

# FLECS

---

A Structured Fortran Preprocessor  
Processed by T<sub>E</sub>X on 17 December 2009

Terry Beyer  
Computing Center  
University of Oregon  
Eugene, Oregon 97403

Additional Modifications by

Robert E. Bruccoleri  
Department of Macromolecular Modeling  
Bristol-Myers Squibb Pharmaceutical Research Institute  
P.O. Box 4000  
Princeton, N.J. 08543-4000  
Internet: [bruc@bms.com](mailto:bruc@bms.com)

---

This file documents FLECS — A Structured Fortran Preprocessor

Neither the authors nor the University of Oregon shall be liable for any direct or indirect, incidental, consequential, or specific damages of any kind or from any cause whatsoever arising out of or in any way connected with the use or performance of the Flecs system or its documentation.

Copyright 1992 Bristol-Myers Squibb Company

Copyright 1987 Robert E. Bruccoleri

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice and this permission notice and warranty disclaimer appear in all copies.

BRISTOL-MYERS SQUIBB COMPANY AND ROBERT E. BRUCCOLERI DISCLAIM, AND THE USER WAIVES, ALL REPRESENTATIONS AND WARRANTIES, EXPRESS OR IMPLIED, WITH REGARD TO THIS SOFTWARE, INCLUDING WITHOUT LIMITATION ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR USE OR PURPOSE. BRISTOL-MYERS SQUIBB COMPANY AND ROBERT E. BRUCCOLERI MAKE NO REPRESENTATION OR WARRANTY, EXPRESS OR IMPLIED, AS TO WHETHER THE USE OF THIS SOFTWARE INFRINGES ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT OF ANY THIRD PARTY. IN NO EVENT SHALL BRISTOL-MYERS SQUIBB COMPANY OR ROBERT E. BRUCCOLERI BE LIABLE, AND BY USING THIS SOFTWARE USER AGREES THAT BRISTOL-MYERS SQUIBB OR ROBERT E. BRUCCOLERI SHALL NOT BE LIABLE, FOR ANY SPECIAL, DIRECT, INDIRECT, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, OR ANY OTHER DAMAGES OF ANY NATURE WHATSOEVER, THAT MAY BE INCURRED BY THE USER OR ANY THIRD PARTY ARISING OUT OF OR IN CONNECTION WITH ANY USE OF PERFORMANCE OF THIS SOFTWARE, INCLUDING WITHOUT LIMITATION LOST PROFITS OR DAMAGES RESULTING FROM ANY LOSS OR USE OF DATA.

Please communicate any errors, ambiguities, or omissions to the authors. As of April 8, 1991, Robert Bruccoleri was actively using FLECS.

# Table of Contents

<b>Preface by Robert E. Bruccoleri .....</b>	<b>1</b>
<b>Acknowledgements of Terry Beyer .....</b>	<b>3</b>
<b>1 Introduction .....</b>	<b>5</b>
1.1 Retention of Fortran Features .....	5
1.2 Correlation of Flecs and Fortran Sources .....	6
1.3 Structured Statements .....	6
1.4 Indentation, Lines and the Listing .....	8
<b>2 Flecs Statements .....</b>	<b>11</b>
2.1 Control Structures .....	11
2.1.1 Decision Structures .....	11
2.1.1.1 IF .....	11
2.1.1.2 UNLESS .....	12
2.1.1.3 WHEN...ELSE .....	13
2.1.1.4 CONDITIONAL .....	14
2.1.1.5 SELECT .....	15
2.1.2 Loop Structures .....	15
2.1.2.1 DO .....	16
2.1.2.2 WHILE .....	16
2.1.2.3 REPEAT WHILE .....	18
2.1.2.4 UNTIL .....	19
2.1.2.5 REPEAT UNTIL .....	20
2.2 Internal Procedures .....	20
2.3 Translator Directives .....	22
<b>3 Restrictions and Notes .....</b>	<b>25</b>
<b>4 Errors .....</b>	<b>27</b>
4.1 Syntax Errors .....	27
4.2 Context Errors .....	27
4.3 Undetected Errors .....	27
4.4 Other Errors .....	28
<b>5 Procedure for Use .....</b>	<b>29</b>
5.1 Source Preparation .....	29
5.2 Running the Translator .....	29

<b>6</b>	<b>Flecs Implementation</b>	<b>31</b>
6.1	Necessary and Desirable Modifications	31
6.2	System Structure	31
6.3	Character String Conventions	32
6.4	Translation Parameters	32
6.5	Character Processing Subroutines	34
6.5.1	CATNUM (conCATenate NUMber to string)	35
6.5.2	CATSTR (conCATenate STRing to string)	35
6.5.3	CATSUB (conCATenate SUBstring to string)	36
6.5.4	CHTYP (CHaracter TYPE)	36
6.5.5	CPYSTR (CoPY STRing)	37
6.5.6	CPYSUB (CoPY SUBstring)	37
6.5.7	HASH (HASH function)	38
6.5.8	MAKEST (MAKE STRing)	38
6.5.9	PUTNUM (PUT NUMber)	38
6.5.10	STREQ (STRing EQuality)	39
6.5.11	STRLT (STRing Less Than)	39
6.5.12	STRUP (STRing UPpercase)	40
6.5.13	TRIM (TRIM string of blanks)	40
6.6	I/O Interface	41
6.6.1	Files and Devices	41
6.6.2	Classes of Input/Output	41
6.6.2.1	Class <i>Flecs</i>	42
6.6.2.2	Class <i>Fort</i>	43
6.6.2.3	Class <i>List</i>	43
6.6.2.4	Class <i>Err</i>	43
6.6.3	Line Numbers	43
6.7	I/O Subroutines	44
6.7.1	OPENF (OPEN Files)	45
6.7.2	GET (GET input)	47
6.7.3	PUT (PUT out strings)	48
6.7.4	CLOSEF (CLOSE Files)	49
6.7.5	FLSTOP (FLecs STOP)	49
<b>7</b>	<b>Installation and Modification</b>	<b>51</b>
7.1	The Standard Version	51
7.2	The Initial Working Version	52
7.3	Procedure for Creating the Initial Version	53
7.4	Documentation	53
7.5	Desirable Modifications	53
7.5.1	Efficiency	53
7.5.2	Convenience	53
	<b>Index</b>	<b>55</b>

## Preface by Robert E. Bruccoleri

I started using FLECS in 1978 when I wrote the analysis facility in CHARMM, a macromolecular mechanics program.<sup>1</sup> It has been an invaluable tool in my programming work. Robert C. Ladner introduced me to FLECS, and I have been maintaining the translator since then. The current version described here is much easier to port from machine to machine largely because of Terry Beyer's good design, and because Fortran-77 supports character strings. The elegant string library written by Terry is now much simpler.

This manual was scanned from a copy of the typewritten manual. All of the figures were remade using the GnuEmacs picture mode in character form, and the rest of the text was edited into Texinfo form. As much of the original text from Terry's manuals was left intact, but since the scanning process introduced many errors, please inform me of any mistakes.

Finally, I thank Terry Beyer for having written FLECS, and for having put the program in the public domain. It has had an enormous impact in the development of CHARMM and of CONGEN.<sup>2</sup> The spirit of sharing among many programmers is still strong, and this spirit is a way for us to make our society a little bit richer for everyone.

---

<sup>1</sup> B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus, "CHARMM — A Program for Macromolecular Energy, Minimization, and Dynamics Calculations", *J. Comput. Chem.* **4**, 187–217 (1983).

<sup>2</sup> R. E. Bruccoleri, M. Karplus, "Prediction of the Folding of Short Polypeptide Segments by Uniform Conformational Sampling", *Biopolymers* **26**, 137-168 (1987).



## Acknowledgements of Terry Beyer

The author is indebted to many people for assistance of one form or another during the course of this project. Mike Dunlap, Kevin McCoy, and Peter Moulton deserve special thanks for many helpful and fruitful discussions, suggestions, and encouragements. I am grateful to my wife, Kathleen, who assisted in many ways including shielding me from the harsh reality of JCL and 360 Assembly Language. Text preparation was adroitly accomplished by Marva VanNatta, Allyene Tom, Diane Lane, and Kathleen Beyer.

This project was initiated while the author was working under a grant provided by the Office of Scientific and Scholarly Research of the Graduate School at the University of Oregon. Work on the project has also been supported in part by the Department of Computer Science and by the Computing Center of the University of Oregon.





# 1 Introduction

Fortran contains four basic mechanisms for controlling program flow: `CALL/RETURN`, `IF`, `DO`, and various forms of the `GO TO`.

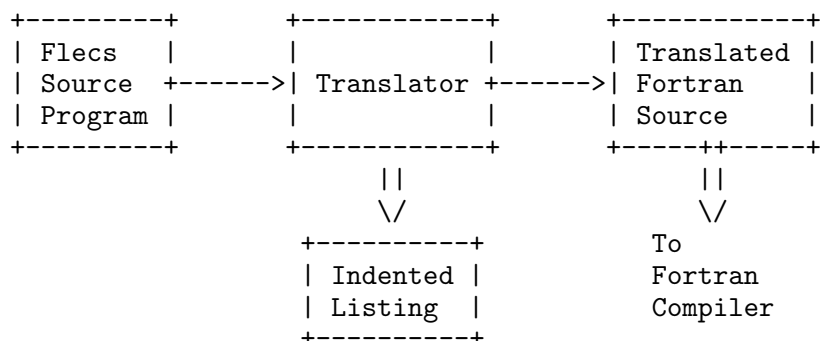
Flecs is a language extension of Fortran which has additional control mechanisms. These mechanisms make it easier to write Fortran by eliminating much of the clerical detail associated with constructing Fortran programs. Flecs is also easier to read and comprehend than Fortran.

This manual is intended to be a brief but complete introduction to Flecs. It is not intended to be a primer on Flecs or structured programming. The reader is assumed to be a knowledgeable Fortran programmer.

For programmers to whom transportability of their programs is a concern, it should be noted that the Flecs translator source code is made freely available. The translator was written with transportability in mind and requires little effort to move from one machine to another.

At Oregon, Flecs was implemented on both the PDP-10 and the IBM S/360. It has since been implemented on a VAX/VMS, Silicon Graphics Iris 4D workstation, Convex, and Cray running Unicos. The manner of implementation is that of a preprocessor which translates Flecs programs into Fortran programs. The resulting Fortran program is then processed in the usual way. The translator also produces a nicely formatted listing of the Flecs program which graphically presents the control structures used.

The following diagram illustrates the translating process.



## 1.1 Retention of Fortran Features

The Flecs translator examines each statement in the Flecs program to see if it is an extended statement (a statement valid in Flecs but not in Fortran). If it is recognized as an extended statement, the translator generates the corresponding Fortran statements. If, however, the statement is not recognized as an extended statement, the translator assumes it must be a Fortran statement and passes it through unaltered. Thus the Flecs system does not restrict the use of Fortran statements, it simply provides a set of additional statements which may be used. In particular, `GO TO`s, arithmetic `IF`s, `CALL`s, arithmetic statement functions, and any other Fortran statements, compiler dependent or otherwise, may be used in a Flecs program.

Flecs ignores alphabetic case in all comparisons, so upper and lower case can be intermixed at will.

Since there are conflicts between the Flecs syntax and Fortran-77 syntax as well as conflicts with Fortran extensions, it is possible to turn off all translation activity within a portion of a program. See Section 2.3 [Translator Directives], page 22, for a description of the `TRANSLATE` variable.

## 1.2 Correlation of Flecs and Fortran Sources

One difficulty of preprocessor systems like Flecs is that error messages which come from the Fortran compiler must be related back to the original Flecs source program. There is no good solution to this problem. Currently, the FLECS listing contains the sequential statement number of the Fortran statements that correspond to FLECS statements. Since many Fortran compilers count lines from the beginning of the file (as opposed to the beginning of a program unit), this is very helpful. For other compilers, the wide availability of multi-window editors such as GNU Emacs make it possible to simultaneously view the Flecs source file with either the Flecs listing or the Fortran translation.

