
CENTPT = DATA(I,J)
PLEFT = DATA(I-1,J)
PTRGHT = DATA(I+1,J)
TEMP = TEMPURTR(I,J)
TEMPRT = TEMPURTR(I+1,J)
TEMPLFT = TEMPURTR(I-1,J)
CALL INTGRTE(CENTPT,PLEFT,PTRGHT,TEMPRT,TEMPLFT)
CALL EQNOST(CENTPT,TEMP)
DATA(I,J) = CENTPT
TEMPURTR(I,J) = TEMP
31 CONTINUE
```

Your first impulse probably is to put this away until there are global FORTRAN compiler that vectorize messes like this. However, this represents a very common situation and is not nearly as hopeless as it first appears. However it is hopeless for CFT thus, it is your chore to put the DO loops inside the subroutine or, conversely, the subroutines inside the DO loops. Putting DO loops inside subroutines or the converse operation is probably the most complex part of vectorizing codes and is the part that is most likely to be beneficial in the long run. The other techniques discussed in this note are more likely to be handled by the compiler or by a vectorizer someday.

Putting DO loops inside subroutines probably entails subscripting the variable names being passed to the subroutines and passing the entire arrays at once. Putting the subroutine in the loop means expanding the subroutine code in line in the loop.

In the above loops, this can be done easily. Perhaps in your problem the surface is not flat but is a sphere and so the right and left points wrap around at the ends causing nonlinear indexing. Then, you will have to try to separate the "bad points" and perhaps use some of the techniques suggested in later sections.
To illustrate putting DO loops into subroutines and vice versa, suppose the subroutines are as follows:

```
SUBROUTINE INTGRT(C,PL,PR,TL,TR)
COMMON DELTAX,DELTAT,GAMMAI,V,R
C = C + DELTAT *0.5 * (PL+PR) *DELTAX/(TR-TL)
RETURN
END

SUBROUTINE EQNST(P,T,R)
COMMON DELTAX,DELTAT, GAMMAI,V,R
TP = (P/(V*R))**GAMMAI
RETURN
END
```

Then the two rewrites of the loop look like this:

Case 1. Putting loops inside Subroutines.

The entire DO 31 loop pair replaced by:

```
CALL INTGRTV (DATA,TEMPURTR)
CALL EQNSTV (DATA, TEMPURTR)
```

Where INTGRTV is a vector version of INTGRT and EQNSTV is a vectorized EQNST.

Then, these new subroutines are:

```
SUBROUTINE INTGRTV(D,T)
DIMENSION D(100,100), T(100,100)
COMMON DELTAX, DELTAT, GAMMAIV, R
DO 32 I=2,99
DO 32 J=2,99
D (I,J) = D(I,J) + DELTAX*0.5*(D(I-1,J)+D(I+1,J))
   *DELTAX/(T(I+1,J) - T(I-1,J))
32 CONTINUE
RETURN
END
```

*Reversing the order of the I and J loops would cause an unvectorizable dependency; see section 4.
SUBROUTINE EQNOSTV (P,T)
DIMENSION P(100,100), T(100,100)
DO 33 I=2,99
DO 33 J=2,99
T(I,J) = (P(I,J)/(V*R))**GAMMAI
33 CONTINUE
RETURN
END

Case 2: Putting the subroutines inside the loops.

In this case, the DO 31 pair of loop becomes

DO 34 I=2,99
DO 34 J=2,99
DATA (I,J) = DATA (I,J) + DELTAT *0.5* (DATA(I+1,J)
$ +DATA(I-1,J)) *DELTAX/TEMPURTR(I+1,J)
TEMPURTR(I,J) = (DATA(I,J)/(V*R))**GAMMAI
34 CONTINUE
RETURN
END

The second alternative is also especially suitable for functions, i.e., expand the code in line as in the next example:

DO 35 I = 1,1000
35 X(I) = ALOG2(Y(I)+1)...
   ...
   ...
FUNCTION ALOG2 (X)
DATA CST /.../
ALOG2 = CST*ALOG(Y(I)+1)
RETURN
END

DO 36 I = 1,1000
36 X(I) = CST*ALOG(Y(I)+1)...
   ...
   ...
FUNCTION ALOG2 (X)
DATA CST /.../
ALOG2 = CST*ALOG(Y(I)+1)
RETURN
END
The DO 35 loop will not vectorize because of the call to a routine that the compiler doesn't recognize but the DO 36 loop will vectorize.

One common coding technique is to use vector subroutines such as VADD, VMULT, and so on. The principal part of the program may then look like this:

```
        
        
        CALL VADD(A,B,C,N)
        CALL VMULT(C,A,E,N)
        CALL VADD(E,B,A,N)
        
```

Expanding these subroutines in line and, where possible, combining the many DO loops into a few will ensure vectorization and will allow intermediate variables to be held in registers rather than being returned to memory:

```
DO 37 I = 1,N
A(I) = (B(I) + A(I)) * A(I) + B(I)
37 CONTINUE
```

Presumably, the VADD, VMULT, etc. vectorize but the DO 37 loop is faster because the sum A + B and the product (A + B) * A do not have to be stored, but can be kept in a register and A does not have to be fetched a second time. Thus, the DO 34 loop is significantly faster than the series of calls.
SECTION 4

RECURSION AND DIRECTING THE

COMPILER TO VECTORIZE

Suppose the key inner loop in your program is like the DO 41 inner loop, which doesn't vectorize and is therefore one that you want to spend some time on.

\[
\text{DO 41 I = 1,100}
\]

\[
\text{A(I) = A(I+L) ...}
\]

\[
\text{41 CONTINUE}
\]

If the loop is truly recursive*, the situation may be hopeless. However, if the value of L is such that there is no recursion (e.g., if L is greater than 1000), the easiest approach is to try directing the compiler to vectorize the loop and see if the answers remain the same. Placing the following compiler directive in front of the DO loop to be vectorized allows the compiler to vectorize a loop that has an apparent vector dependency or recursion:

\[
\text{CDIR$ IVDEP (see CFT manual sections 5.4, 5.4.3)}
\]

In other words, if either real or imagined recursion causes the loop not to be automatically vectorized by the compiler, the IVDEP compiler directive causes the computations to be done in vector mode. Note, however, that if CALL or IF statements or anything besides or in addition to apparent recursion prevents vectorization, CDIR$ IVDEP has no effect. Also, the effect of the IVDEP is limited to only the next DO loop; a separate IVDEP must be provided for each loop with an ignorable dependency; and the IVDEP should immediately precede the DO statement.

Returning to the example, first try printing some of the A terms the first few times through the vectorized loop to assure that vectorizing the loop does not change the results. Though this hardly proves that no problems can arise, it may help your analysis. This brute force approach is inelegant and error-prone, especially in those cases where one's insight into the physics of the situation does not provide some assurance that the loop is recursion-free. If the value of L differs with each pass through the loop, you may find it useful to make a copy of the loop with the compiler directive to vectorize, ignoring vector dependencies and a copy of the loop without the directive and select for use the vectorizable block only when it is correct. This means that you need to know what values of L are acceptable for vectorization of which loops.

---

*Recursion is a buzzword used to describe the case where output is propagated back into the input. It is explained further below.
Some rules of thumb to follow are the following:

1. If the sign of L is the same as the sign of the loop increment, there is never any recursion.

2. If the sign of L is the opposite of the loop index, there is probably recursion. An exception is when the loop increment and L have a least common multiple larger than the maximum value of the loop index.

3. There is no recursion of concern if the magnitude of L is such that there is no overlap of subscripts between the right and left sides of the computation. In fact, if L divided by the loop increment is greater than 64, you have no worries because the compiler breaks loops into 64-at-a-time blocks for vectorization.

If these simple rules do not help, you may have to analyze the problem further to determine when you can safely use vector computations.

"Recursion" is a mathematical term used to describe feedback, a noun you may find more familiar and easier to remember. The phenomenon referred to is the use of the output of one pass through the loop for the input to a computation on a subsequent pass. Consider the following simple examples:

```
42  DO  I = 1,1000          SUM = 0
     X(I) = X(I - 1) + 1     DO 43  I = 1,1000
            SUM = SUM + X(I) * Y(I)
43          
```

In the code on the left, the value of X(1) is used to compute X(2); X(2) is used to compute X(3), and so on. If this were done in vector mode, all of the X terms would be fetched at once, I would be added to each of them, and only the first value would be known to be correct. In the code on the right, the value of SUM is used for each subsequent pass through the loop. Inserting a CDIR$ IVDEF would probably produce wrong answers in the DO 42 loop and would have no affect on the DO 43 loop because the reason for scalar mode of the DO 43 loop is loop collapsing, as well as recursion.

The following loops are similar to these but are nonrecursive:

```
44  DO  I = 1,1000          DO 45  I = 1,1000
    X(I) = X(I + 1) + 1     45  A(I) = A(I) + X(I) * Y(I)
```

In the DO 44 loop, no X value is reused after being computed so there is no feedback. Similarly, the DO 45 loop is not recursive because no A value is reused after being generated; it is merely stored.
To understand the effects of recursion on vectorization, it is important to realize that vectorization is an essentially parallel computation on a group of values. Consider the simple case:

\[
\begin{align*}
\text{DO 46 I} & \ = 2,3 \\
\text{A(I-1)} & \ = 3.0 \\
46 & \ B(I) \ = \ A(I)
\end{align*}
\]

This loop is equivalent to the sequential statements.

\[
\begin{align*}
\text{A(1)} & \ = 3.0 \\
\text{B(2)} & \ = \ A(2) \\
\text{A(2)} & \ = 3.0 \\
\text{B(3)} & \ = \ A(3)
\end{align*}
\]

Vectorization, in effect, reorders the sequence to:

\[
\begin{align*}
\text{A(1)} & \ = 3.0 \\
\text{A(2)} & \ = 3.0 \\
\text{B(2)} & \ = \ A(2) \quad \text{(Now 3.0)} \\
\text{B(3)} & \ = \ A(3)
\end{align*}
\]

and the "vectorized" sequence probably produces different results.

Whenever CFT encounters a loop which might be recursive, it generates correct scalar code rather than fast and possibly incorrect vector code, because vector and scalar versions of a recursive loop generally produce different results.

Recursion can cause problems if numerically equal subscript values occur on different passes through a DO loop and at least one of them is on the left of the equal sign.

There are two general classes of recursion:

1. A value is prematurely destroyed if vectorized. The preceding is an example of this. The loop can often be made non-recursive by reordering the statements or by using temporary storage.

\[
\begin{align*}
\text{DO 47 I} & \ = 2,3 \\
\text{B(I)} & \ = \ A(I) \\
47 & \ A(I-1) \ = 3.0 \\
\text{DO 48 I} & \ = 2,3 \\
\text{TEMP} & \ = \ A(I) \\
48 & \ B(I) \ = \ \text{TEMP} \\
\text{A(I-1)} & \ = 3.0
\end{align*}
\]
2. A value is not ready when needed. This is the one-line
recursion relationship:

```
DO 49 I = 2,3
49   A(I) = B*A(I-1)+C
```

Because of the group computation in vector mode, both input A values
(the group A(1), A(2)) are used to compute the output value group A(2)
and A(3). However, in this case the original A(2), not B*A(1)+C, is
used to compute A(3), a probable error.

In many cases, CPT is not able to determine whether or not a subscript
leads to recursion. For example:

```
DO 410 I = 1,10
410   A(I,J) = A(I-1, JPLUS1)
```

is recursive if J is ever the same as JPLUS1. If the programmer knows
from the physics of the situation, for example, that J and JPLUS1 will
never be the same, then our

```
CDIR$ IVDEP
```

is appropriate. Alternatively, the loop could be rewritten as

```
DO 411 I = 1,3
411   A(I,J) = A(I-1, J+1)
```

and CPT would automatically vectorize it. In general, it is an aid to
vectorization if subscripts can be explicitly written out. For example:

```
DO 412 I = 1,3
412   A(I) = A(I+N)
```

In many cases, N is not really a "variable"; it has a constant value and
often never even changes from run to run. A "variable" is used simply
to provide some flexibility in case the problem ever changes. Rather
than initialize N with

```
DATA N/3/
```

or

```
N = 3
```
it is much better to use:

```
PARAMETER (N = 3)
```

and CPT would automatically vectorize the sample loop.

In the following illustrations of recursive and non-recursive loops, assume that \(X(I) = 2I, Y(I) = -I,\) and \(Z(I) = 0\) before the codes are run. The final values of \(X\) are given after the loop for subscripts = \(1, 2, 3, \ldots\).

**Recursive:**

```
DO 413 I = 2,5
X(I) = X(I - 1) + 1.
413 CONTINUE

X = 2, 3, 4, 5, 6
```

```
CDIR$ IVDEP
DO 414 I = 2,5
X(I) = X(I - 1) + 1.
414 CONTINUE
X = 2, 3, 5, 7, 9
```

the last four of which are bad values because of forced vectorization of a recursive loop.

**Non-recursive:**

```
DO 415 I = 2,5
X(I) = Y(I - 1) + 1.
415 CONTINUE
X = 2, 0, -1, -2, -3
```

```
The compiler vectorizes the DO 415 loop automatically.
```

```
DO 416 I = 1,5
X(I) = X(I) + 1.
416 CONTINUE
X = 3, 5, 7, 9, 11
```

```
The compiler vectorizes the DO 416 loop automatically.
```

```
DO 417 I = 1,5
X(I) = Z(I) + X(I) \times Y(I)
417 CONTINUE
X = -2, -8, -18, -32, -50
```

```
The compiler vectorizes the DO 417 loop automatically.
```
Here, the 418 loop does not vectorize but the 419 loop does. The compiler does not know that L is not negative.

Recursive:

```
L = -1
.
.
.
DO 420 I = 1, L
X(I) = X(I + L)
CONTINUE
X = 0, 0, 0, 0, 0
```  

```
CDIRS IVDEP
DO 421 I = 1, L
X(I) = X(I + L)
CONTINUE
X = 0, 2, 4, 6, 8
```

The last four are incorrect because of forced vectorization of a recursive loop.

Here, the value of X(1) is fed back to compute X(2), i.e., the loop is recursive and the vectorized version of the loop produces wrong results. In scalar mode, the computations proceed...