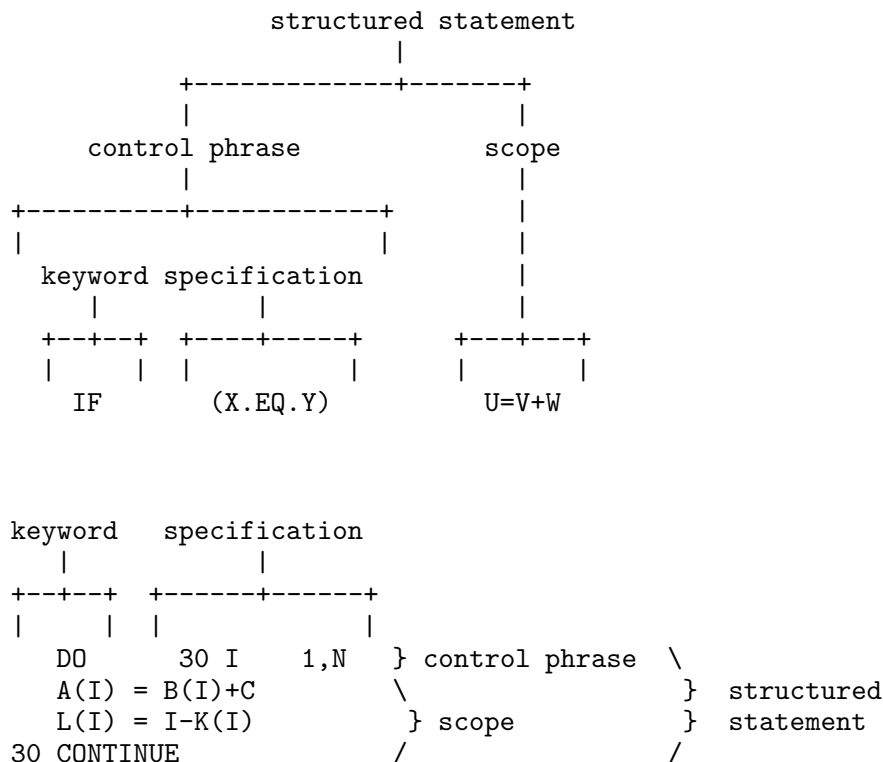
Another possibility is the use of line numbers (not to be confused with Fortran statement numbers) which can be placed on Flecs source statements. These line numbers then appear on the listing. When an error message is produced by either the Flecs translator or the Fortran compiler, it will include the line number of the offending Flecs source statement, making it easy to locate on the listing.

If the programmer chooses not to supply line numbers, the translator will assign sequential numbers and place them on the listing and in the Fortran source. Thus, errors from the compiler may still be related to the Flecs listing.

Details of line numbering are machine dependent and are given in Chapter 5 [Procedure for Use], page 29. On most card oriented systems, the line numbers may be placed in columns 76-80 of each card. Other systems may have special provisions for line numbers.

## 1.3 Structured Statements

A basic notion of Flecs is that of the *structured statement* which consists of a control phrase and its scope. The control phrase is divided into two parts, a keyword followed by some additional information called the specification. Fortran has two structured statements, the logical IF and the DO. The following examples illustrate this terminology:



Note that each structured statement consists of a control phrase which controls the execution of a set of one or more statements called its scope. Also note that each control phrase consists of a keyword plus some additional information called the specification. A statement which does not consist of a control phrase and a scope is said to be a simple statement. Examples of simple statements are assignment statements, subroutine **CALLs**, arithmetic **IFs**, and **GO TOs**.

The problem with the Fortran logical **IF** statement is that its scope may contain only a single simple statement. This restriction is eliminated in the case of the **DO**, but at the cost of clerical detail (having to stop thinking about the problem while a statement number is invented). Note also that the **IF** specification is enclosed in parentheses while the **DO** specification is not.

In **Flecs** there is a uniform convention for writing control phrases and indicating their scopes. To write a structured statement, the keyword is placed on a line beginning in column 7 followed by its specification enclosed in parentheses. The remainder of the line is left blank. The statements comprising the scope are placed on successive lines. The end of the scope is indicated by a **FIN** statement. This creates a multi-line structured statement.

Examples of multi-line structured statements:

```

IF (X.EQ.Y)
  U = V+W
  R = S+T
FIN

DO (I = 1,N)
  A(I) = B(I)+C
  C = C*2.14-3.14
FIN

```

Note: The statement number has been eliminated from the **DO** specification since it is no longer necessary, the end of the loop being specified by the **FIN**.

Nesting of structured statements is permitted to any depth.

Example of nested structured statements:

```

IF (X.EQ.Y)
  U = V+W
  DO (I = 1,N)
    A(I) = B(I)+C
    C = C*2.14-3.14
  FIN
  R = S+T
FIN

```

When the scope of a control phrase consists of a single simple statement, it may be placed on the same line as the control phrase and the **FIN** may be dispensed with. This creates a one-line structured statement.

Examples of one-line structured statements:

```

IF (X.EQ.Y) U = V+W

DO (I = 1,N) A(I) = B(I)+C

```

Since each control phrase must begin on a new line, it is not possible to have a one-line structured statement whose scope consists of a structured statement.

Example of an invalid construction:

```

IF (X.EQ.Y) DO (I = 1,N) A(I) = B(I)+C

```

To achieve the effect desired above, the **IF** must be written in a multi-line form.

Example of a valid construction:

```

IF (X.EQ.Y)
  DO (I = 1,N) A(I) = B(I)+C
  FIN

```

In addition to IF and DO, Flecs provides several useful structured statements not available in Fortran. After a brief excursion into the subject of indentation, we will present these additional structures.

## 1.4 Indentation, Lines and the Listing

In the examples of multi-line structured statements above, the statements in the scope were indented which helps to reveal the structure of the program. The rules for using indentation and FINs are quite simple and uniform. The control phrase of a multi-line structured statement always causes indentation of the statements that follow. Nothing else causes indentation. A level of indentation (i.e. a scope) is always terminated with a FIN. Nothing else terminates a level of indentation.

When writing a Flecs program on paper the programmer should adopt the indentation and line drawing conventions shown below. When preparing a Flecs source program in machine readable form, however, each statement should begin in column 7. When the Flecs translator produces the listing, it will reintroduce the correct indentation and produce the corresponding lines. If the programmer attempts to introduce his own indentation with the use of leading blanks, the program will be translated correctly, but the resulting listing will be improperly indented.

Example of indentation:

1. Program as written on paper by programmer.

```

  IF (X.EQ.Y)
    U = V+W
    DO (I = 1,N)
      A(I) = B(I)+C
      C = C*2.14-3.14
    FIN
  R = S+T
FIN

```

2. Program as entered into computer.

```

  IF (X.EQ.Y)
  U = V+W
  DO (I = 1,N)
  A(I) = B(I)+C
  C = C*2.14-3.14
  FIN
  R = S+T
  FIN

```

3. Program as listed by Flecs translator.

```

  IF (X.EQ.Y)
  .  U = V+W
  .  DO (I = 1,N)
  .    A(I) = B(I)+C
  .    C = C*2.14-3.14
  .    ...FIN
  .  R = S+T
  .  ...FIN

```

The correctly indented listing is a tremendous aid in reading and working with programs. Except for the dots and spaces used for indentation, the lines are listed exactly as they appear in

the source program. That is, the internal spacing of columns 7-72 is preserved. There is seldom any need to refer to a straight listing of the unindented source.

The listing file contains two columns of numbers to the left of the source code. The first number specifies the line number in the FLECS input file. The second number gives the line number of the last statement output to the Fortran output file. It gives a good indication for the Fortran statement that corresponds to the FLECS statement on any given line.

Comment lines are treated in the following way on the listing to prevent interruption of the dotted lines indicating scope. A comment line which contains only blanks in columns 2 through 6 will be listed with columns 7 through 72 indented at the then- current level of indentation as if the line were an executable statement. If, however, one or more non-blank characters appear in columns 2 through 6 of a comment card, it will be listed without indentation. Blank lines may be inserted in the source and will be treated as empty comments.



## 2 Flecs Statements

There are three classes of Flecs statements; control structures, internal procedures, and translator directives. Control structures are statements which implement a particular control flow. Internal procedures provide a mechanism for partitioning a subprogram into smaller pieces. Translator directives control certain aspects of the translation process.

### 2.1 Control Structures

The complete set of control structures provided by Flecs is given below together with their corresponding flow charts. The symbol, **L**, is used to indicate a logical expression. The symbol, **S**, is used to indicate a scope of one or more statements. Some statements, as indicated below, do not have a one-line construction.

A convenient summary of the information in this chapter may be found in Appendix A.

#### 2.1.1 Decision Structures

Decision structures are structured statements which control the execution of their scopes on the basis of a logical expression or test.

##### 2.1.1.1 IF

Description: The IF statement causes a logical expression to be evaluated. If the value is true, the scope is executed once and control passes to the next statement. If the value is false, control passes directly to the next statement without execution of the scope.

General Form:

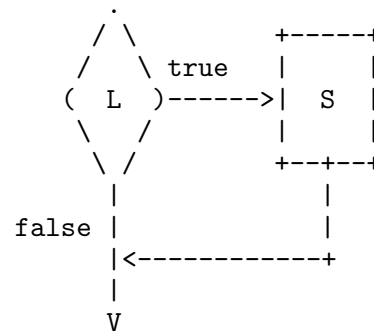
IF (L) S

Examples:

```
IF (X.EQ.Y) U = V+W

IF (T.GT.0 .AND. S.LT.R)
.  I = 1+1
.  Z = 0.1
...FIN
```

Flow Chart:



### 2.1.1.2 UNLESS

Description: UNLESS (L) is functionally equivalent to IF(.NOT.(L)), but is more convenient in some contexts.

General Form:

```
UNLESS (L) S
```

Examples:

```
UNLESS (X.NE.Y) U = V+W
```

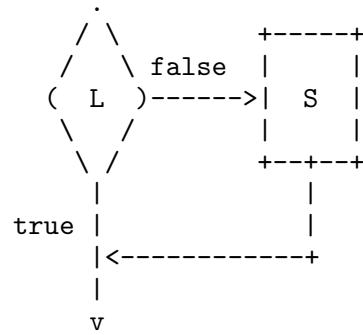
```
UNLESS (T.LE.O.OR.S.GE.R)
```

```
.  I = 1+1
```

```
.  Z = 0.1
```

```
...FIN
```

Flow Chart:





### 2.1.1.3 WHEN...ELSE

Description: The WHEN...ELSE statements correspond to the IF...THEN...ELSE statement of Algol, PL/1, Pascal, etc. In Flecs, both the WHEN and the ELSE act as structured statements although only the WHEN has a specification. The ELSE statement must immediately follow the scope of the WHEN. The specifier of the WHEN is evaluated and exactly one of the two scopes is executed. The scope of the WHEN statement is executed if the expression is true and the scope of the ELSE statement is executed if the expression is false. In either case, control then passes to the next statement following the ELSE statement.

General Form:

```
WHEN (L) S1
ELSE S2
```

Examples:

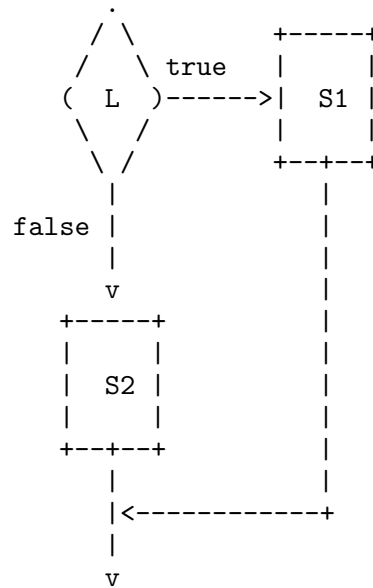
```
WHEN (X.EQ.Y) U = V+W
ELSE U = V-W
```

```
WHEN (X.EQ.Y)
. U = V+W
. T = T+1.5
...FIN
ELSE U = V-W
```

```
WHEN (X.EQ.Y) U = V+W
ELSE
. U = V-W
. T = T+1.5
...FIN
```

```
WHEN (X.EQ.Y)
. U = V+W
. T = T-1.5
...FIN
ELSE
. U = V-W
. T = T+1.5
...FIN
```

Flow Chart:



NOTE: WHEN and ELSE always come as a pair of statements, never separately. Either the WHEN or the ELSE or both may assume the multi-line form. ELSE is considered to be a control phrase, hence cannot be placed on the same line as the WHEN. Thus “WHEN (L) S1 ELSE S2” is not valid.

### 2.1.1.4 CONDITIONAL

Description: The **CONDITIONAL** statement is based on the LISP conditional. A list of logical expressions is evaluated one by one until the first expression to be true is encountered. The scope corresponding to that expression is executed, and control then passes to the first statement following the **CONDITIONAL**. If all expressions are false, no scope is executed. (See, however, the note about **OTHERWISE** below.)

General Form:

```

CONDITIONAL
. (L1) S1
. (L2) S2
. . .
. . .
...FIN

```

Examples:

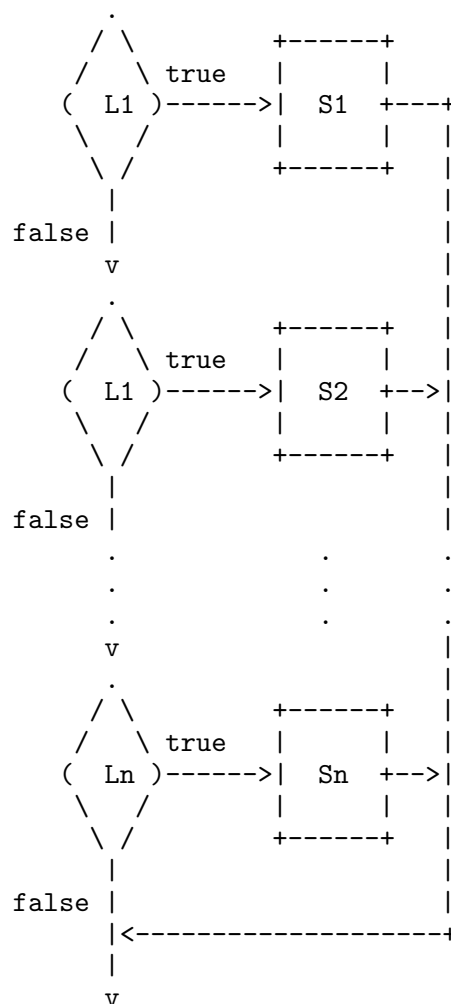
```

CONDITIONAL
. (x.LT.-5.0)  U = U+W
. (X.LE.1.0)   U = U+W+Z
. (x.LE.10.5)  U = U-Z
...FIN

CONDITIONAL
. (A.EQ.B) Z = 1.0
. (A.LE.C)
. . Y = 2.0
. . 2 = 3.4
. . FIN
. (A.GT.C.AND.A.LT.B) Z = 6.2
. (OTHERWISE) Z = 0.0
...FIN

```

Flow Chart:



Notes: The **CONDITIONAL** itself does not possess a one-line form. However, each “ $(Ln) Sn$ ” is treated as a structured statement and may be in one-line or multi-line form.

The reserved word **OTHERWISE** represents a catchall condition. That is, “**(OTHERWISE)  $Sn$** ” is equivalent to “**(.TRUE.)  $Sn$** ” in a **CONDITIONAL** statement.

### 2.1.1.5 SELECT

Description: The **SELECT** statement is similar to the **CONDITIONAL** but is more specialized. It allows an expression to be tested for equality to each expression in a list of expressions. When the first matching expression is encountered, a corresponding scope is executed and the **SELECT** statement terminates. In the description below, **E**, **E1**, ..., **En** represent arbitrary but compatible expressions. Any type of expression (integer, real, complex, ...) is allowed as long as the underlying Fortran system allows such expressions to be compared with an **.EQ.** or **.NE.** operator.

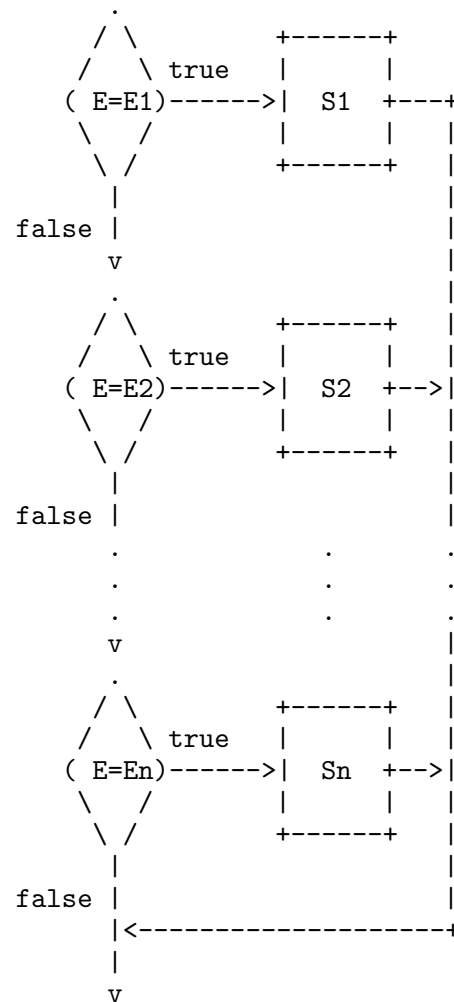
General Form:

```
SELECT (E)
. (E1) S1
. (E2) S2
. . .
. . .
. (En) Sn
...FIN
```

Example:

```
SELECT (OPCODE(PC))
. (JUMP) PC = AD
. (ADD)
. . A = A+B
. . PC = PC+1
. ...FIN
. (SKIP) PC = PC+2
. (STOP) CALL STOPCD
...FIN
```

Flow Chart:



Notes: As in the case of **CONDITIONAL**, at most one of the **Si** will be executed.

The catchall **OTHERWISE** may also be used in a **SELECT** statement. Thus “(OTHERWISE) **Sn**” is equivalent to “(E) **Sn**” within a “**SELECT (E)**” statement.

The expression is reevaluated for each comparison in the list, thus lengthy, time consuming, or irreproducible expressions should be precomputed, assigned to a variable, and the variable used in the specification portion of the **SELECT** statement

### 2.1.2 Loop Structures

The structured statements described below all have a scope which is executed a variable number of times depending on specified conditions.

Of the five loops presented, the most useful are the `DO`, `WHILE`, and `REPEAT UNTIL` loops. To avoid confusion, the `REPEAT WHILE` and `UNTIL` loops should be ignored initially.

### 2.1.2.1 DO

Description: The Flecs `DO` loop is intended as a substitute for the Fortran `DO` loop, with the primary difference being syntactic. The Flecs translator is capable of generating two different forms of Fortran code for the `DO` loop, depending on the setting of the translator directive, `FAKEDO`, see Section 2.3 [Translator Directives], page 22, for more information. In one form, the `DO` is translated directly into the Fortran code by constructing a statement label and a `CONTINUE` statement at the end of the statements. This form presents a problem when the loop contains internal procedures, see Section 2.2 [Internal Procedures], page 20. Internal procedure calls are translated into `GO TO` statements, and violate the Fortran-77 standard which prohibits extended range `DO` loops. Therefore, Flecs can generate an alternate form of `DO` loop, the `FAKEDO` form, which replaces the `DO` loop with equivalent Fortran code.

In the syntax of Flecs `DO` loop, the statement number is omitted from the `DO` statement, the incrementation parameters are enclosed in parenthesis, and the scope is indicated by either the one line or multi-line convention.

General Form:

```
DO (V=START,STOP,INC) S
```

Form 1:

```
DO 10 V=START,STOP,INC
S
10 CONTINUE
```

Form 2 (Fakedo):

```
VAR=START
GO TO 20
10 VAR=VAR+(INC)
20 IF(INC.LT.0) GO TO 30
IF(VAR.GT.STOP) GO TO 50
GO TO 40
30 IF(VAR.LT.STOP) GO TO 50
40 S
GO TO 10
50 CONTINUE
```

Examples:

```
DO (I = 1,N) A (I) = 0.0

DO (J = 3,K,J)
. B(J) = B(J-1)*B(J-2)
. C(J) = SIN(B(J))
...FIN
```

### 2.1.2.2 WHILE

Description: The `WHILE` loop causes its scope to be repeatedly executed while a specified condition is true. The condition is checked prior to the first execution of the scope, thus if the condition is initially false the scope will not be executed at all.

General Form:

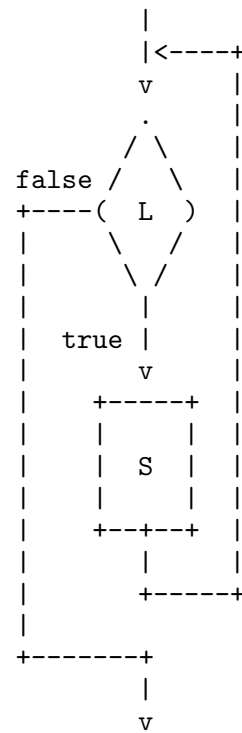
```
WHILE (L) S
```

Examples:

```
WHILE (X.LT.A(I)) I = I+1
```

```
WHILE (P.NE.0)
.  VAL(P) = VAL(P)+1
.  P = LINK(P)
...FIN
```

Flow Chart:



### 2.1.2.3 REPEAT WHILE

Description: By using the REPEAT verb, the test can be logically moved to the end of the loop. The REPEAT WHILE loop causes its scope to be repeatedly executed while a specified condition remains true. The condition is not checked until after the first execution of the scope. Thus the scope will always be executed at least once and the condition indicates under what conditions the scope is to be repeated. Note: "REPEAT WHILE(L)" is functionally equivalent to "REPEAT WHILE(.NOT. (L))"

General Form:

```
REPEAT WHILE (L) S
```

Examples:

```
REPEAT WHILE (N.EQ.M(I)) I = I+1
```

```
REPEAT WHILE (LINK(Q).NE.0)
```

```
. R = LINK(O)
```

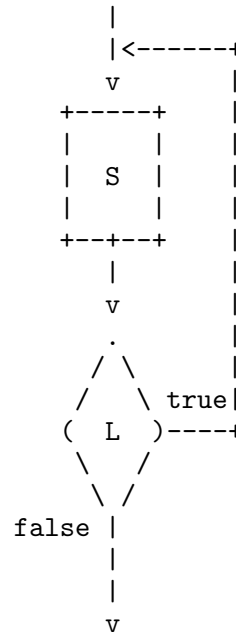
```
. LINK(Q) = P
```

```
. P = Q
```

```
. Q = R
```

```
...FIN
```

Flow Chart:



### 2.1.2.4 UNTIL

Description: The UNTIL loop causes its scope to be repeatedly executed until a specified condition becomes true. The condition is checked prior to the first execution of the scope, thus if the condition is initially true, the scope will not be executed at all. Note that “UNTIL (L)” is functionally equivalent to “WHILE (.NOT. (L))”.

General Form:

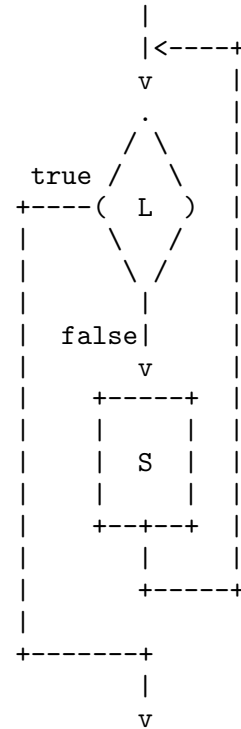
```
UNTIL (L) S
```

Examples:

```
UNTIL (X.EQ.A(I))  I = I+1
```

```
UNTIL (P.EQ.0)
.  VAL(P) = VAL(P)+1
.  P = LINK(P)
...FIN
```

Flow Chart:



### 2.1.2.5 REPEAT UNTIL

Description: By using the REPEAT verb, the test can be logically moved to the end of the loop. The REPEAT UNTIL loop causes its scope to be repeatedly executed until a specified condition becomes true. The condition is not checked until after the first execution of the scope. Thus the scope will always be executed at least once and the condition indicates under what conditions the repetition of the scope is to be terminated.

General Form:

```
REPEAT UNTIL (L) S
```

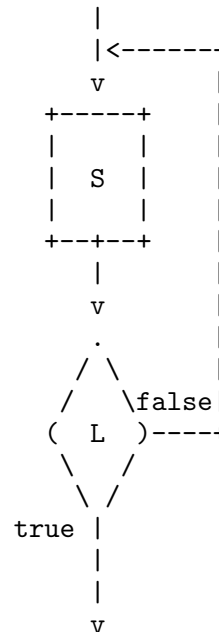
Examples:

```
REPEAT UNTIL (N.EQ.M(I)) I = I+1
```

```
REPEAT UNTIL (LINK(Q).EQ.0)
```

```
. R = LINK(Q)
. LINK(Q) = P
. P = Q
. Q = R
...FIN
```

Flow Chart:



## 2.2 Internal Procedures

In Flecs a sequence of statements may be declared an *internal procedure* and given a name. The procedure may then be invoked from any point in the program by simply giving its name.

*Procedure names* may be any string of letters, digits, and hyphens (i.e. minus signs) beginning with a letter and containing at least one hyphen. Internal blanks are not allowed. The only restriction on the length of a name is that it may not be continued onto a second line.

Examples of valid internal procedure names:

```
INITIALIZE-ARRAYS
GIVE-WARNING
SORT-INTO-DESCENDING-ORDER
INITIATE-PHASE-3
```

A *procedure declaration* consists of the keyword “TO” followed by the procedure name and its scope. The set of statements comprising the procedure is called its scope. If the scope consists of a single simple statement it may be placed on the same line as the “TO” and procedure name, otherwise the statements of the scope are placed on the following lines and terminated with a FIN statement. These rules are analogous with the rules for forming the scope of a structured statement.

General Form of procedure declaration:

```
TO procedure-name
```

Examples of procedure declarations:



```

TO RESET-POINTER  P = 0

TO DO-NOTHING CONTINUE

TO SUMMARIZE-FILE
.  INITIALIZE-SUMMARY
.  OPEN-FILE
.  REPEAT UNTIL (EOF)
.  .  ATTEMPT-TO-READ-RECORD
.  .  WHEN (EOF) CLOSE-FILE
.  .  ELSE UPDATE-SUMMARY
.  ...FIN
.  OUTPUT-SUMMARY
...FIN

```

An *internal procedure reference* is a procedure name appearing where an executable statement would be expected. In fact an internal procedure reference is an executable simple statement and thus may be used in the scope of a structured statement as in the last example above. When control reaches a procedure reference during execution of a Flecs program, a return address is saved and control is transferred to the first statement in the scope of the procedure. When control reaches the end of the scope, control is transferred back to the statement logically following the procedure reference.

A typical Flecs program or subprogram consists of a sequence of Fortran declarations: (e.g. INTEGER, DIMENSION, COMMON, etc.) followed by a sequence of executable statements called the body of the program followed by the Flecs internal procedure declarations, if any, and finally the END statement.