```
X(1) = X(0) (=0 by assumption)
X(2) = X(1) (=0 from last computation)
X(3) = X(2) (=0 from last computation)
X(4) = X(3) (=0 from last computation)
X(5) = X(4) (=0 from last computation)
```
CFT generates code that executes as above because its approach to vectorization is conservative. When forced to vectorize, the loop executes:

\[ X = \text{shifted } X = (0, 2, 4, 6, 8) \]

so that \( X(2) = \) original value of \( X(1) \), not the just-computed value; similarly, \( X(3) = \) original \( X(2) \), not the newly computed value, etc. Also, in this example it is assumed that 0 is a legal subscript, i.e., \( X \) is declared DIMENSION \( X(0:50) \).

Many examples of recursion are of the following form where \( L \) is negative (that is, opposite in sign to the increment of \( J \), which is 1 here) and \( K \) is positive (of the same sign as the increment of \( I \) in this example):

\[
\begin{align*}
\text{DO 422 } & I = 1,100 \\
\text{DO 422 } & J = 1,100 \\
422 & \quad A(I,J) = A(I + K, J + L) \\
\end{align*}
\]

Here, by inverting the order of the loops, you can remove the recursion and allow vectorization by using the CDIR$ IVDEP directive. This type of loop order inversion is frequently too complex to analyze easily and you may need to go back to the physics of the situation or to that unfortunate alternative of printing gobs of values to determine a reasonable way to reorder or rewrite the code.

The following examples show a simple but real case where the compiler's overly conservative attitude is easy to see and correct. Case 1 runs about four times slower than Case 2. The cause of such a large speed increase is the complexity of the loop. Loops with very few computations generally have less speed up.
CASE 1

NL1 = 1
NL2 = 2
.
.
.
DO 423  KX = 2,3  
DO 423  KY = 2,3  

DU1 = U1(KX,KY + 1,NL1) - U1(KX,KY - 1,NL1)  
DU2 = U2(KX,KY + 1,NL1) - U2(KX,KY - 1,NL1)  
DU3 = U3(KX,KY + 1,NL1) - U3(KX,KY - 1,NL1)  

U1(KX,KY,NL2) = U1(KX,KY,NL1) + ALL*DU1 + A12*DU2 + A13*DU3  
$ + SIG*(U1(KX+1,KY,NL1) - 2.*U1(KX,KY,NL1) + U1(KX-1,KY,NL1))  

U2(KX,KY,NL2) = U2(KX,KY,NL1) + A21*DU1 + A22*DU2 + A23*DU3  
$ + SIG*(U2(KX+1,KY,NL1) - 2.*U2(KX,KY,NL1) + U2(KX-1,KY,NL1))  

U3(KX,KY,NL2) = U3(KX,KY,NL1) + A31*DU1 + A32*DU2 + A33*DU3  
$ + SIG*(U3(KX+1,KY,NL1) - 2.*U3(KX,KY,NL1) + U3(KX-1,KY,NL1))  

423  CONTINUE  
.
.
.

The values of NL1 and NL2 are swapped before the next pass through loop.
CASE 2

DO 424 KX = 2,3

CDIRS IVDEP

DO 424 KY = 2,21

DU1 = U1(KX,KY + 1,NL1) - U1(KX,KY - 1,NL1)
DU2 = U2(KX,KY + 1,NL1) - U2(KX,KY - 1,NL1)
DU3 = U3(KC,KY + 1,NL1) - U3(KX,KY - 1,NL1)

U1(KX,KY,NL2) = U1(KX,KY,NL1) + A11*DU1+A12*DU2+A13*DU3

$ +SIG*(U1(KX+1,KY,NL1)-2.*U1(KX,KY,NL1)+U1(KX-1,KY,NL1))

U2(KX,KY,NL2) = U2(KX,KY,NL1) + A21*DU1+A22*DU2+A23*DU3

$ +SIG*(U2(KX+1,KY,NL1)-2.*U2(KX,KY,NL1)+U2(KX-1,KY,NL1))

U3(KX,KY,NL2) = U3(KX,KY,NL1) + A31*DU1+A32*DU2+A33*DU3

$ +SIG*(U3(KX+1,KY,NL1)-2.*U3(KX,KY,NL1)+U3(KX-1,KY,NL1))

424 CONTINUE

I hope these examples shed some light on this rather abstruse topic.
SECTION 5
IRREGULAR ADDRESSING

Irregular or nonlinear addressing arises in situations such as those using data structures requiring subscripted subscripts. Subscripted subscripts do not occur explicitly in FORTRAN-66 code but may effectively occur in certain types of programs as below:

In the DO 51 loop, Y essentially has a subscripted subscript

\[
\begin{align*}
\text{DO 51 I} & = 1,1000 \\
J & = \text{INDEX (I)} \\
X(I) & = Y(J) \\
\text{51} & \text{ CONTINUE}
\end{align*}
\]

Change to:

\[
\begin{align*}
\text{DO 52 I} & = 1,1000 \\
J & = \text{INDEX (I)} \\
\text{52} & \text{ TEMP(I)} = Y(J) \\
\text{DO 53 I} & = 1,1000 \\
\text{53} & \text{ X(I)} = \text{TEMP(I)} \\
\end{align*}
\]

The DO 51 loop cannot vectorize with CFT 1.06 because of the nonlinear indexing. The DO 52 loop similarly doesn't vectorize but the DO 53 loop does and, if the computations are extensive, the speed-up can be dramatic.

In general, if the computations are sufficiently complicated to warrant the work, you can restructure the loop into two or three loops. The first new loop is a GATHER loop in which all the data to be manipulated are collected into vectors. Next is the computation loop. Then is the SCATTER loop, in which results are distributed from the vector used in the computation loop to their proper locations. Quite often, as in this example, there is no SCATTER loop. There are $SCILIB$ routines for doing the GATHER and SCATTER (see Appendix A and CRI Manual 2240014).

The following example illustrates this again for a particle pushing algorithm.
CASE 1

DO 54 K = 1,150
IX = GRD(K)
XI = IX
VX(K) = VX(K) + EX(IX) + (XX(K) - XI) * DEX(IX)
XX(K) = XX(K) + VX(K) + FLX

C IX IS, IN EFFECT, A SUBSCRIPTED SUBSCRIPT
IR = XX(K)
RI = IR
RX1 = XX(K) - RI
IR = IR - (IR/64) * 64
XX(K) = RI + RX1

C IR IS AN IRREGULAR SUBSCRIPT
RH(IR) = RH(IR) + 1.0 - RX1
RH(IR + 1) = RH(IR + 1) + RX1

54 CONTINUE
CASE 2

DO 55 K = 1,150
IX = GRD(K)
XIV(K) = IX
EXC(K) = EX(IX)
DEXC(K) = DEX(IX)
55 CONTINUE

C GATHER LOOP ABOVE
C XI IS VECTORIZED INTO XIV
C EX IS GATHERED INTO EXC
C DEX IS GATHERED INTO DEXC
DO 56 K = 1,150
VX(K) = VX(K) + EXC(K) + (XX(K) - XIV(K)) * DEXC(K)
XX(K) = XX(K) + VX(K) + FLX
IRV(K) = XX(K)
RI = IRV(K)
RXLV(K) = XX(K) - RI
XX(K) = RI + RXLV(K)
56 CONTINUE

C COMPUTATION LOOP WHICH VECTORIZES IS ABOVE
DO 57 K = 1,150
RH(IRV(K)) = RH(IRV(K)) + 1.0 - RXLV(K)
RH(IRV(K) + 1) = RH(IRV(K) + 1) + RXLV(K)
57 CONTINUE

C SCATTER LOOP

The code in Case 2 runs more than twice as fast as that in Case 1.
SECTION 6
MISCELLANEOUS TECHNIQUES

This section includes a group of examples that do not readily fit into the categories discussed above. In some sense, this is a bag-of-tricks chapter demonstrating several additional loop restructuring techniques as well as all multi-loop techniques. The techniques here are harder to describe in a general and systematic fashion.

A matrix multiply represents an algorithm that can benefit from loop restructuring. For example, the following code illustrates the common way of coding the matrix multiply:

```
DO 61 I = 1,L
DO 61 J = 1,M
C(I,J) = 0.0
DO 61 K = 1,N
61 C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

The recursion on C(I,J) and loop collapsing prevent vectorization now (CFT 1.06) and will always prevent as full vectorization as the rewrite below. This rewritten code vectorizes fully, resulting in a speedup of 5 to 10 times:

```
DO 63 I = 1,L
DO 62 J = 1,M
62 C(I,J) = 0
DO 63 K = 1,N
DO 63 J = 1,M
63 C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

In many similar situations, although the result is not going into a subscripted variable but into a scalar temporary you can reorder the loops and store the results as a vector temporary instead of as a scalar temporary.
The next example shows several stages in the speed-up process. Case 2 is more than 50% faster than Case 1 and Case 3 is almost four times as fast as Case 1.

**CASE 1**

\[ Q = 0.0 \]

\[ \text{DO 64 K = 1,996,5} \]

\[ Q = Q + Z(K) \ast X(K) + Z(K + 1) \ast X(K + 1) \]
\[ + Z(K + 2) \ast X(K + 2) + Z(K + 3) \ast X(K + 3) \]
\[ + Z(K + 4) \ast X(K + 4) \]

64 \hspace{1cm} \text{CONTINUE}

In this original case, the loop was quintupled, presumably to cut loop overhead or allow greater overlap of operations.

**CASE 2**

\[ \text{DO 65 K = 1,996,5} \]

\[ TP(K) = Z(K) \ast X(K) + Z(K + 1) \ast X(K + 1) \]
\[ + Z(K + 2) \ast Z(K + 2) + Z(K + 3) \ast X(K + 3) \]
\[ + Z(K + 4) \ast X(K + 4) \]

65 \hspace{1cm} \text{CONTINUE}

\[ Q = 0.0 \]

\[ \text{DO 66 K = 1,996,5} \]

\[ Q = Q + TP(K) \]

66 \hspace{1cm} \text{CONTINUE}
CASE 3

\[ Q = SDOT(1000, Z, 1, X, 1) \]

Here, SDOT is the BLA single-precision dot function (see Appendix A or CRI publication 2240204).

As an aid to remembering the calling sequences for the basic linear algebra functions, the first argument is the vector length, the remaining arguments are in pairs: a vector operand followed by its increment in memory.

Thus, if \( A \) and \( B \) are declared \( DIMENSION \ A(M, N), B(N, L) \) and you want to compute the dot product of the \( I \)th row of \( A \) with the \( J \)th column of \( B \), use:

\[ AB = SDOT(N, A(I, 1), M, B(1, J), 1) \]

where \( N \) = the vector length = number of elements in each vector operand, \( A(I, 1) \) and \( B(1, J) \) are the starting locations in memory of the operands, \( M \) = memory increment of the first operand vector and \( 1 \) = memory increment of the second operand vector.

Appendix A lists the BLA subroutines briefly as well as a few other useful routines that are in \$SCILIB.

A planned enhancement to CFT is to perform scalar operations for individual statements in otherwise vectorizable loops. An industrious programmer can achieve this now by using \textsc{VFUNCT}s:

\texttt{CDIR \$ VFUNCTION ...}

which tells the compiler of external non-library vector functions. For CFT 1.06, these can be written only in \textsc{CAL}. The CFT manual sections 5 and Appendix F provide the information necessary to link such routines to \textsc{FORTRAN} programs.
CFT 1.06 does not vectorize code blocks that contain IF statements. Many types of loops with IF statements are not hopeless, however. Several things can be done depending on the structure of the code. CFT will eventually vectorize many of these for you but in the meantime you can help by using some of the built-in functions such as AMAX1, ABS, CVMGT, CVMGZ, ...etc. (See CFT manual appendixes B and C). For example:

```
DO 71 I = 1,1000
   IF(A(I) .LT. 0.) A(I) = 0.
71       B(I) = SQRT (A(I)) ...
```

which can be converted to:

```
DO 72 I = 1,1000
   A(I) = AMAX1(A(I), 0.)
72       B(I) = SQRT (A(I)) ...
```

The DO 71 loop doesn't vectorize now; the DO 72 loop does.

All the built-in arithmetic functions in FORTRAN (in $FTLIB) have both vector and scalar versions; the compiler calls the vector version for vectorizable loops*. The vector merge operations, CVMG*, are typeless functions that allow you to merge the results of different vector computations such as the following:

```
DO 74 I = 1,1000
   IF(A(I) .LT. 0.) GOTO 73
   B(I) = A(I) + D(I) ...
   GOTO 74
73      B(I) = A(I) * E(I) ...
74      CONTINUE
```

This can be rewritten to vectorize as