Here is a complete (but uninteresting) Flecs program which illustrates the placement of the procedure declarations.

```

00010 C INTERACTIVE PROGRAM FOR THE PDP-10 TO COMPUTE X**2
00020 C ZERO IS USED AS A SENTINEL VALUE TO TERMINATE EXECUTION.
00030
00040      REAL  X,XSQ
00050      REPEAT UNTIL (X.EQ.0)
00060      .  GET-A-VALUE-OF-X
00070      .  IF (X.NE.0)
00080      .  .  COMPUTE-RESULT
00090      .  .  TYPE-RESULT
00100      .  .  FIN
00110      ...FIN
00120      CALL EXIT

-----

00130      TO GET-A-VALUE-OF-X
00140      .  TYPE 10
00150 10      .  FORMAT ( ' X = ', $ )
00160      .  ACCEPT 20,X
00170 20      .  FORMAT ( F )
00180      ...FIN

-----

```

```

00190          TO COMPUTE-RESULT XSQ = X*X

-----

00200          TO TYPE-RESULT
00210          .   TYPE 30,XSQ
00220  30      .   FORMAT(' X-SQUARED  =  ',F7.2)
00230          .. FIN
00240          END

```

Notes concerning internal procedures:

1. All internal procedure declarations must be placed at the end of the program just prior to the `END` statement. The appearance of the first “`TO`” statement terminates the body of the program. The translator expects to see nothing but procedure declarations from that point on.
2. If an internal procedure is called from within a `DO` loop, you must ensure that either `FAKEDO` is enabled, see Section 2.3 [Translator Directives], page 22, or that the Fortran compiler correctly handles extended range `DO` loops. By default, `FAKEDO` is enabled.
3. The order of the declarations is not important. Alphabetical by name is an excellent order for programs with a large number of procedures.
4. Procedure declarations may not be nested. In other words, the scope of a procedure may not contain a procedure declaration. It may of course contain executable procedure references.
5. Any procedure may contain references to any other procedures (excluding itself).
6. Dynamic recursion of procedure referencing is not permitted.
7. All program variables within a main or subprogram are global and are accessible to the statements in all procedures declared within that same main or subprogram.
8. There is no formal mechanism for defining or passing parameters to an internal procedure. When parameter passing is needed, the Fortran function or subroutine subprogram mechanism may be used or the programmer may invent his own parameter passing methods using the global nature of variables over internal procedures.
9. The Flecs translator separates procedure declarations on the listing by dashed lines as shown in the preceding example.

## 2.3 Translator Directives

The operation of Flecs translator can be modified using translator directives. These commands are specified using a special form of comment statement which has the form,

```
C $FLECS directive state
```

Any amount of white space can separate the various components. At present, all of the directives are Boolean, i.e., they are either on or off. The *state* can be specified as `ON`, `OFF`, `TRUE` or `FALSE`. The following directives are provided:

**CPASS** Normally, the Flecs translator does not pass comment to the output Fortran source code. When this directive is enabled, comments pass through.

**FAKEDO, DFAKDO**

These directives control how `DO` loops are translated. They only apply to the Flecs form of `DO` loops. `DO` loops that are coded in Fortran (using an explicit statement number) are *not* affected by these directives. When a new subprogram is translated, `FAKEDO` is set to the value in `DFAKDO` (`DFAKDO` stands for Default FAKE DO). Whenever a `DO` loop is encountered, a Fortran style `DO` loop will be generated if `FAKEDO` is `FALSE`.

Otherwise, a semantic equivalent will be generated using assignments, `IF` statements, and `GOTO`s. See See Section 2.1.2.1 [DO], page 16, for a description of both generated forms.

#### `TRANSLATE`

`TRANSLATE` controls whether the translator actually performs any translations. Normally, `TRANSLATE` is turned on, but it can be turned off when pure Fortran code is incorporated into a Flecs subprogram. It applies for only one subprogram after which it is turned on.



### 3 Restrictions and Notes

If Flecs were implemented by a nice intelligent compiler this chapter would be much shorter. Currently, however, Flecs is implemented by a sturdy but naive translator. Thus the Flecs programmer must observe the following restrictions.

1. Flecs must invent many statement numbers in creating the Fortran program. It does so by beginning with a large number (usually 99999) and generating successively smaller numbers as it needs them. Do not use a number which will be generated by the translator. A good rule of thumb is to *avoid using 5 digit statement numbers*.
2. The Flecs translator must generate integer variable names. It does so by using names of the form “I~~nnnnn~~” when ~~nnnnn~~ is a 5 digit number related to a generated statement number. *Do not use variables of the form I~~nnnnn~~ and avoid causing them to be declared other than INTEGER.* For example the declaration “IMPLICIT REAL (A-Z)” leads to trouble. Try “IMPLICIT REAL (A-H, J-Z)” instead.
3. The translator does not recognize continuation lines in the source file. Thus Fortran statements may be continued since the statement and its continuations will be passed through the translator without alteration, see Section 1.1 [Retention of Fortran Features], page 5. However, *an extended Flecs statement which requires translations may not be continued.* The reasons one might wish to continue a Flecs statement are 1) It is a structured statement or procedure declaration with a one statement scope too long to fit on a line, or 2) it contains an excessively long specification portion or 3) both of the above. Problem 1) can be avoided by going to the multi-line form. Frequently problem 2) can be avoided when the specification is an expression (logical or otherwise) by assigning the expression to a variable in a preceding statement and then using the variable as the specification.
4. Fortran-77 prohibits extended range DO loops. This presents a problem when internal procedures are used within DO loops. Some Fortran compilers work correctly with this combination; others do not. Flecs can generate an alternate form of DO loop which uses IFs and GOTOs to achieve the same semantics as a DO loop. The translator directives, FAKEDO and DFAKDO, can be used to control this feature. These directives are normally turned on, so that all Flecs programs have the correct semantics, but when code optimization is desired, it is necessary to be selective in their use.
5. *Blanks are meaningful separators in Flecs statements; don't put them in dumb places* like the middle of identifiers or key words and do use them to separate distinct words like REPEAT and UNTIL.
6. Let Flecs indent the listing. *Start all statements in column 7*, and the listing will always reveal the true structure of the program. (as understood by the translator, of course).
7. As far as the translator is concerned, FORMAT statements are executable Fortran statements since it doesn't recognize them as extended Flecs statements. Thus, *only place FORMAT statements where an executable Fortran statement would be acceptable.* Don't put them between the end of a WHEN statement and the beginning of an ELSE statement. Don't put them between procedure declarations.

Incorrect Examples:	Corrected Examples:
<pre>           WHEN (FLAG) WRITE(3,30) 30  FORMAT(' TITLE: ')           ELSE LINE = LINE+1 </pre>	<pre>           WHEN (FLAG) .   WRITE( 3, 30) 30  .   FORMAT(' TITLE: ') .   ...FIN           ELSE LINE = LINE+1 </pre>

<pre>       TO WRITE-HEADER       . PAGE = PAGE+1       . WRITE(3,40) H,PAGE       ...FIN 40  FORMAT (70A1,I3) </pre>	<pre>       TO WRITE-HEADER       . PAGE = PAGE+1       . WRITE(3,40) H,PAGE       . FORMAT (70A1,I3) 40  ...FIN </pre>
---	---

8. The translator, being simple-minded, recognizes extended Flecs statements by the process of scanning the first identifier on the line. If the identifier is one of the Flecs keywords **IF**, **WHEN**, **UNLESS**, **FIN**, etc., the line is assumed to be a Flecs statement and is treated as such. Thus, *the Flecs keywords are reserved, and may not be used as variable names*. In case of necessity, a variable name, say **WHEN**, may be slipped past the translator by embedding a blank within it. Thus “WH EN” will look like “WH” followed by “EN” to the translator which is blank sensitive, but like “WHEN” to the compiler which ignores blanks. Another alternative is to turn off the **TRANSLATE** translator directive, see Section 2.3 [Translator Directives], page 22, which inhibits the translation process entirely.
9. In scanning a parenthesized specification, the translator scans from left to right to find the parenthesis which matches the initial left parenthesis of the specification. The translator, however, is ignorant of Fortran syntax including the concept of Hollerith constants and will treat Hollerith parenthesis as syntactic parenthesis. Thus, *avoid placing Hollerith constants containing unbalanced parenthesis within specifications*. If necessary, assign such constants to a variable, using a **DATA** or assignment statement, and place the variable in the specification.

Incorrect Example:

```
IF (J.EQ. '(')
```

Corrected Example:

```
LP = '( '
IF(J.EQ.LP)
```

10. The Flecs translator will not supply the statements necessary to cause appropriate termination of main and sub-programs. Thus, *it is necessary to include the appropriate **RETURN**, **STOP**, or **CALL EXIT** statement prior to the first internal procedure declaration*. Failure to do so will result in control entering the scope of the first procedure after leaving the body of the program. Do not place such statements between the procedure declarations and the **END** statement.

## 4 Errors

This section provides a framework for understanding the error handling mechanisms of the Flecs Translator. The system described below is at an early point in evolution, but has proven to be quite workable.

The Flecs translator examines a Flecs program on a line by line basis. As each line is encountered it is first subjected to a limited *syntax* analysis followed by a *context* analysis. Errors may be detected during either of these analysis. It is also possible for errors to go *undetected* by the translator.

### 4.1 Syntax Errors

When a syntax error is detected by the translator, it ignores the statement. On the Flecs listing the line number of the statement is overprinted with hyphens (“-”) to indicate that the statement has been ignored. The nature of the syntax error is given in a message on the following line.

The fact that a statement has been ignored may, of course, cause some context errors in later statements. For example the control phrase “WHEN (X(I).LT.(3+4)” has a missing right parenthesis. This statement will be ignored, causing as a minimum the following ELSE to be out of context. The programmer should of course be aware of such effects. More is said about them in the next section.

### 4.2 Context Errors

If a statement successfully passes the syntax analysis, it is checked to see if it is in the appropriate context within the program. For example an ELSE must appear following a WHEN and nowhere else. If an ELSE does not appear at the appropriate point or if it appears at some other point, then a context error has occurred. A frequent source of context errors in the initial stages of development of a program comes from miscounting the number of FIN's needed at some point in the program.

With the exception of excess FIN's which do not match any preceding control phrase and are ignored, all context errors are treated with a uniform strategy. When an out-of-context source statement is encountered, the translator generates a “STATEMENT(S) NEEDED” message. It then invents and processes a sequence of statements which, if they had been included at that point in the program, would have placed the original source statement in a correct context. A message is given for each such statement invented. The original source statement is then processed in the newly created context.

By inventing statements, the translator is not trying to patch up the program so that it will run correctly. It is simply trying to adjust the local context so that the original source statement and the statements which follow will be acceptable on a context basis. As in the case of context errors generated by ignoring a syntactically incorrect statement, such an adjustment of context frequently causes further context errors later on. This is called *propagation of context errors*.

One nice feature of the context adjustment strategy is that context errors cannot propagate past a recognizable procedure declaration. This is because the “TO” declaration is in context only at indentation level O. Thus to place it in context, the translator must invent enough statements to terminate all open control structures which precede the “TO”. The programmer who modularizes his program into a collection of relatively short internal procedures, limits the potential for propagation of context errors.

### 4.3 Undetected Errors

The Flecs translator is ignorant of most details of Fortran syntax. Thus most Fortran syntax errors will be detected by the Fortran compiler not the Flecs translator. In addition there are two major classes of Flecs errors which will be caught by the compiler not the translator.

The first class of undetected errors involve misspelled Flecs keywords. A misspelled keyword will not be recognized by the translator. The line on which it occurs will be assumed to be a Fortran statement and will be passed unaltered to the compiler which will no doubt object to it. For example a common error is to spell UNTIL with two L's. Such statements are passed to the compiler, which then produces an error message. The fact that an intended control phrase was not recognized frequently causes a later context error since a level of indentation will not be triggered.

The second class of undetected errors involves unbalanced parentheses. (See also note 8 in Chapter 3 [Restrictions and Notes], page 25.). When scanning a parenthesized specification, the translator is looking for a matching right parenthesis. If the matching parenthesis is encountered before the end of the line, the remainder of the line is scanned. If the remainder is blank or consists of a recognizable internal procedure reference, all is well. If neither of the above two cases hold, the remainder of the line is assumed (without checking) to be a simple Fortran statement which is passed to the compiler. Of course, this assumption may be wrong. Thus the statement

```
WHEN (X.LT.A(I)+Z)) X = 0
```

is broken into

```
keyword:          WHEN
specification:    (X.LT.A(I)+Z)
Fortran statement: ) X = 0
```

Needless to say the compiler will object to “) X = 0” as a statement.

Programmers on batch oriented systems have less difficulty with undetected errors due to the practice of running the program through both the translator and the compiler each time a run is submitted. The compiler errors usually point out any errors undetected by the translator.

Programmers on timesharing systems tend to have a bit more difficulty since an undetected error in one line may trigger a context error in a much later line. Noticing the context error, the programmer does not proceed with compilation and hence is not warned by the compiler of the genuine cause of the error. One indication of the true source of the error may be an indentation failure at the corresponding point in the listing.

## 4.4 Other Errors

The Translator detects a variety of other errors such as multiply defined, or undefined procedure references. The error messages are self-explanatory. (Really and truly!)



## 5 Procedure for Use

The following subsections describe the procedures for using the Flecs translator on the various machines where it has been ported.

### 5.1 Source Preparation

Prepare a Flecs source file with any name of your choosing and an extension of `‘.flx’`. It is important to use lower case on Unix machines for the extension. As with many Fortrans, the “tab to column 7” convention may be used.

### 5.2 Running the Translator

On the VAX running VMS, the Silicon Graphics Iris 4D, the Convex, and the Cray running UNICOS; Flecs is executed as a command to the shell. The usage is as follows:

```
flecs files
```

where *files* is a list of Flecs source files, each separated by a blank (additionally commas on the VAX). If a file specified in a command has no suffix, then `‘.flx’` will be added automatically. The translator will use the main part of the file name as the prefix for the Flecs listing and translated Fortran output. The Flecs listing has a suffix of `‘.fli’`, and the Fortran listing has a suffix of `‘.f’` on Unix machines, and `‘.for’` on VMS machines.

Example:

```
flecs util string.flx
```

would read the files `‘util.flx’` and `‘string.flx’`, and produce Fortran translations into `‘util.f’` and `‘string.f’`, and write Flecs listings into `‘util.fli’` and `‘string.fli’`. On VMS machines, the Fortran translations would be written into `‘util.for’` and `‘string.for’`, instead.

In the event of errors, failure status codes are returned to the shell.

In environments where Fortran programs cannot gain access to the command line directly, Flecs can be modified to read from Fortran units. See Chapter 6 [Flecs Implementation], page 31, for more information.



## 6 Flecs Implementation

The Flecs translator system is written in Flecs and has been designed with ease of transportation from one machine to another and adaptability to varying system configurations as two of its primary goals. The Flecs system may be freely copied and transported without explicit permission of the authors, provided that the copyright notice and warranty disclaimer are included.

This chapter is intended to assist programmers who wish to further modify a Flecs translator which has already been adapted to their machine. For the sake of completeness it also describes the changes necessary in moving Flecs from one machine to another.

Only those portions of the translator which are pertinent to adapting it to varying machine and system configurations are considered. The internal logic of the translator is not discussed except as it relates to these goals.

It is assumed the reader has a knowledge of the Flecs language and translator system equivalent to that found in the Flecs Users Manual.

### 6.1 Necessary and Desirable Modifications

Modification of the standard Flecs system may be necessary or desirable to accomplish any of the following goals:

1. To adapt the translator to a new character set;
2. To adapt the translator to a new operating system and I/O environment;
3. To adapt to idiosyncrasies in the Fortran dialect with which the translator must interface;
4. To adjust the size and capacity of the translator to account for available memory capacity and expected workload;
5. To improve the speed of the translator by replacing the standard Flecs-coded subroutines with more efficient versions.

In adapting the translator to a new computer one can identify two classes of modifications. The first class consists of those modifications which are absolutely necessary before the translator can function at all on the given machine. We call these the *necessary modifications*. Examples of necessary modifications are those required to adapt the translator to the appropriate character set, memory size, and I/O configurations. Most, if not all, of the necessary modifications can be performed by rewriting certain `DIMENSION`, `DATA`, and `FORMAT` statements and by rewriting a few of the machine dependent subroutines. Explicit instructions for carrying out the necessary modifications are given in this manual.

The second class of modifications are those which improve the quality of service provided by the translator. We call these the *desirable modifications*. Examples of desirable modifications are improvement in execution time, adaptation to convenient but unorthodox features of the local Fortran dialect, and integration of the translator into the operating system so that it becomes more convenient to use. Many of the desirable modifications may be accomplished by rewriting some of the standard subroutines. Suggestions for carrying out these modifications are included in this manual. Desirable modifications which can be accomplished only by altering the fundamental logical structure of the system are beyond the scope of this manual.

### 6.2 System Structure

The standard Flecs translator system consists of a main program, I/O interface subroutines, character processing subroutines, and a trivial statement number generating subroutine. The entire collection consisting of the main program and all subprograms will be referred to as the Flecs system or simply the system. The main program by itself will be referred to as the Flecs translator or simply the translator.

The Flecs translator embodies the algorithm for translating Flecs to Fortran and for producing the Flecs listing. The string processing subroutines are used by both the translator and the I/O subroutines to perform most of the necessary character string manipulation. The five I/O subroutines `OPENF`, `GET`, `PUT`, `CLOSEF`, and `FLSTOP` represent the interface between the Flecs system and the operating system of the computer. The statement number generating subroutine `NEWNO` generates a new Fortran statement number each time it is called.

Certain variables within the translator are called *parameters* and are assigned initial values by `DATA` statements within the translator. Many of the necessary or desirable modifications may be effected by altering the initial values of these parameters. Some of the parameters are communicated to various subroutines in the system via the labeled `COMMON /PARAM/`. With this exception, all communication between programs is via the usual Fortran linkages.

Most I/O and operating system dependencies have been isolated within the I/O interface subroutines. By altering or rewriting these routines, the translator may be adapted to a variety of I/O and operating system environments from card-oriented batch systems to terminal-oriented time sharing systems. Alteration of the I/O subroutines may also be used to compensate partially for certain pathologies in the local Fortran dialect.

In order to further simplify the maintenance of Flecs across different platforms, the C preprocessor<sup>1</sup> is used to conditionally compile operating system dependencies, and to allow for debugging statements to be conditionally included. If the C preprocessor is not available on your system, one can obtain a free C preprocessor from the Free Software Foundation as part of the GNU Emacs distribution or as part of their C compiler, or it can be obtained from DECUS. As a last resort, the selection of appropriate statements can be done by hand. The meaning of the various preprocessor variables is easily determined from the source code.

Some of the subroutines in the translator have been written to exercise uncommon features in the translator. By so doing, the translator makes a good test case for itself.

## 6.3 Character String Conventions

The Flecs translator is basically a character string processing program. In the past, character string processing in Fortran was only possible in machine dependent ways, and the original implementation of Flecs reflected those limitations. With the advent of Fortran-77, a character data type is provided in the language, and character string processing is much simpler.

Logically, the translator uses varying length character strings. These are implemented in the translator by using two variables for each ‘string’, one variable holds the characters and is defined as `CHARACTER` type, and the additional variable holds the length and has `INTEGER` type. When a character string is passed to a subroutine, and the subroutine is only reading the characters, then a substring reference can be used to refer to just the relevant part of the string. If the subroutine modifies the string, then both the string and its length must be passed so that the length can be updated.

The current version of the translator is written for the Ascii character set, but adaptation to other character sets is straightforward. It is assumed that the internal character codes for the digits “0” through “9” are consecutive and increasing in value. For example, in 7-bit ASCII the values are 48 through 57. This assumption is used within the translator only in the procedure `SCAN-STATEMENT-NUMBER`.

## 6.4 Translation Parameters

This chapter describes the meaning and use of the various parameters which control the functioning of the translator. Each parameter is given an initial value by a `DATA` statement in the “PARAMETERS”

---

<sup>1</sup> B. W. Kernighan and D. W. Ritchie, *The C Programming Language, 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

section of the declarations (see the listing of the translator). The parameters are listed below in alphabetical order.

- CHSPAC** (CHaracter code for SPACe)  
This parameter is set to the value of the character code for a space (or blank). It is used by the translator in checking on the continuation column (column 6) and is made available to the subroutines via the **COMMON /PARAM/**.
- CHZERO** (CHaracter code for ZERO)  
The procedure **SCAN-STATEMENT-NUMBER** creates a Fortran integer whose value is equal to the statement number of the current Flecs source statement. To do so it must know the machine's code values for the characters “@”, . . . , “@”. It computes these values from **CHZERO**, the code for “@”, by assuming the values are consecutive and increasing in order. (If this assumption is not correct for a given machine, the procedure **SCAN-STATEMENT-NUMBER** must be rewritten.) For example, the 7- bit ASCII code for zero is 60 (octal) 48 (decimal). Initialization of this variable is done using the Fortran intrinsic function, **ICHAR**.
- COGOTO, FAKE, LONG, SHORT**  
(Internal procedure linkage)  
When an internal procedure is called, the translator will generate a new statement number which will be used to label the statement following the procedure call. generate a **GO TO** statement to the first statement of the internal procedure, and the statement following the **GO TO** will be labeled with a generated statement number. When the internal procedure finishes execution, it must return to the appropriate calling point. The mechanism by which an internal procedure is called and by which control is returned to the calling point can be implemented by basically two different methods, a computed **GOTO** or an assigned **GOTO**. The four variables control which method is used. One and only one variable should be set to **.TRUE.**, the remainder must be set to **.FALSE.**  
  
The most portable method is the computed **GO TO** method. If the variable, **COGOTO** is **.TRUE.**, then the translator will use a computed **GOTO** for returning from an internal procedure.  
  
The other three variables permit various types of assigned **GO TO** statements to be used. **SHORT** specifies the most straightforward, a **GO TO i**, with no list of of possible statement numbers following. **LONG** will construct an assigned **GO TO** with the full list of possibilities following the variable name. **FAKE** can be used only with compilers which insist on the presence of a list of statement numbers after an assigned **GOTO** but ignore its content. It should not be used if the short form is valid. If the short form is not valid then **FAKE** may be used provided that 1) it works properly 2) the user is willing to trade trickiness and obscurity for efficiency.
- LWIDTH** (Listing WIDTH)  
This parameter informs the translator of the number of printable positions available for creating the Flecs listing. A common value is 132 for many high speed line printers.
- MAXSTK** (MAXimum STAcK size)  
The parameter **MAXSTK** informs the translator of the size of the array **STACK**. This array is used for two purposes. It contains the working stack of the translator at one end beginning with **STACK(1)** and the internal procedure cross reference table at the other end beginning with **STACK(MAXSTK)**. The size of the array is changed by altering its **DIMENSION** statement and by supplying its new size to the translator via **MAXSTK**. A value of 2000 has been found to be more than sufficient for translating the translator itself on all machines encountered so far.

<b>PRIME</b>	(a PRIME number) This parameter should be a positive prime number. The cross reference table of internal procedure names is maintained as a hash table with <b>PRIME</b> number of buckets. Considerations in selecting a value of <b>PRIME</b> are: <ol style="list-style-type: none"> <li>1. Larger values of <b>PRIME</b> tend to cause fewer collisions in the hash table and make table referencing more efficient.</li> <li>2. The number of entries in array <b>STACK</b> used for bucket header pointers is exactly <b>PRIME</b>. Hence, smaller values of <b>PRIME</b> save space.</li> <li>3. The relationships among the value of <b>PRIME</b>, the hashing algorithm employed by subprogram <b>HASH</b>, and the character codes of the machine may be adjusted to reduce the expected number of collisions. (See Knuth, D. E., Vol. 3 of <i>The Art of Computer Programming</i>, section 6.4.)</li> </ol>
<b>OUTCNT</b>	(OUTput CouNter) Keeps track of the number of Fortran statements output thus far. It is printed on the listing file so that FLECS statements can be related to Fortran statement numbers.
<b>SAFETY</b>	(SAFETY margin) Within the main loop of the procedure <b>PROCESS-PROGRAM</b> , entries may be pushed onto the working stack. Rather than check for stack overflow as each entry is pushed on, a check is made at the beginning of the loop to insure that at least <b>SAFETY</b> entries are available between the current top of the stack and the bottom of the table area. It is normally not possible to push more than <b>SAFETY</b> items onto the stack in one cycle of the loop. Those procedures which might cause such an overflow contain their own overflow checking code. The value of <b>SAFETY</b> need not be altered by the implementator unless he is altering the logic of stack or table usage within the translator.
<b>SEEDNO</b>	(SEED for statement Numbers) The translator generates new Fortran statement numbers in descending sequence starting with the value <b>SEEDNO-1</b> . Normally, <b>SEEDNO</b> is initialized to 100000 so that numbers are generated in the sequence 99999, 99998, . . . . However, on some machines of small word length it may not be possible to represent a number this large in a Fortran integer. In this case, a smaller seed number may be chosen.

## 6.5 Character Processing Subroutines

There are a number of character or character string processing subprograms supplied with the system. They have been written in FLECS with transportability, not efficiency, as a primary goal. These routines are heavily used by the translator, but are not particularly efficient as written and could be replaced by more efficient versions if desired. However, one should keep in mind that the time for compiling the Fortran code is usually much longer than the time for translation, so the gains may not have much effect. However, some implementations of Fortran I/O or string manipulation are particularly inefficient. Anyone porting the translator should measure the execution time of translation versus compilation of a FLECS program. If the translation times are excessive relative to the compilation time, then some effort should be spent on efficiency. Previous experience indicates that profiling followed by a few small modifications will solve most efficiency problems.