```
DO 75 I = 1,1000
   B(I) = CVMGT (A(I) * E(I) ..., (A(I) + D(I) ..., A(I).LT.0)
```

*Some are actually pseudo vector routines; they allow the loop to vectorize but are performed in scalar mode.
The mnemonic for the CVMG* group of functions is that the last letter of the name is the condition on which the first argument is used. Since these functions are Boolean, they can be used with integer or floating operands and results and in scalar loops as well as in vector loops. Thus, if you are not sure that you are computing the value of B correctly, you can put a print statement in the loop, which causes it to be scalar, and still obtain the same results, albeit much slower than before.

Table 2 lists the merge functions and some of the other ones that you may want in similar situations.

<table>
<thead>
<tr>
<th>FUNCTION NAME</th>
<th>RESULT TYPE</th>
<th>ARGUMENT TYPES</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMAX0(X₁,X₂...)</td>
<td>Real</td>
<td>Real</td>
<td>Largest X₁</td>
</tr>
<tr>
<td>AMAX1(I₁,I₂...)</td>
<td>Real</td>
<td>Integer</td>
<td>Largest I₁, floated</td>
</tr>
<tr>
<td>MAX0(X₁,X₂...)</td>
<td>Integer</td>
<td>Real</td>
<td>Largest X₁, truncated</td>
</tr>
<tr>
<td>MAX1(I₁,I₂...)</td>
<td>Integer</td>
<td>Integer</td>
<td>Largest I₁</td>
</tr>
<tr>
<td>CVMG(T</td>
<td>X,Y,L)</td>
<td>Boolean (single word)</td>
<td>Boolean (single word)</td>
</tr>
<tr>
<td>CVMGZ(X,Y,Z)</td>
<td>Boolean (single word)</td>
<td>Boolean (single word)</td>
<td>X if Z is zero, Y if Z is nonzero</td>
</tr>
</tbody>
</table>

Another technique that works in some cases is inverting the order of loops so that the IF statements are in the outer loops rather than in the inner loops. Also, if the purpose of the IF test is to separate an exceptional case from other cases and if the computation is extensive, it may be worthwhile to write a loop to do the testing and to write a vectorizing loop for the computations.

Here are some more examples:

```
  Y(I) = 1.0
  IF(X(I).EQ.0.) GOTO 76
  Y(I) = 1.0/X(I)
76    CONTINUE
```

Change this to:

```
  Y(I) = 1.0/CVMGZ (1.,X(I),X(I))...
```
which allows a loop containing it to vectorize and yet does not cause a divide fault. Here, CVMGZ selects 1 when X(I) = 0; otherwise it selects X(I). Alternatively, if this exceptional condition only occurs in cases when the result is not used, you can surround the loop containing it with CALL CLEARFI and CALL SETFI to turn the floating point interrupt off and then on again. This allows generation of an infinity without interrupting the program.

The next example illustrates loop reordering and IF statement removal:

CASE 1

DO 77 K = 1,3
FR(K) = 0
77 CONTINUE
DO 79 JA = 1,500
IF (JA .EQ IA) GOTO 79
DS = 0
DO 78 K = 1,3
A(K) = RS(K,IA) - RS(K,JA)
DS = DS + A(K) ** 2
78 CONTINUE
DS = SQRT(DS)
IF (DS .GT. RAD) GOTO 79

79 CONTINUE

CASE 2

DO 710 JA = 1,500
DSV(JA) = 0
C DSV IS A VECTOR OF DS VALUES
710 CONTINUE
DO 712 K = 1,3
711 DSV(JA) = A(K) ** 2
712 CONTINUE
DO 713 JA = 1,500
DSV(JA) = SQRT(DSV(JA))
713 CONTINUE
DO 714 JA = 1,500
IF (DSV(JA) .GT. RAD) GOTO 714

714 CONTINUE

Some of the loops in Case 1 vectorize but this vector length is only 3. In Case 2, the inner loops vectorize with a vector length of 500. In particular, the 713 loop uses a vector square root saving a great deal of time. As it turns out, in the "real life" example, most of the time DS was greater than RAD (last statement shown in the loop) so the rest of the loop did not need any work. Even though additional vectorization could be done, it would not have been very productive. With the change illustrated, the entire kernel ran more than four times faster than the original.
APPENDIX A

$SCILIB$ SUBROUTINES

This appendix summarizes the scientific library subroutines. For a current and more complete description of these functions, refer to the Library Subroutine Reference Manual, CRI Publication 2240014.

LEGEND:

N  Vector length
X,Y Floating point vectors
IX,IY Increments in memory of floating point vectors
C,D Complex vectors
IC,ID Increments in memory of complex vectors
NB Number of bits per word selected for PACK/UNPACK
NW Number of words in unpacked array.

<table>
<thead>
<tr>
<th>Name(Parameters)</th>
<th>Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISAMAX(N,X,IX)</td>
<td>Integer function</td>
<td>Index to real array element having maximum absolute value</td>
</tr>
<tr>
<td>ICAMAX(N,C,IC)</td>
<td>Integer function</td>
<td>Index to complex array element having maximum modulus.</td>
</tr>
<tr>
<td>SASUM(N,X,IX)</td>
<td>Real function</td>
<td>Sums the absolute value of a real array</td>
</tr>
<tr>
<td>SCASUM(N,C,IC)</td>
<td>Real function</td>
<td>Sums the absolute values of real and imaginary parts of complex array</td>
</tr>
<tr>
<td>SAXPY(N,X,IX,Y,IY)</td>
<td>Subroutine</td>
<td>Performs vector computations $y\leftarrow ax+y$ on real arrays, x,y.</td>
</tr>
<tr>
<td>CAXPY(N,C,IC,D,ID)</td>
<td>Subroutine</td>
<td>Performs vector computation $y\leftarrow ax+y$ in complex arrays x,y.</td>
</tr>
<tr>
<td>SCOPLY(N,X,IX,Y,IY)</td>
<td>Subroutine</td>
<td>Copies real array x into real array y.</td>
</tr>
<tr>
<td>CCOPY(N,C,IC,D,ID)</td>
<td>Subroutine</td>
<td>Copies complex array c into complex array d.</td>
</tr>
<tr>
<td>Name (Parameters)</td>
<td>Type</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>SDOT(N,X,IX,Y,IY)</td>
<td>Real function</td>
<td>DOT product of real arrays x,y.</td>
</tr>
<tr>
<td>CDOTC(N,C,IC,D,ID)</td>
<td>Complex function</td>
<td>DOT product of complex arrays c,d.</td>
</tr>
<tr>
<td>CDOTU(N,C,IC,D,ID)</td>
<td>Complex function</td>
<td>DOT product of complex arrays c,d.</td>
</tr>
<tr>
<td>SNRM2(N,X,IX,Y,IY)</td>
<td>Real function</td>
<td>Euclidean norm of real array x.</td>
</tr>
<tr>
<td>SCNRM2(N,C,IC)</td>
<td>Real function</td>
<td>Euclidean norm of complex array c.</td>
</tr>
<tr>
<td>SROT(N,X,IX,Y,IY)</td>
<td>Subroutine</td>
<td>Performs Givens transformation on real arrays x,y.</td>
</tr>
<tr>
<td>SROTG(...)</td>
<td>Subroutine</td>
<td>Calculates parameters for SROTX.</td>
</tr>
<tr>
<td>SROTM(...)</td>
<td>Subroutine</td>
<td>Modified Givens transformation.</td>
</tr>
<tr>
<td>SROTMG(...)</td>
<td>Subroutine</td>
<td>Sets up parameters for modified Givens.</td>
</tr>
<tr>
<td>SSCAL(N,A,X,IX)</td>
<td>Subroutine</td>
<td>Rescales real array x: x&lt;-&gt;ax, a real.</td>
</tr>
<tr>
<td>CSSCAL(N,A,C,IC)</td>
<td>Subroutine</td>
<td>Rescales complex array c: c&lt;-&gt;ac, with a real.</td>
</tr>
<tr>
<td>CSCAL(N,A,C,IC)</td>
<td>Subroutine</td>
<td>Rescales complex array c: c&lt;-&gt;ac, with a complex.</td>
</tr>
<tr>
<td>SSWAP(N,X,IX,Y,IY)</td>
<td>Subroutine</td>
<td>Swaps real arrays x,y.</td>
</tr>
<tr>
<td>CSWP(N,C,IC,D,ID)</td>
<td>Subroutine</td>
<td>Swaps complex arrays c,d.</td>
</tr>
<tr>
<td>MXMA(...)</td>
<td>Subroutine</td>
<td>Completely general matrix multiply.</td>
</tr>
<tr>
<td>CFFT2(...)</td>
<td>Subroutine</td>
<td>Fourier transforms binary radix complex array.</td>
</tr>
<tr>
<td>RCFPT2(...)</td>
<td>Subroutine</td>
<td>Fourier transforms binary radix real to complex.</td>
</tr>
<tr>
<td>CRFFT2(...)</td>
<td>Subroutine</td>
<td>Fourier transforms binary radix complex array to real.</td>
</tr>
<tr>
<td>Name (Parameters)</td>
<td>Type</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PACK(P,NB,U,NW)</td>
<td>Subroutine</td>
<td>Packs power of 2 bit partial word lists.</td>
</tr>
<tr>
<td>UNPACK(P,NB,U,NW)</td>
<td>Subroutine</td>
<td>Unpacks list into power of 2 bit partial words.</td>
</tr>
<tr>
<td>MINV(...)</td>
<td>Subroutine</td>
<td>Returns solution of general linear equation set, matrix inverse optional.</td>
</tr>
<tr>
<td>SSUM(N,C,IC,D,ID)</td>
<td>Real function</td>
<td>Sums the elements of a real array.</td>
</tr>
<tr>
<td>CSUM(N,C,IC)</td>
<td>Complex function</td>
<td>Sums the elements of a complex array.</td>
</tr>
<tr>
<td>CROT(N,X,IX,Y,IY)</td>
<td>Subroutine</td>
<td>Applies complex Givens rotation.</td>
</tr>
<tr>
<td>CROTG(N,C,IC,D,ID)</td>
<td>Subroutine</td>
<td>Sets up rotational parameters for CROT.</td>
</tr>
<tr>
<td>FILTERG(...)</td>
<td>Subroutine</td>
<td>Performs general filtering and auto-correlation.</td>
</tr>
<tr>
<td>FILTERS(...)</td>
<td>Subroutine</td>
<td>Calculates symmetric filter coefficient.</td>
</tr>
<tr>
<td>OPFLIT(...)</td>
<td>Subroutine</td>
<td>Wiener-Levinson equation solver.</td>
</tr>
</tbody>
</table>
APPENDIX B

FORTRAN DIALECTICAL DIFFERENCES

As an aid for conversions, this appendix contains a number of tables that compare the FORTRAN compiler dialects for several manufacturers. The following tables are included.

1. Hardware dependencies
2. Coding features
3. Declaratives and ordering
4. Names and variables
5. Constants, literals, and strings
6. Arithmetic and expressions
7. Branching and control statements
8. Input/output formatting
9. Subroutines and functions
10. Intrinsic or inline functions
11. External functions.

All of these tables are based on a scan of manuals and are thus prone to error and very prone to omissions. Further, as manufacturers bring their FORTRAN dialects into conformance with FORTRAN X3.9-1978, some of these differences can be expected to disappear. The tables also reflect 1975-1979 versions of FORTRAN. In particular, while CDC FTN 5 is to be an ANSI-1978 version here. CDC means either FTN 4 or RUN FORTRAN. IBM means FORTRAN H. Univac means either FORTRAN V or ASCII FORTRAN. ICL means either 1900 or 2900 FORTRAN. The tables do not generally include any FORTRAN statement, syntax, or peculiarity where CFT is believed to be at least as general as the other dialects shown. "Unusual" and its synonyms mean "non-CFT."
<table>
<thead>
<tr>
<th>Item</th>
<th>CTP</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>DEC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of characters per word, bits per character</td>
<td>8,8</td>
<td>Unspecified</td>
<td>Unspecified</td>
<td>10,6</td>
<td>4,8</td>
<td>A: 4,8</td>
<td>2000: 4,8</td>
<td>False: 4,8</td>
</tr>
<tr>
<td>Character manipulation in CTP cannot involve more than 8 characters per word. Card images require 10AB format i.e., 10 words to store image.</td>
<td>True/False representation</td>
<td>Negative</td>
<td>Unspecified</td>
<td>Unspecified</td>
<td>FTM: Neg, 8</td>
<td>True: LSB-1</td>
<td>True / 0</td>
<td>True / 0</td>
</tr>
<tr>
<td>Equivalences to logical or binary settings of logicals</td>
<td>Positive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>False: LSB=0</td>
</tr>
<tr>
<td>Collation: letters before or after number, each contiguous. Testing for character sequences; letters greater than numbers internally for CTP.</td>
<td>After, yes</td>