Functional definitions of the character processing subroutines are given below listed in alphabetical order by subroutine name. In describing the arguments of the subroutines, the designations “in”, “out”, and “in/out” are used with the following meanings:

<b>in</b>	The argument contains a value supplied to the subroutine by the calling program. The value is not altered by the subprogram. In the case of character strings, no length is passed. Rather, the maximum length of the string as specified by the Fortran function, <b>LEN</b> , gives the length of the string.
-----------	---

- out            The initial value of the argument is ignored by the subroutine and it is assigned a new value by the subroutine. If an argument is a string, and the length can be changed by the subroutine, then a length variable for the string is also passed.
- in/out        The argument contains a value supplied to the subroutine by the calling program which may be altered by the subroutine. If an argument is a string, and the length can be changed by the subroutine, then a length variable for the string is also passed.

The input values need not be validity checked by the subroutines to insure that they fall within the proper ranges.

### 6.5.1 CATNUM (conCATenate NUMber to string)

This subroutine is used to concatenate the decimal character representation of a number to the end of a string. The invocation is

```
CALL CATNUM(A,ALEN,N)
```

where

- A            (in/out) is a varying length string;
- ALEN        (in/out) is the current length of A;
- N            (in) is an integer variable whose value is in the range 0 through 99999.

The value of N is converted to a five digit decimal character representation. Leading zeroes are supplied if necessary. The resulting five characters are concatenated to the end of the string A. The length of A, ALEN, is increased by five.

Example:

```
Before:  A      'IF (X.LT.3) GO TO '
         ALEN   18
         N      973

After:   A      'IF (X.LT.Y) GO TO 00973'
         ALEN   23
```

### 6.5.2 CATSTR (conCATenate STRing to string)

This subroutine is used to concatenate one string to another. The invocation is

```
CALL CATSTR(A,ALEN,B)
```

where

- A            (in/out) is a varying length string;
- ALEN        (in/out) is the length of A;
- B            (in) is a string whose full length is used.

The character string represented by B is concatenated to the right hand end of A thus increasing the length of A by an amount equal to the length of B. If B is of length zero, A will remain unchanged.

Example:

```
Before:  A      'IF (X.LT.Y) '
         ALEN   11
         B      ' GO TO '

After:   A      'IF (X.LT.Y) GO TO '
         ALEN   18
```

### 6.5.3 CATSUB (conCATenate SUBstring to string)

This subroutine is used to concatenate a portion of one string to another. The invocation is

```
CALL CATSUB(A,ALEN,B,START,LENGTH)
```

where

**A** (in/out) is a varying length string;  
**ALEN** (in/out) is the length of **A**;  
**B** (in) is a string;  
**START** (in) is a positive integer representing the position of the first character of the substring of **B** to be concatenated to the string **A**;  
**LENGTH** (in) is the length of the substring of **B** to be concatenated to string **A**.

The substring of **B** beginning with the character having position **START** and of length **LENGTH** is concatenated to the end of string **A**. The length of **A** is increased by **LENGTH**. If **LENGTH** equals 0, **A** is not altered.

Example:

```
Before:  A      ' IF (.NOT.'
         ALEN    9
         B      'WHILE (X.LT.Y) I = I+1'
         START   7
         LENGTH  8

After:   A      ' IF (.NOT.(X.LT.Y)'
         ALEN   17
```

### 6.5.4 CHTYP (CHaracter TYPE)

In scanning the Flecs source program, the translator must be able to identify the syntactic category to which the characters it encounters belong. For the purposes of the translator, only the following syntactic categories are required. Each type is followed by the internal code used by the translator. The code, 8, is used for signifying end of line.

Letter (1) one of the characters “**A**”, “**B**”, “**Z**” or “**a**”, “**b**”, “**z**”.

Digit (2) one of the characters “**0**”, “**1**”, . . . , “**9**”.

Hyphen (3) the hyphen or minus sign, “**-**”.

Left parenthesis  
 (4) “**(**”

Right parenthesis  
 (5) “**)**”.

Blank (6) the blank or space character or any other white space character such as the ASCII horizontal tab.

Equals (9) “**=**”.

Comma (10) “**,**”.

Other (7) any character not falling in one of the other categories.

The invocation for the function CHTYP is

```
I = CHTYP(CH)
```

where



**CH** (in) is an integer representing the character code for the character to be classified;  
**CHTYP** (out) is an integer in the range 1 through 10 giving the syntactic category of the character according to the scheme described above.

Note that the value of **CH** represents the character as its integer code value, not as a Hollerith value. For example, the space or blank character would be represented by a value of 32 (decimal) 40 (octal) for 7-bit ASCII and 64 (decimal) 40 (hex) for 8-bit EBCDIC.

This function is of course character set dependent and must be rewritten for a new machine.

Examples: (Assuming 32 is the code for space and 49 is the code for “1”.)

```
Before:  CH      32
After:   CHTYP   6
```

```
Before:  CH      49
After:   CHTYP   2
```

### 6.5.5 CPYSTR (CoPY STRing)

This subroutine is used to copy a string without modification into a new location. The invocation is

```
CALL CPYSTR(A,ALEN,B)
```

where

**A** (out) is a varying length string;  
**ALEN** (out) is the length of **A**;  
**B** (in) is a string.

The string **A** is set equal to the string **B**. The string **B** is not disturbed.

Example:

```
Before:  B      'GO TO 97781'
After:   A      'GO TO 97781'
        ALEN  11
```

### 6.5.6 CPYSUB (CoPY SUBstring)

This subroutine is used to create a string which is equal to a substring of a second string. The invocation is

```
CALL CPYSUB(A,ALEN,B,START,LENGTH)
```

where

**A** (out) is a varying length string;  
**ALEN** (out) is the length of **A**;  
**B** (in) is a string;  
**START** (in) is a positive integer representing the position of the first character of the substring of **B** to be copied;  
**LENGTH** (in) is the length of the substring to be copied.

The string **A** is set equal to the substring of **B** of length **LENGTH** starting with the character position **START**. The initial value of **A** is ignored. The length of the string **A** becomes **LENGTH**, including the case where **LENGTH** = 0.

Example:

```

Before:  B =      '      IF (X.LT.Y)'
          START = 9
          LENGTH = 8

After:   A =      '(X.LT.Y)'
          ALLEN = 8

```

### 6.5.7 HASH (HASH function)

This integer function subprogram is used by the translator to compute a hash code from a string. The code is used in maintaining hash coded symbol tables. The invocation is

```
I = HASH(A,PRIME)
```

where

A (in) is a string;

PRIME (in) is a positive prime integer;

HASH (out) is an integer in the range 0 through PRIME-1 which has been computed by hashing the character string A.

Example: (An example is somewhat meaningless, but here's one anyway.)

```

Before:  A      'PERFORM-INITIALIZATION'
          PRIME  53

After:   HASH   11

```

### 6.5.8 MAKEST (MAKE STring)

In the original version of the Flecs translator, integer arrays were used to store all character strings. In order to simplify the conversion of Flecs to use Fortran-77 character strings, the old array names were preserved as were their effective lengths. MAKEST is used to copy the string constant to the string variable holding it, and it verifies that the length matches correctly if the preprocessor variable, DEBUG, is turned on. It is used only in the initialization phase of the translator. The invocation is

```
CALL MAKEST(String,ST)
```

where

String (out) is the copy of string made by MAKEST;

ST (in) is the string being copied.

Example:

```

Before:  ST =      'WHEN'
After:   STRING = 'WHEN'

```

### 6.5.9 PUTNUM (PUT NUMber)

This subroutine is used to place the decimal representation of an integer at the beginning of a string which has already been created. The invocation is

```
CALL PUTNUM(A,N)
```

where

A (in/out) is a string;

N (in) is an integer in the range 0 through 99999.

The decimal character representation of the value of `N`, padded if necessary with leading zeroes to a length of five characters, replaces the first five characters on the string `A`. The length of the string is not checked or altered by this process. Only the first five characters of the string are altered.

Examples:

```
Before:  A = '      CONTINUE'
         N = 740
After:   A = '00740 CONTINUE'

Before:  A = 'HI MOM'
         N = 91983
After:   A = '91983M'
```

### 6.5.10 STREQ (STRing Equality)

This logical function is used to test the equality of two strings. The comparison is made without regard to upper or lower case. The invocation is

```
CALL STREQ(A,B)
```

where

`A` (in) is a string;

`B` (in) is a string;

`STREQ` (out) is a logical value which is set `.TRUE.` if the two strings are identical in length and content and `.FALSE.` otherwise. Case is ignored.

Examples

```
Before:  A = 'REPEAT'
         B = 'repeat'

After:   STREQ = .TRUE.

Before:  A = 'UNTIL'
         B = 'UNTILL'

After:   STREQ = .FALSE.
```

### 6.5.11 STRLT (STRing Less Than)

This logical function is used to determine whether or not one string is lexicographically less than another. The comparison is made without regard to upper or lower case. The invocation is

```
L = STRLT(A,B)
```

where

`A` (in) is a string;

`B` (in) is a string;

`STRLT` (out) is a logical value which is set to `.TRUE.` if the character string `A` is lexicographically strictly less than the character string `B`. The collating sequence of the underlying character set is used. Case is ignored.

Examples:

```
Before:  A =      'CANCEL-THE-ORDER'
         B =      'CANCEL-WORK-REQUEST'
```

```
After:   STRLT = .TRUE.
```

```
Before:  A =      'CANCEL-THE-ORDER'
         B =      'CANCEL-THE-ORDER'
```

```
After:   STRLT = .FALSE.
```

```
Before:  A =      'CANCEL-WORK-REQUEST'
         B =      'cancel-the-order'
```

```
After:   STRLT = .FALSE.
```

### 6.5.12 STRUP (STRing UPpercase)

All comparisons in the translator are done in a case insensitive manner. This is achieved by converting all alphabetic characters to upper case before comparison. This conversion is done by STRUP.

The invocation of STRUP is

```
CALL STRUP(A)
```

where

**A** (in/out) is a character string whose lower case alphabetic characters are to be converted to upper case.

Example:

```
Before:  A = 'This is a test'
After:   A = 'THIS IS A TEST'
```

### 6.5.13 TRIM (TRIM string of blanks)

TRIM removes the trailing blanks off of a string. This is done by reducing the length until a non-blank character is found or the beginning of the string is reached. No characters in the string are changed.

The invocation of TRIM is

```
CALL TRIM(A,ALEN)
```

where

**A** (in) is a varying length character string;

**ALEN** (in/out) is its length.

Example:

```
Before:  A      = 'GO TO 90      '
         ALEN = 12
After:   ALEN = 8
```

## 6.6 I/O Interface

The Flecs translator communicates with its environment through five subroutines; *OPENF*, *GET*, *PUT*, *CLOSEF*, and *FLSTOP*. Briefly, these subroutines correspond to determining if a file is to be processed and initializing it if it is (*OPENF*); reading in Flecs source statements (*GET*); writing translated Fortran statements, listing lines and error messages (*PUT*); performing any file closing actions necessary after processing of a file (*CLOSEF*); and terminating execution (*FLSTOP*). By supplying an appropriate version of these subroutines, the Flecs translator may be adapted to a wide variety of batch or time sharing environments without altering the logic of the translator itself. The following discussion is based on the assumption that the implementor does not wish to alter the logic of the translator. Hence, the phrase “must not...” should be taken to mean “must not, unless you wish to alter the translator...”

The following sections discuss various topics of interest to programmers who wish to write their own I/O subroutines.

### 6.6.1 Files and Devices

The basic task of the Flecs translator is to read an input file containing one or more Flecs subprograms and to produce two output files; one containing the translated Fortran version of the subprograms and the other containing the indented listing of the Flecs subprograms together with any error messages, and cross reference tables. Let us refer to these files as follows:

*Flecs/in*     the file containing the Flecs source.

*Fort/out*    the file containing the Fortran translation.

*List/out*    the file containing the Flecs listing.

In a simple batch processing environment, these three files might be the only files to which the translator has access. The Flecs system would open the *Flecs/in* file, process it to produce the *Fort/out* and *List/out* files, close these three files and then terminate execution. In a more ambitious Flecs implementation, one might wish to process several different *Flecs/in* files and produce the corresponding output files before terminating execution. In the latter case, one might expect to find a control file containing information which informs the Flecs system of the number of *Flecs/in* files to be processed and indicates how these files should be accessed. One might also wish to produce an annotated copy of the control file for inclusion in the printed output. Let us refer to these two additional files as follows:

*Control/in*                    the file containing control information.

*Control/out*                   the file containing an annotated listing of the control information.

In some cases it would be desirable to produce only one listing file in which case *List/out* and *Control/out* would actually be the same file although they would correspond to logically different functions.

In a time sharing environment, the basic three files; *Flecs/in*, *Fort/out*, and *List/out*; would still exist. The function of the *Control/in* and *Control/out* files might be replaced by an interactive dialogue via the time sharing terminal. In addition, it is common practice for error messages produced by the translation process to be routed to the terminal in addition to *List/out*, so that they may be brought to the immediate attention of the user.

### 6.6.2 Classes of Input/Output

The Flecs translator proper has no direct knowledge of the *Control/in* or *Control/out* files or of a time sharing terminal. All communication with these files or devices is under control of the

I/O subroutines. The translator program itself is concerned with four classes of I/O. One of these classes is input of Flecs source lines. The other three are output classes. In the discussion to follow it will prove useful to have names for these four classes as given below.