<td>Unspecified</td>
<td>Unspecified</td>
<td>Before, Yes</td>
<td>Before, No</td>
<td>After, Yes</td>
<td>Before, No</td>
<td>After, Yes</td>
</tr>
<tr>
<td>Internal Character Code</td>
<td>ASCII</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ASCII (9 Bit)</td>
</tr>
<tr>
<td>Binary constants</td>
<td>...B</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>FTM...B</td>
<td>x...</td>
<td>O...</td>
<td>x...</td>
<td>O...</td>
</tr>
<tr>
<td>Item</td>
<td>CFT</td>
<td>ANSI-66</td>
<td>ANSI-77</td>
<td>CDC</td>
<td>IBM</td>
<td>UNIVAC</td>
<td>ICL</td>
<td>HONEYWELL</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>-------</td>
<td>-------</td>
<td>--------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Comment indicators (in column 1)</td>
<td>C,*</td>
<td>C</td>
<td>C,*</td>
<td>C;*</td>
<td>C</td>
<td>C;* in line for rest of line</td>
<td>C</td>
<td>C,*</td>
</tr>
<tr>
<td>Continuation allowed (line length, cords)</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>1320 characters total</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>Multiple statements per line, separator is:</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>$</td>
<td>Not allowed</td>
<td>$ for comment</td>
<td>Not allowed</td>
<td>;</td>
</tr>
<tr>
<td>Multiple replacement statement</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Yes</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>PROGRAM statement</td>
<td>Defines program name</td>
<td>Defines program name</td>
<td>Defines files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pseudo functions (functions usable on left or right of *), must be recoded without pseudo functions.</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Name only allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Partially reserved words</td>
<td>FORMAT</td>
<td>None</td>
<td>END</td>
<td>PTN; FORMAT</td>
<td>RUN; CALL, END</td>
<td>O,...</td>
<td>E,...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>END</td>
<td>None</td>
<td>END</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unusual characters allowed</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td></td>
<td></td>
<td>in CALL and ** in CALL and in CALL</td>
<td>in CALL and ** in CALL</td>
<td>in CALL and ** in CALL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$ in name</td>
<td></td>
<td>$ in name</td>
<td>$ in name</td>
<td>$ in name</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>for concatenation</td>
<td>for concatenation</td>
<td>for concatenation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>in MS 1/0</td>
<td>in MS 1/0</td>
<td>in MS 1/0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>; for statement terminator</td>
<td>; for statement terminator</td>
<td>; for statement terminator</td>
</tr>
<tr>
<td>Item</td>
<td>CFT</td>
<td>ANSI-64</td>
<td>ANSI-77</td>
<td>CDC</td>
<td>IBM</td>
<td>UNIVAC</td>
<td>ICL</td>
<td>HONEYWELL</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>------</td>
<td>------</td>
<td>--------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>Data assignable in declarative</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Precision of variables (by &quot;n&quot;) in type statement</td>
<td>Yes</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Yes</td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>&quot;TYPE&quot; type allowed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Non CPT types</td>
<td>None</td>
<td>None</td>
<td>Character</td>
<td>ECS</td>
<td>Character</td>
<td>Character</td>
<td>Character</td>
<td>Abnormal</td>
</tr>
<tr>
<td>DATA(...) allowed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Arithmetic statement functions can be other than after declarative</td>
<td>No</td>
<td>Not allowed</td>
<td>No</td>
<td>FTN; No</td>
<td>UNIV; Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>before executable statements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parenthesis required in PARAMETER</td>
<td>Yes</td>
<td>Not allowed</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>IMPLICIT must precede all other declaratives, executables</td>
<td>Yes</td>
<td>Not allowed</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Results of DATA statement type mismatch</td>
<td>Converted except for logical, complex</td>
<td>Converted except for logical character</td>
<td>Runs ignored</td>
<td>Runs ignored</td>
<td>Convert V; Ignore</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For RUN and UNIVAC FORTRAN V must correct types of constants in DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>statements.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMON irregularities; initial common block lengths</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Numbered</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>must be as long as ever required. Numbered common must be changed to</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>named.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 4: Names of Variables

<table>
<thead>
<tr>
<th>Item</th>
<th>CFT</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>CDC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of characters in names</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>$ in name</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Yes</td>
<td>After initial letter</td>
<td>Not allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Lower case characters</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Treated as upper case</td>
<td>Not allowed</td>
<td>Treated as upper case</td>
</tr>
</tbody>
</table>

### Table 5: Constants, Literals, and Strings

<table>
<thead>
<tr>
<th>Item</th>
<th>CFT</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>CDC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non CPT types</td>
<td>None</td>
<td>None</td>
<td>Character</td>
<td>Not allowed</td>
<td>Quad precision complex</td>
<td>Quad precision double-complex</td>
<td>Quadruple double-comp. character</td>
<td>Character</td>
</tr>
<tr>
<td>Double precision changes to single</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>I.J.K,L,M,N,E,R,G,O,D.</td>
</tr>
<tr>
<td>Double precision and quad precision functions must be changed to correspond to actual argument types where these are changed.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unusual constant forms</td>
<td>...B for octal</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>FTM...B</td>
<td>... for Hex</td>
<td>Data initialization only.</td>
<td>D... for octal</td>
<td>19000D... for octal</td>
</tr>
<tr>
<td>Alternate character codes (See also third hardware item)</td>
<td>None</td>
<td>Not Specified</td>
<td>Not Specified</td>
<td>Display code</td>
<td>EBCDIC</td>
<td>V; Field data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 6 ARITHMETIC AND EXPRESSIONS

<table>
<thead>
<tr>
<th>Item</th>
<th>CFT</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>CDC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
</table>
| 1/X X = FLGAT(1/0)/X  
For CDC Subscript (1) can  
be added in arithmetic  
statements for subscripted  
variables if one is not present. | Yes | Not allowed | Yes | No | Yes | Yes | Yes |          |
| In A=B # (and other similar  
ones) | Not allowed | Not allowed | Not allowed | Not allowed | Not allowed | A; No | V; Yes | Not allowed |
| Subscripts required for  
multi-dimensional arrays | At least one | All | All | None | All | All | All |          |
| Non-integer subscripts | Not allowed | Not allowed | Not allowed | FTC; Yes | Yes; real | Yes; real only |          |

### TABLE 7 BRANCHING AND CONTROL STATEMENTS

<table>
<thead>
<tr>
<th>Item</th>
<th>CFT</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>CDC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
</table>
| Results of out-of-range com-  
puted GO TO. | Fall through | Not allowed | Fall through | Fatal error | Fall through | Fall through | Fall through | Fatal error |
| Arithmetic IF can have missing  
statement labels. | Not allowed | Not allowed | Not allowed | Yes, GOTO can  
also  
900; Yes | No | Yes |
| Arithmetic IF can have complex  
argument | Not allowed | Not allowed | Not allowed | Yes; only  
real part  
tested | Not allowed | Not allowed |          |
| DO I = 10,1 executes  
This is a difficult error to  
monitor unless IF's are inserted  
for all DO statement but it can  
be fixed with ON#3 on the CFT statement  
Complex relational operator.  
Complex | No times | Not allowed | No times | Once | Once | As if  
DO I=10,1,-1 | Once |          |
| Labels on non-executable  