<i>Flecs</i>	This is the only input class and corresponds to the current <i>Flecs/in</i> file.
<i>Fort</i>	This is an output class which corresponds to the current <i>Fort/out</i> file.
<i>List</i>	This is an output class which corresponds to the current <i>List/out</i> file and is used for all lines of the listing not involving errors.
<i>err</i>	this is an output class used when errors are detected. It always corresponds to the <i>List/out</i> file and may also correspond to the user's time sharing terminal or to <i>Control/out</i> if errors are to be produced there as well as on the listing.

When the Flecs translator wishes to read a line of class *Flecs*, it uses the subroutine **GET**. When it wishes to write a line of output in any of the remaining three classes, it calls **PUT** and informs **PUT** of the desired class of output.

### 6.6.2.1 Class *Flecs*

The Flecs translator expects the strings given it as the result of a call to **GET** to be in a standard format as follows:

1. Columns 73-80 are not included in the string. The information in these columns may not be passed through the translator. (With the possible exception of line number information. see Section 6.6.3 [Line Numbers], page 43) If it is desired to have some or all of the information in these columns appear on various output lines, the information must pass directly from **GET** to **PUT**, via **COMMON** perhaps, and appended to the output lines as they are written. In this case **GET** and **PUT** can use the fact that the translator passes line numbers to correlate the input stream with the output streams.
2. Trailing blanks are to be trimmed from the string before it is given to the translator. The logic of the translator is sensitive to this point.
3. Blank lines are permitted and should be presented to the translator as a string of length 0. They appear on *List/out* but not in *Fort/out*.
4. The conventions for comment cards, the statement number field and the continuation column are as in ANSI Standard Fortran. That is, a comment is designated by a “C” in column 1. The statement number must appear in columns 1 through 5. The continuation character must appear in column 6 and the statement must begin in column 7 or later. If special control characters and conventions are used at the installation, the lines must be reformatted before being presented to the translator for processing. For example, the PDP-10 uses the ASCII horizontal tab character to allow the programmer to tab over to the continuation or statement field. The PDP-10 version of **GET** replaces these tab characters by an appropriate number of blanks. See also note 7 below concerning uses of columns 1 through 6.
5. The character set used in columns 7-72 may be highly installation dependent. In particular, horizontal tabs may appear in these columns. The logical dependency of the translator upon the character set is discussed in Section 6.3 [Character String Conventions], page 32, and Section 6.5.4 [CHTYP], page 36.
6. The Flecs translator expects only one statement (or continuation thereof) per line. Installations which employ multiple statement per line conventions may pass such lines through the Flecs translator provided they contain no Flecs constructs. They will be treated by the Flecs translator as a single Fortran statement and will be passed directly to classes *Fort* and *List*.
7. Installations which employ non-standard uses of column 1 through 6 (e.g. , D for Debug or D for double precision, etc.) may pass such statements through the Flecs translator without rewriting its logic. This is so because the translator examines each line to see if it is a comment

or a recognizable Flecs construct. If it is neither, the translator assumes it is a valid line of Fortran and passes it on to the compiler unaltered. Columns 7 through 72 of such lines will be indented on the listing. Since the translator stops scanning as soon as it finds a non-standard use of columns 1 through 6, it is not possible to apply non-standard uses of columns 1 through 6 to lines containing Flecs constructs.

If form feed or new page control characters are sensed in *Flecs/in*, they may be passed directly to *List/out* and *Fort/out* as appropriate. They should not be presented to the translator.

### 6.6.2.2 Class *Fort*

The lines presented by the translator for output in this class are either direct copies of input lines presented to the translator as class *Flecs*, standard ANSI FORTRAN statements, or IF statements containing an expression which has been generated in part from expressions supplied on an input line. Flecs occasionally generates a Fortran statement internally which extends beyond col 72. When this occurs the statement is broken at columns 72-73 and the remainder is continued on the next line using a “1” in column 6.

### 6.6.2.3 Class *List*

Class *List* output is intended to be used to create a listing of the Flecs program for human consumption. In particular, carriage control characters should be supplied by PUT if this is appropriate. (Normally, single spacing will be used, but see the comment above about new page control under Class *Flecs* and the comment below about overprinting. ) In installations supporting line numbered files this file would normally not be line numbered. The line numbers supplied by the translator appear on many of the lines of the listing but are present as textual characters, not as true line numbers.

The strings presented for output under this class normally appear in *List/out* beginning in column 7. Column 6 contains a blank and columns 1-5 contain one of the following three types of character strings.

1. 5 blanks.
2. The line number supplied by the translator.
3. The line number supplied by the translator overprinted by 5 hyphens (minus signs).

See Section 1.4 [Indentation], page 8, for examples of how these three options are used in the listing of a program.

The translator informs PUT of the option desired for each string via one of the arguments to PUT.

If for some reason, overprinting is not possible, it may be omitted but an alternate convention for graphically indicating an ignored line should be adopted.

### 6.6.2.4 Class *Err*

This class is identical to class *List* except that the lines go not only to *List/out* but also to the user's time sharing console or to *Control/out* if these exist.

## 6.6.3 Line Numbers

The Flecs translator expects every line of class *Flecs* supplied to it by the subroutine GET to be associated with a line number in the range 1 to 99999. The line number is placed to the left of the line on the listing produced by the translator and is associated with all error messages and Fortran code produced which correspond with that input line. In addition, the second column of numbers in the listing specifies the line count in the Fortran output file. In this way, the Flecs source lines may be correlated with error messages from the Fortran compiler provided the compiler reproduces the line numbers in its error messages.

The actual values of the line numbers are of no importance to the logic of the Flecs translator as long as they are in the range 1 to 99999. (The value zero is used for special purposes within the translator.) The translator simply copies these numbers from one place to another. Although convention dictates that the line numbers be increasing in value, the translator does not check for this and will not be bothered by non-increasing, duplicate, decreasing, or random ordering. Such orderings would tend, of course, to destroy the utility of the line number as a means of relating the Flecs and Fortran sources.

The line number may be obtained by **GET** in one of at least four ways. The choice of the method to be used is up to the implementor.

1. Many time sharing installations have a specially formatted type of file called a line numbered file where each line comes ready equipped with a five digit line number. These line numbers are useful in connection with certain text editors. If *Flecs/in* is such a file, then the line number can and should be obtained from the file.
2. If the file being read is not a line numbered file in the sense of (1) above, but does contain 80 column card images with five digit user-supplied sequence numbers appearing in, for instance, columns 76-80, then these numbers should be used as the line numbers.
3. Some implementation dependent scheme other than the above may be chosen to allow the user to number his source lines.
4. If the line being read does not come equipped with a user or system supplied line number, **GET** could supply a constant value as the line number. This would, however, totally defeat the utility of the line number as described in Section 1.2 [Correlation of Flecs and Fortran Sources], page 6. A better scheme is to have **GET** increment the last line number it supplied to the Flecs translator by a constant each time a line is read.

A combination of the above methods is often useful. For example, the PDP-10 installation uses the line number if a line number file is supplied. Otherwise, it creates its own numbers by incrementing. A card image oriented batch system might well use columns 76-80 whenever these contain digits, otherwise increment the last number supplied. In this way, unnumbered cards could be inserted into a numbered deck and would receive reasonable sequential numbers since the last card with a number would serve as a basis for the numbering of the following blank cards.

The Flecs translator associates the line number supplied by **GET** during input with all Fortran statements generated as a result. Since one line of input may produce several lines of output, the same number may appear more than once in the output. The actual placement of these line numbers is under control of **PUT**. Normally, duplicate line numbers which appear on output will cause no problems. In time sharing installations which have line numbered files and Fortran compilers which list line numbers along with errors, it is desirable to have *Fort/out* be a line numbered file. The file thus created will contain duplicate line numbers. This may or may not be noticed by the compiler. On the PDP-10, for instance, the F40 compiler is insensitive to duplicate line numbers. If the compiler objects to duplicate line numbers, or if the installation does not support duplicate line numbers, the line numbers supplied to **PUT** may be placed in columns 76-80. Hopefully, the numbers will find their way into the compiler's error messages from there. An alternative is to have **PUT** check outgoing line numbers for failure to be sequential and reassign them if necessary. This will however defeat the strict correspondence between listing and Fortran source numbers.

## 6.7 I/O Subroutines

The five subroutines described in this chapter are interrelated and may actually be five entry points to one subprogram or may communicate information among themselves via labeled **COMMON**. It is assumed the reader is already familiar the I/O interface.



### 6.7.1 OPENF (OPEN Files)

This subroutine is called by the translator to determine whether or not to process a file of Flecs source code and to perform any file initialization necessary in case a file is to be processed. It may also initialize data values available to GET, PUT, and CLOSEF such as initial default line numbers and Fortran I/O unit numbers.

The subroutine is invoked by

```
CALL OPENF(CALLNO,DONE,SVER)
```

where the arguments are as follows:

- |        |  |
|--------|--|
| CALLNO | (in) is an integer supplied by the translator to inform OPENF of the number of times it has been called during this execution of the translator. The first time OPENF is called this value will be 1. On each successive call the value will increase by one. The utility of this argument is discussed below. |
| DONE   | (out) is a logical value to be set by OPENF each time it is called. A value of .TRUE. indicates that there are no more input files to be translated, while .FALSE. indicates that another (or the first) input file is to be translated.   |
| SVER   | (in) is a string containing the Flecs revision number which is provided to OPENF so that it may be passed on to the user of Flecs via <i>Control/out</i> or via a time sharing terminal, if desired.   |

In a simple batch processing situation where *Flecs/in*, *Fort/out*, and *List/out* always correspond to fixed unit numbers which require no opening action and where multiple input files are never considered, the following routine would be adequate since no initialization is required.

```
SUBROUTINE OPENF(CALLNO,DONE,SVER)
  INTEGER CALLNO
  LOGICAL DONE
  WHEN (CALLNO.EQ.1) DONE = .FALSE.
  ELSE DONE = .TRUE.
END
```

In a batch processing situation where multiple files may be processed under control of *Control/in* the following sketch might be appropriate.

```

SUBROUTINE OPENF(CALLNO,DONE,SVER)

***declarations go here ***

IF (CALLNO.EQ.1)
.  INITIALIZE-CONTROL-IN
.  INITIALIZE-CONTROL-OUT
...FIN
REPEAT UNTIL (READY)
.  ATTEMPT-TO-READ-NEXT-FILE-SPECIFICATION
.  CONDITIONAL
.  .  (ENDFIL)
.  .  .  WRITE-FINAL-MESSAGE-TO-CONTROL-OUT
.  .  .  CLOSE-CONTROL-IN
.  .  .  CLOSE-CONTROL-OUT
.  .  .  READY = .TRUE.
.  .  .  DONE = .TRUE.
.  .  ...FIN
.  .  (BADSPECS)
.  .  .  WRITE-BAD-FILE-SPECS-TO-CONTROL-OUT
.  .  .  READY = .FALSE.
.  .  ...FIN
.  .  (OTHERWISE)
.  .  .  OPEN-APPROPRIATE-FILES
.  .  .  INITIALIZE-SUBROUTINES-GET-AND-PUT
.  .  .  READY = .TRUE.
.  .  .  DONE = .FALSE.
.  .  ...FIN
.  ...FIN
...FIN

*** internal procedure definitions go here ***

END

```

In a time sharing environment where the user is queried about which files he wishes to translate, the following sketch might be appropriate:

```

SUBROUTINE OPENF(CALLNO,DONE,SVER)

  ***declarations go here ***

  IF (CALLNO .EQ. 1) TYPE-INTRODUCTORY-MESSAGE
  READY = .FALSE.
  REPEAT UNTIL (READY)
  .  GET-FILE-NAME-OR-QUIT-RESPONSE-FROM-USER
  .  WHEN (QUIT)
  .    TYPE-GOODBYE-MESSAGE
  .    READY = .TRUE.
  .    DONE = .TRUE.
  .    ...FIN
  .  ELSE
  .    CHECK-VALIDITY-AND-EXISTENCE-OF-FILE
  .    CONDITIONAL
  .    . (INVALID)
  .    . . TYPE-BAD-NAME-MESSAGE
  .    . . READY = .FALSE.
  .    . . ...FIN
  .    . (NOEXIST)
  .    . . TYPE-CANNOT-FIND-FILE
  .    . . READY = .FALSE.
  .    . . ...FIN
  .    . (OTHERWISE)
  .    . . OPEN-FLECS-IN
  .    . . OPEN-FORT/OUT
  .    . . OPEN-LIST/OUT
  .    . . INITIALIZE-SUBROUTINES-GET-AND-PUT
  .    . . DONE = .FALSE.
  .    . . READY = .FALSE.
  .    . . ...FIN
  .    . ...FIN
  .    ...FIN
  ...FIN

  ***procedure declarations go here ***

END

```

If OPENF returns a value of DONE = .TRUE. to the translator, then the translator will make no further calls to the I/O subroutines and will execute a call to FLSTOP which will return to operating system, see Section 6.7.5 [FLSTOP], page 49. It may be desirable in some circumstances to have OPENF terminate execution of the translator directly. See the discussion below of how CLOSEF may be used to initiate a Fortran compilation following processing of a file. If multiple files are to be processed then it may be desirable for OPENF to initiate the Fortran compilations after all the Flecs files have been translated. OPENF may also analyze error information accumulated by CLOSEF and make special efforts to inform the operating system if severe errors have been encountered. For example, under OS on the IBM 360/370, a condition code can be used to block a following jobstep in which Fortran compilation was to follow.