non-FORMAT statements | Not allowed | Not allowed | Labels allowed;  
Reference not  
allowed | Not allowed | Not allowed | A; FORMAT | Not allowed | can end DO.  
V; Yes |
<table>
<thead>
<tr>
<th>Item</th>
<th>CPT</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>CDC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free format I/O</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>* for statement label in I/O statement</td>
<td>END*, ERR*, IOSTAT=</td>
<td>END*, ERR*, IOSTAT=</td>
<td>END*, ERR*, IOSTAT=</td>
<td>END*, ERR*, IOSTAT=</td>
<td>Not allowed</td>
</tr>
<tr>
<td>End-of-file, error checks</td>
<td>END*, ERR*, IEOF</td>
<td>END*, ERR*, IOSTAT=</td>
<td>FTH*, EOF, ICHX, RUN, EOF, U</td>
<td>FTH*, No</td>
<td>FTH*, Yes</td>
<td>Not allowed</td>
<td>As</td>
<td>No</td>
</tr>
<tr>
<td>TAPE, I/O TAPE, etc. allowed with R/W statements</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>FTH*, No</td>
<td>FTH*, Yes</td>
<td>Not allowed</td>
<td>As</td>
<td>No</td>
</tr>
<tr>
<td>Random mass storage I/O</td>
<td>GETPOS</td>
<td>None</td>
<td>READ(.REC=)</td>
<td>REAMS/ WRITES etc.</td>
<td>FIND</td>
<td>FIND</td>
<td>READ(u,l,clause)</td>
<td>READ(u,l,clause)</td>
</tr>
<tr>
<td>Other I/O statements</td>
<td>SetPOS</td>
<td>None</td>
<td>WRITE(.REC=)</td>
<td>WRITEC</td>
<td>READ(u,l,DEFIN, etc.</td>
<td>READ(u,l,DEFIN, etc.</td>
<td>READ(u,l,DEFIN, etc.</td>
<td>LIKE IBM</td>
</tr>
<tr>
<td>None CPT FORMAT specs</td>
<td>None</td>
<td>None</td>
<td>EM.dEe,etc.</td>
<td>Er.dEe, etc.</td>
<td>Qw.d</td>
<td>E,F ok for integers; G ok integer, logical</td>
<td>Qw.d, V, Hv.c</td>
<td>G ok for logical or integer</td>
</tr>
<tr>
<td>Format paren nesting maximum</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3+</td>
<td>3</td>
<td>3+</td>
<td>3+</td>
</tr>
<tr>
<td>Encode/Decode peculiarities</td>
<td>None</td>
<td>Not allowed</td>
<td>READ &amp; WRITE to character strings</td>
<td>REREAD</td>
<td>Char count optional; no. of chars converted available. ERR= allowed</td>
<td>Char count not included. ERR= allowed</td>
<td>Char count not included. ERR= allowed</td>
<td>Char count not included. ERR= allowed</td>
</tr>
<tr>
<td>Line</td>
<td>CFT</td>
<td>AMLI-66</td>
<td>AMLI-77</td>
<td>CDC</td>
<td>IBM</td>
<td>UNIVAC</td>
<td>ICL</td>
<td>HONEYWELL</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>In program; RETURN-STOP</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>FTN: Yes/Yes</td>
<td>Yes</td>
<td>&quot;Internal&quot; subroutines allowed</td>
<td>STOP required</td>
<td></td>
</tr>
<tr>
<td>In subroutine; END = RETURN</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>RUN: Yes/No</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alternate returns syntax</td>
<td>* before</td>
<td>Not allowed</td>
<td>* before</td>
<td>&quot;-&quot;, &quot;...&quot;, 4 before</td>
<td>6 before</td>
<td>6 before</td>
<td>4 before</td>
<td>FORTAN 77</td>
</tr>
<tr>
<td>label in CALL</td>
<td>label in CALL</td>
<td>returns[&quot;-&quot;, &quot;-&quot;, label in CALL</td>
<td>6 in subr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>* in subr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENTRY has own calling sequence and usable as a function</td>
<td>Yes</td>
<td>Not allowed</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CDC ENTRY statements must have correct calling sequences added.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>END statement required</td>
<td>Yes</td>
<td>Illegal</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Dummy argument in slashes = call by address</td>
<td></td>
<td>Illegal</td>
<td>Illegal</td>
<td>Illegal</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;A subprogram name&quot; allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Yes</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td></td>
</tr>
<tr>
<td>In EXTERNAL only</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overlay syntax</td>
<td>LDR commands</td>
<td>Unspecified</td>
<td>Unspecified</td>
<td>OVERLAY[...]</td>
<td></td>
<td>Use BANK statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROOT, POOL,</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOVL, CALL overlay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(...).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFINE used in arithmetic statement functions</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Allowed</td>
<td>Not allowed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### TABLE 10 INTRINSIC OR INLINE FUNCTIONS

<table>
<thead>
<tr>
<th>Item</th>
<th>CFT</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>CDC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory management</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>EFS</td>
<td>None</td>
<td>EFS</td>
<td></td>
<td>BANK</td>
</tr>
<tr>
<td>1/O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler directives</td>
<td>CDIRS; LIST</td>
<td>Not specified</td>
<td></td>
<td>FTN; No RUN; between routines, . in column 1</td>
<td></td>
<td></td>
<td></td>
<td>Compiler</td>
</tr>
<tr>
<td>Assembler code with FORTRAN</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td>IDENT</td>
<td>Not allowed</td>
<td>Not allowed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code insertion</td>
<td>Via UPDATE</td>
<td>Not allowed</td>
<td>Via UPDATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actual functions that are unusual</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>SHIFT RUN; FORTRAN II syntax</td>
<td>All double precision</td>
<td>All double precision</td>
<td>Double precision + D,E,Q,R,I,J,K prefixes; fatal functions; many additional such as trig with degrees.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 11 EXTERNAL FUNCTIONS

<table>
<thead>
<tr>
<th>Item</th>
<th>CFT</th>
<th>ANSI-66</th>
<th>ANSI-77</th>
<th>CDC</th>
<th>IBM</th>
<th>UNIVAC</th>
<th>ICL</th>
<th>HONEYWELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>General differences</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>All double precision</td>
<td>All double precision</td>
<td>All double precision</td>
<td>GAMMA</td>
<td>LIAGAMA</td>
</tr>
<tr>
<td>Specific functions</td>
<td></td>
<td></td>
<td></td>
<td>FORTRAN II functions; LOCIF,ABS, etc</td>
<td>SLITE</td>
<td>DISPLA</td>
<td>TIME</td>
<td>IOCHREC</td>
</tr>
</tbody>
</table>
APPENDIX C

TRACEBACK

There is a FORTRAN library routine TRBK (file) that you can call to determine how the program reached the current routine. If there is no argument, the trace is printed in the logfile, if file is specified, "$OUT" perhaps, the traceback goes to that file.

When a task terminates abnormally, TRBK is automatically called so a trace of the subroutine calling tree to the point of error is provided. To make this as useful as possible, turn on the block listing from CFT:

CFT,ON=B, ...

and the load map on the loader card: LDR,MAP,... so that the addresses provided can be easily localized in the FORTRAN source. Below is an annotated abort message to illustrate using the trace information.

AB053 - FLOATING POINT ERROR

AB000 - JOB STEP ABORTED.  P = 0144760A

- ******** WAS CALLED BY SOLVE AT LOCATION 0144760A
- SOLVE WAS CALLED BY EQNS AT LOCATION 0012341C
- EQNS WAS CALLED BY $MAIN AT LOCATION 0003411A

The cause of termination is a floating point overflow from the first line of the abort message. Another common diagnostic is "OPERAND RANGE ERROR", which occurs when an attempt is made to reference some part of memory outside your user area, most likely a wild index. If all output is lost, one possible cause is using block common with a DIMENSION of 1, COMMON X(1), but storing into Xs with subscripts much larger than 1. (This may overwrite I/O buffers and tables.)

The last instruction being executed at the time of abort was at location 00144760A. For the FORTRAN programmer, discard the parcel, A in this case, leaving the OCTAL address of the instruction word. The actual error probably occurred on earlier instruction. The load map ADDRESS gives the base address of all routines. In this case, find the base address of SOLVE, where the error occurred (from the third line of the abort message). Subtract the base address from the absolute P-counter address given to find the relative address in SOLVE. (Remember to subtract in OCTAL!!!)
Because the BLOCK (GN=B) listing was turned on for CFT, you will have a list of all blocks which will allow you to find the FORTRAN code block where the abort occurred:

- P-Counter - 0144760 (A)
- Loader ADDRESS of SOLVE - 0104700
- Relative address in SOLVE - 0040060

Partial block listing for SOLVE:

```
  ...
  ...
  SOLVE VECTOR BLOCK BEGINS AT SEQ. NO. 1372, P=40035B
  SOLVE BLOCK BEGINS AT SEQ. NO. 1380, P=40055D
  SOLVE BLOCK BEGINS AT SEQ NO. 1401, P=40077A
  ...
  ...
  ```

Because the relative error address is 40060, which is between 40055 and 40077. The bomb occurred for a FORTRAN statement between numbers 1380 and 1401. The listing should now help you find the probable source of the error quickly.

Similarly, the point at which SOLVE was called by EQNS can be determined. The absolute address of the CALL was 0012341C and EQNS was called by the main program at absolute locations 0003411A.
APPENDIX D
COMPLIER DIRECTIVES

Compiler directive lines begin with characters CDIR$ in columns 1 through 5 and any of the directives listed below in columns 7 through 72.

<table>
<thead>
<tr>
<th>DIRECTIVE</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJECT</td>
<td>Ejects to top of next page.</td>
</tr>
<tr>
<td>LIST</td>
<td>Resumes listable output.</td>
</tr>
<tr>
<td>NOLIST</td>
<td>Suppresses production of listable output.</td>
</tr>
<tr>
<td>CODE</td>
<td>Produces code list.</td>
</tr>
<tr>
<td>NOCODE</td>
<td>Suppresses production of CPT-generated code lists.</td>
</tr>
<tr>
<td>VECTOR</td>
<td>Enables vectorization of inner DO-loops.</td>
</tr>
<tr>
<td>NOVECTOR</td>
<td>Suppresses vectorization of inner DO-loops.</td>
</tr>
<tr>
<td>IVDEP</td>
<td>Ignores vector dependencies in the next DO-loops.</td>
</tr>
<tr>
<td>INT24</td>
<td>Identifies listed variables and arrays as 24-bit integers, equivalent to INTEGER *2 declarative.</td>
</tr>
<tr>
<td>FLOW</td>
<td>Enables flowtrace.</td>
</tr>
<tr>
<td>NOFLOW</td>
<td>Disables flowtrace.</td>
</tr>
<tr>
<td>SHED</td>
<td>Enables the scheduler.</td>
</tr>
<tr>
<td>NOSCH</td>
<td>Disables the scheduler.</td>
</tr>
<tr>
<td>VFUNCTION</td>
<td>Identifies external vector functions.</td>
</tr>
<tr>
<td>BOUNDS</td>
<td>Checks array references for out-of-hand subscripts.</td>
</tr>
</tbody>
</table>
### APPENDIX E

**CHARACTER SETS**

<table>
<thead>
<tr>
<th>CHAR</th>
<th>ASCII</th>
<th>HEX</th>
<th>ASCII CARD CODE</th>
<th>CHAR</th>
<th>ASCII</th>
<th>HEX</th>
<th>ASCII CARD CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>000</td>
<td>00</td>
<td>12-0-9-8-1</td>
<td>@</td>
<td>100</td>
<td>40</td>
<td>8-4</td>
</tr>
<tr>
<td>SOH</td>
<td>001</td>
<td>01</td>
<td>12-9-1</td>
<td>A</td>
<td>101</td>
<td>41</td>
<td>12-1</td>
</tr>
<tr>
<td>STX</td>
<td>002</td>
<td>02</td>
<td>12-9-2</td>
<td>B</td>
<td>102</td>
<td>42</td>
<td>12-2</td>
</tr>
<tr>
<td>ETX</td>
<td>003</td>
<td>03</td>
<td>12-9-3</td>
<td>C</td>
<td>103</td>
<td>43</td>
<td>12-3</td>
</tr>
<tr>
<td>EOT</td>
<td>004</td>
<td>04</td>
<td>9-7</td>
<td>D</td>
<td>104</td>
<td>44</td>
<td>12-4</td>
</tr>
<tr>
<td>ENQ</td>
<td>005</td>
<td>05</td>
<td>0-9-8-5</td>
<td>E</td>
<td>105</td>
<td>45</td>
<td>12-5</td>
</tr>
<tr>
<td>ACK</td>
<td>006</td>
<td>06</td>
<td>0-9-8-6</td>
<td>F</td>
<td>106</td>
<td>46</td>
<td>12-6</td>
</tr>
<tr>
<td>BEL</td>
<td>007</td>
<td>07</td>
<td>0-9-8-7</td>
<td>G</td>
<td>107</td>
<td>47</td>
<td>12-7</td>
</tr>
<tr>
<td>BS</td>
<td>010</td>
<td>08</td>
<td>11-9-6</td>
<td>H</td>
<td>110</td>
<td>48</td>
<td>12-8</td>
</tr>
<tr>
<td>HT</td>
<td>011</td>
<td>09</td>
<td>12-9-5</td>
<td>I</td>
<td>111</td>
<td>49</td>
<td>12-9</td>
</tr>
<tr>
<td>LF</td>
<td>012</td>
<td>0A</td>
<td>0-9-5</td>
<td>J</td>
<td>112</td>
<td>4A</td>
<td>11-1</td>
</tr>
<tr>
<td>VT</td>
<td>013</td>
<td>0B</td>
<td>12-9-8-3</td>
<td>K</td>
<td>113</td>
<td>4B</td>
<td>11-2</td>
</tr>
<tr>
<td>FF</td>
<td>014</td>
<td>0C</td>
<td>12-9-8-4</td>
<td>L</td>
<td>114</td>
<td>4C</td>
<td>11-3</td>
</tr>
<tr>
<td>CR</td>
<td>015</td>
<td>0D</td>
<td>12-9-8-5</td>
<td>M</td>
<td>115</td>
<td>4D</td>
<td>11-4</td>
</tr>
<tr>
<td>SO</td>
<td>016</td>
<td>0E</td>
<td>12-9-8-6</td>
<td>N</td>
<td>116</td>
<td>4E</td>
<td>11-5</td>
</tr>
<tr>
<td>SI</td>
<td>017</td>
<td>0F</td>
<td>12-9-8-7</td>
<td>O</td>
<td>117</td>
<td>4F</td>
<td>11-6</td>
</tr>
<tr>
<td>DLE</td>
<td>020</td>
<td>10</td>
<td>12-11-9-8-1</td>
<td>P</td>
<td>120</td>
<td>50</td>
<td>11-7</td>
</tr>
<tr>
<td>DC1</td>
<td>021</td>
<td>11</td>
<td>11-9-1</td>
<td>Q</td>
<td>121</td>
<td>51</td>
<td>11-8</td>
</tr>
<tr>
<td>DC2</td>
<td>022</td>
<td>12</td>
<td>11-9-2</td>
<td>R</td>
<td>122</td>
<td>52</td>
<td>11-9</td>
</tr>
<tr>
<td>DC3</td>
<td>023</td>
<td>13</td>
<td>11-9-3</td>
<td>S</td>
<td>123</td>
<td>53</td>
<td>0-2</td>
</tr>
<tr>
<td>DC4</td>
<td>024</td>
<td>14</td>
<td>4-8-9</td>
<td>T</td>
<td>124</td>
<td>54</td>
<td>0-3</td>
</tr>
<tr>
<td>NAK</td>
<td>025</td>
<td>15</td>
<td>9-8-5</td>
<td>U</td>
<td>125</td>
<td>55</td>
<td>0-4</td>
</tr>
<tr>
<td>SYN</td>
<td>026</td>
<td>16</td>
<td>9-2</td>
<td>V</td>
<td>126</td>
<td>56</td>
<td>0-5</td>
</tr>
<tr>
<td>EOB</td>
<td>027</td>
<td>17</td>
<td>0-9-6</td>
<td>W</td>
<td>127</td>
<td>57</td>
<td>0-6</td>
</tr>
<tr>
<td>CAN</td>
<td>030</td>
<td>18</td>
<td>11-9-8</td>
<td>X</td>
<td>130</td>
<td>58</td>
<td>0-7</td>
</tr>
<tr>
<td>EM</td>
<td>031</td>
<td>19</td>
<td>11-9-8-1</td>
<td>Y</td>
<td>131</td>
<td>59</td>
<td>0-8</td>
</tr>
<tr>
<td>SUB</td>
<td>032</td>
<td>1A</td>
<td>9-8-7</td>
<td>Z</td>
<td>132</td>
<td>5A</td>
<td>0-9</td>
</tr>
<tr>
<td>ESC</td>
<td>033</td>
<td>1B</td>
<td>0-9-7</td>
<td>[</td>
<td>133</td>
<td>5B</td>
<td>12-8-2</td>
</tr>
<tr>
<td>FS</td>
<td>034</td>
<td>1C</td>
<td>11-9-8-4</td>
<td>\</td>
<td>134</td>
<td>5C</td>
<td>0-8-2</td>
</tr>
<tr>
<td>GS</td>
<td>035</td>
<td>1D</td>
<td>11-9-8-5</td>
<td>]</td>
<td>135</td>
<td>5D</td>
<td>11-8-2</td>
</tr>
<tr>
<td>RS</td>
<td>036</td>
<td>1E</td>
<td>11-9-8-6</td>
<td>^</td>
<td>136</td>
<td>5E</td>
<td>11-8-7</td>
</tr>
<tr>
<td>US</td>
<td>037</td>
<td>1F</td>
<td>11-9-8-7</td>
<td>_</td>
<td>137</td>
<td>5F</td>
<td>0-8-5</td>
</tr>
<tr>
<td>Space</td>
<td>040</td>
<td>20</td>
<td>None</td>
<td>\</td>
<td>140</td>
<td>60</td>
<td>8-1</td>
</tr>
<tr>
<td>CHAR</td>
<td>ASCII</td>
<td>HEX</td>
<td>ASCII CARD CODE</td>
<td>CHAR</td>
<td>ASCII</td>
<td>HEX</td>
<td>ASCII CARD CODE</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-----</td>
<td>----------------</td>
<td>------</td>
<td>-------</td>
<td>-----</td>
<td>----------------</td>
</tr>
<tr>
<td>:</td>
<td>041</td>
<td>21</td>
<td>12-8-7</td>
<td>a</td>
<td>141</td>
<td>01</td>
<td>12-0-1</td>
</tr>
<tr>
<td>&quot;</td>
<td>042</td>
<td>22</td>
<td>8-7</td>
<td>b</td>
<td>142</td>
<td>02</td>
<td>12-0-2</td>
</tr>
<tr>
<td>#</td>
<td>043</td>
<td>23</td>
<td>8-3</td>
<td>c</td>
<td>143</td>
<td>03</td>
<td>12-0-3</td>
</tr>
<tr>
<td>$</td>
<td>044</td>
<td>24</td>
<td>11-8-3</td>
<td>d</td>
<td>144</td>
<td>04</td>
<td>12-0-4</td>
</tr>
<tr>
<td>&amp;</td>
<td>045</td>
<td>25</td>
<td>0-8-4</td>
<td>e</td>
<td>145</td>
<td>05</td>
<td>12-0-5</td>
</tr>
<tr>
<td>'</td>
<td>046</td>
<td>26</td>
<td>12</td>
<td>f</td>
<td>146</td>
<td>06</td>
<td>12-0-6</td>
</tr>
<tr>
<td>(</td>
<td>047</td>
<td>27</td>
<td>8-5</td>
<td>g</td>
<td>147</td>
<td>07</td>
<td>12-0-7</td>
</tr>
<tr>
<td>)</td>
<td>050</td>
<td>28</td>
<td>12-8-5</td>
<td>h</td>
<td>150</td>
<td>08</td>
<td>12-0-8</td>
</tr>
<tr>
<td>*</td>
<td>051</td>
<td>29</td>
<td>11-8-5</td>
<td>i</td>
<td>151</td>
<td>09</td>
<td>12-0-9</td>
</tr>
<tr>
<td>+</td>
<td>052</td>
<td>2A</td>
<td>11-8-4</td>
<td>j</td>
<td>152</td>
<td>0A</td>
<td>12-1-1</td>
</tr>
<tr>
<td>,</td>
<td>053</td>
<td>2B</td>
<td>12-8-6</td>
<td>k</td>
<td>153</td>
<td>0B</td>
<td>12-1-2</td>
</tr>
<tr>
<td>-</td>
<td>054</td>
<td>2C</td>
<td>0-8-3</td>
<td>l</td>
<td>154</td>
<td>0C</td>
<td>12-1-3</td>
</tr>
<tr>
<td>.</td>
<td>055</td>
<td>2D</td>
<td>11</td>
<td>m</td>
<td>155</td>
<td>0D</td>
<td>12-1-4</td>
</tr>
<tr>
<td>/</td>
<td>056</td>
<td>2E</td>
<td>12-8-3</td>
<td>n</td>
<td>156</td>
<td>0E</td>
<td>12-1-5</td>
</tr>
<tr>
<td>0</td>
<td>057</td>
<td>2F</td>
<td>0-1</td>
<td>o</td>
<td>157</td>
<td>0F</td>
<td>12-1-6</td>
</tr>
<tr>
<td>1</td>
<td>060</td>
<td>30</td>
<td>0</td>
<td>p</td>
<td>160</td>
<td>70</td>
<td>12-1-7</td>
</tr>
<tr>
<td>2</td>
<td>061</td>
<td>31</td>
<td>1</td>
<td>q</td>
<td>161</td>
<td>71</td>
<td>12-1-8</td>
</tr>
<tr>
<td>3</td>
<td>062</td>
<td>32</td>
<td>2</td>
<td>r</td>
<td>162</td>
<td>72</td>
<td>12-1-9</td>
</tr>
<tr>
<td>4</td>
<td>063</td>
<td>33</td>
<td>3</td>
<td>s</td>
<td>163</td>
<td>73</td>
<td>11-0-2</td>
</tr>
<tr>
<td>5</td>
<td>064</td>
<td>34</td>
<td>4</td>
<td>t</td>
<td>164</td>
<td>74</td>
<td>11-0-3</td>
</tr>
<tr>
<td>6</td>
<td>065</td>
<td>35</td>
<td>5</td>
<td>u</td>
<td>165</td>
<td>75</td>
<td>11-0-4</td>
</tr>
<tr>
<td>7</td>
<td>066</td>
<td>36</td>
<td>6</td>
<td>v</td>
<td>166</td>
<td>76</td>
<td>11-0-5</td>
</tr>
<tr>
<td>8</td>
<td>067</td>
<td>37</td>
<td>7</td>
<td>w</td>
<td>167</td>
<td>77</td>
<td>11-0-6</td>
</tr>
<tr>
<td>9</td>
<td>070</td>
<td>38</td>
<td>8</td>
<td>x</td>
<td>170</td>
<td>78</td>
<td>11-0-7</td>
</tr>
<tr>
<td>;</td>
<td>071</td>
<td>39</td>
<td>9</td>
<td>y</td>
<td>171</td>
<td>79</td>
<td>11-0-8</td>
</tr>
<tr>
<td>;</td>
<td>072</td>
<td>3A</td>
<td>8-2</td>
<td>z</td>
<td>172</td>
<td>7A</td>
<td>11-0-9</td>
</tr>
<tr>
<td>&lt;</td>
<td>073</td>
<td>3B</td>
<td>11-8-6</td>
<td>7</td>
<td>173</td>
<td>7B</td>
<td>12-0</td>
</tr>
<tr>
<td>=</td>
<td>074</td>
<td>3C</td>
<td>12-8-4</td>
<td>7</td>
<td>174</td>
<td>7C</td>
<td>12-11</td>
</tr>
<tr>
<td>&gt;</td>
<td>075</td>
<td>3D</td>
<td>8-6</td>
<td>7</td>
<td>175</td>
<td>7D</td>
<td>11-0</td>
</tr>
<tr>
<td>?</td>
<td>076</td>
<td>3E</td>
<td>0-8-6</td>
<td>7</td>
<td>176</td>
<td>7E</td>
<td>11-0-1</td>
</tr>
<tr>
<td>DEL</td>
<td>077</td>
<td>3F</td>
<td>0-8-7</td>
<td></td>
<td></td>
<td></td>
<td>12-9-7</td>
</tr>
</tbody>
</table>