### 6.7.2 GET (GET input)

This subroutine is called by the Flecs translator each time it desires to read a line of Flecs source from *Flecs/in*. The invocation is

```
CALL GET(LINENO,STRING,ENDFIL)
```

where

- LINENO** (in/out) is an integer variable to be set to an appropriate line number value by **GET**, see Section 6.7.2 [GET], page 47. The translator initializes **LINENO** to zero before processing a file. There after it does not alter the value of **LINENO**. Thus, except for the first call, the value of **LINENO** upon entry to **GET** is the number assigned to the previous line.
- STRING** (out) is a string of capacity 72 which is to be set equal to the contents of the next line from *Flecs/in*. (See discussion of class *Flecs* in Section 6.6.2.1 [Class Flecs], page 42.)
- ENDFIL** (in/out) is a logical variable which is always **.FALSE.** upon entry to **GET**. It is to be set to **.TRUE.** upon encountering an end-of-file condition in *Flecs/in*. When **GET** returns with a value of **ENDFIL = .TRUE.**, the translator ignores the values of **LINENO** and **STRING** which are returned. The translator contains a loop which is terminated only by the setting of **ENDFIL = .TRUE.** by **GET**.

Most of the details concerning the requirements imposed upon **GET** have been covered in preceding sections, especially in Section 6.6.3 [Line Numbers], page 43, and Section 6.6.2 [Classes of Input/Output], page 41. The implementor may consult the Flecs version of **GET** supplied with the standard Flecs implementation to see an example of the logic involved.

### 6.7.3 PUT (PUT out strings)

This subroutine is called by the Flecs translator each time it desires to output a line of information in one of the three classes *Fort*, *List*, or *Err*. The invocation is

```
CALL PUT(LINENO,STRING,IOCLAS)
```

where

- LINENO** (in) is an integer value supplied to **PUT** by the translator. The value of this integer determines how the line number field is to be displayed in the output. For output of class *Fort*, the value will always be in the range 1 to 99999 and should be included in the output in the manner chosen by the installation. (See discussion of Line Numbers in Section 6.6.3 [Line Numbers], page 43.) For output of classes *List* and *Err*, the value may be positive, zero, or negative and is used both to represent the value of the line number and to indicate which of the three formats is to be used. Specifically:
- A zero value indicates that no line number should be displayed, that is, columns 1-5 should be blank.
  - A positive value indicates that the line number should be displayed in columns 1-5 as a five digit number.
  - A negative value indicates that its absolute value should be displayed in columns 1-5 as a line number but should be overprinted with hyphens.
- STRING** (in) is a string supplied by the translator which represents the characters to be displayed beginning in column 7 of the listing.
- IOCLAS** (in) is an integer value supplied by the translator indicating which output class is to be used. The following codes are used:
- |   |                        |
|---|------------------------|
| 1 | represents <i>Fort</i> |
| 2 | represents <i>List</i> |
| 3 | represents <i>Err</i>  |

### 6.7.4 CLOSEF (CLOSE Files)

This subroutine is called by the Flecs translator after processing of all subprograms in an input file have been completed. It is to perform any necessary file closing actions and will then usually return control to the translator. The invocation is

CALL CLOSEF(MINCNT,MAJCNT)

where

- MINCNT      (in) (MINor error CouNT) is an integer supplied by the translator which indicates how many minor errors were detected by the translator in processing the file.
- MAJCNT      (in) (MAJor error CouNT) is similar to MINCNT but counts major errors. A value of zero indicates no errors. The special value -1 is used to indicate a symbol table overflow in the translator. In this case, CLOSEF may terminate the run or return to the translator which will immediately terminate the run. (See procedure GIVE-UP in the translator.)

The error count information is provided so that it may be included as a message to the user via *Control/out* and/or *List/out* if desired and also so that it may be used to control conditionally the initiation of a Fortran compilation. Excess FINs anywhere in the program or missing FINs prior to a TO or END are minor errors. All other context and all syntax errors are considered major errors.

In a simple batch processing system where *Flecs/in*, *Fort/out*, and *List/out* always correspond to fixed unit numbers, where only one file will be processed on any single run of the translator and where closing of files is not necessary (or is done by the system), the CLOSEF routine can consist simply of a single RETURN statement.

If the Flecs installation has been set up to allow automatic Fortran compilation to follow Flecs translation and if only one file per run of the translator is desired, then CLOSEF can issue the appropriate system calls to initiate such a compilation. The initiation of compilation can also be made conditional upon the values of MINCNT and MAJCNT.

If more than one file is to be processed per run of the translator and automatic Fortran compilation is also desired, then there are at least two options. The operating system may allow control to be transferred to the Fortran compiler from CLOSEF and be returned upon completion. In the more likely event that this is not possible, CLOSEF may stack the error counts away in a common block and the initiation of the Fortran compilations for all the files may be performed by OPENF when it determines that no more files are to be processed.

### 6.7.5 FLSTOP (FLecs STOP)

Termination of the translator is performed by FLSTOP. Depending on the operating system, it will issue an appropriate EXIT call that will transmit to the operating system the success or failure of the translation. It is invoked as follows:

CALL FLSTOP(GMINER,GMAJER)

where

- GMINER      is the total number of minor errors seen by the translator.
- GMAJER      is the total number of major errors seen by the translator.



## 7 Installation and Modification

The process of installing and modifying the Flecs system is straightforward given the fact that Flecs is written in itself, and must be bootstrapped in order to install it on a new machine. The distribution contains several different Fortran translations of the translator in the hope that one of them will compile and link on new machines.

There are very few machine dependencies in the Flecs translator. The most significant is the scanning of the command line. Many machines have Fortran functions, `IARGC` and `GETARG`, which permit the command line to be read. However, for those machines that do not, there is a dummy version of these subroutines in the file, `'argc.f'`, which will result in Flecs reading from standard input to get the files to translate. There is a shell script, `'flecs.sh'`, which can be used to emulate the behavior of reading from the command line.

The makefile for Flecs attempts to provide a number of different targets for building pieces of the translator. However, it will be necessary to review its operations and do some of them “by hand” on some new machines.

The normal compilation of Flecs depends upon using a C Preprocessor in order to select machine dependent statements. If you do not have one specially geared for Fortran, then you can use the one associated with the C compiler. A Fortran C Preprocessor is available for free upon request from Robert E. Brucoleri.

The installation procedure for new machines consists of the following three phases:

1. Examining the standard version of the translator.
2. Creating an initial working version of the translator;
3. Making any further modifications deemed desirable.

These three phases are discussed below.

Note: In going to a machine on which the short form of assigned `GO TO` is not valid, it is vital that the Fortran version of the translator being moved has itself been translated using the variable, `COGOTO`, being set to `.TRUE.`, see Section 6.4 [Translation Parameters], page 32.

### 7.1 The Standard Version

The materials supplied in a distribution of the standard Flecs system consist of a manual for using and maintaining the Flecs translator and the following machine readable files stored in an appropriate archive format. Note that on VMS systems, files with extension, `'f'`, would have an extension of `'for'`.

`'descrip.mms'`

A `'Makefile'` suitable for use with the VAX/VMS layered product, MMS, the Module Management System. It has most of the functionality of the Unix `'Makefile'`.

`'flclean.com'`

A VMS procedure for cleaning up after Flecs files are translated and compiled.

`'flecs.flx'`

Flecs source code for the main program of the translator.

`'flecs.f'`

Fortran source code for the main program part of the translator. This file is automatically generated by the translator from `'flecs.flx'`.

`'flecs.texinfo'`

Texinfo source for this manual.

`'flecs.txt'`

Outline of a short course on Flecs.

- `'flecsubs.flx'`  
Flecs subroutine library. Most machine dependencies are found in here.
- `'flecsubs.f'`  
Fortran translation of `'flecsubs.flx'`.
- `'fleensp.f'`  
A more portable, but less efficient version of the translator.
- `'flecsbsp.f'`  
A more portable, but less efficient version of the Flecs subroutine library.
- `'flecsdoc.texinfo'`  
Texinfo source for this manual. Texinfo is a language for specifying TeX manuals and GNU Emacs Info files, see section “Overview” in *The GNU Texinfo Manual*.
- `'flecsdoc.dvi'`  
A T<sub>E</sub>X DVI file for this manual.
- `'flecsdoc.ps'`  
A Postscript file for this manual.
- `'flecsdoc, flecsdoc-1, flecsdoc-2, flecsdoc-3'`  
Info files for Flecs. These can be installed in the `'info'` directory of your GNU Emacs files.
- `'flifix.flx'`  
A program to add Fortran listing line numbers to a Flecs listing. This program was written only for VAX/VMS, and is currently broken. Since FLECS now outputs the sequential numbering of output Fortran lines, `'flifix'` is less necessary.
- `'makefile.gen'`  
Predecessor makefile for Flecs. Combine this file with one of the `'machine.make'` files to get the correct macros for the compilers and other installation parameters.
- `'*.make'` A number of macro definitions for different machines are provided.
- `'pflecs.f'`  
An assemblage of `'fleensp.f'`, `'fleensp.f'`, and `'argc.f'` that can be compiled to make a version of Flecs that does not read the command line for the files it will translate.
- `'readme.txt'`  
A short description of Flecs

## 7.2 The Initial Working Version

Having obtained the standard version of the Flecs system, the next phase of the installation procedure is to create an initial working version of the Flecs system which runs on the local computer. Until this initial version is running, all modifications to the system source must be made in duplicate, once to the Fortran source and once to the Flecs source. (Except, of course, if the implementor has access to a working Flecs translator on another machine.) The initial translator system can then be used to produce automatically the corresponding Fortran source. Therefore, in this phase only those modifications absolutely necessary to produce a working translator are to be performed. Modifications related to efficiency or ease of use can be deferred to the next phase. The procedure described below is designed to guide the programmer in obtaining a working version of the translator as quickly as possible.



## 7.3 Procedure for Creating the Initial Version

Please read the entire procedure through before beginning.

1. Make a copy of the machine readable files provided. We shall refer these as the reference set.
2. Read Chapter 6 [Flecs Implementation], page 31, and Section 6.6 [I/O Interface], page 41, for a description of the implementation of the translator.
3. Review the makefiles, 'makefile' or 'descrip.mms', and modify them as required by your installation.
4. Compile 'flecs.f' and 'flecsubs.f', and link together.
5. If those compilations fail because of the assigned GO TO's, or because the line numbers are too large, then copy 'flecsp.f' to 'flecs.f', and 'flecsbsp.f' to 'flecsubs.f', and try again.
6. If these compilations fail, modify the translator as necessary to get it running. Keep a record of all changes made.
7. Uses 'flecs.flx' and 'flecsubs.flx' as test cases for your translator. Unless the translation parameters have been changed, your local version of the translator should be able to reproduce the reference copies of the translations.
8. Edit 'flecs.flx' and 'flecsubs.flx' to reflect any changes you made to 'flecs.f' and 'flecsubs.f'.
9. Run the translator again on your modified versions of the sources, compile, and link to produce a native translator. The makefile's may be used for this purpose. Verify the operation of the native translator by translating 'flecs.flx' and 'flecsubs.flx' again.
10. Any further modifications of the translator should be made on the '.flx' versions of the files.
11. Install the Flecs translator. The `install` target of the makefiles can be used. On Unix systems, simply copy the translator into a local 'bin' directory that users have on their path. On VMS systems, a foreign command definition for Flecs should be placed in the appropriate 'LOGIN.COM' file.

## 7.4 Documentation

Machine readable documentation for the Flecs translator is provided as part of the package. In order to regenerate files from the 'flecsdoc.texinfo', you will need to install T<sub>E</sub>X, a version of 'texinfo.tex' with a version number greater than or equal to 2.36, 'texindex.tex', and 'makeinfo'. T<sub>E</sub>X is available from a number of sources, and the remaining utilities are available from the Free Software Foundation.

## 7.5 Desirable Modifications

Having obtained the initial working version of a Flecs translator, the programmer may now proceed to make any further changes to the translator deemed desirable. Some of these changes are discussed briefly below.

### 7.5.1 Efficiency

If the operation of the translator is slow compared to Fortran compilation, suitable performance monitoring tools should be employed to find the bottlenecks. Rewriting these routines in C or assembler should help, but short test cases should be used to verify that improvements will really work.

### 7.5.2 Convenience

The convenience of using the translator is primarily a function of how easily accessible it is within the operating system and how little it disrupts the programmer's previously established working

habits. Ideally the Flecs system would be transparent to the programmer, allowing him to believe he was simply using features of the Fortran language which he had never discovered before. In practice much can be done to give the Flecs translator a low profile, but like any preprocessor, it will never become completely transparent. Use of the Unix utility, **make**, or of the VMS utility, **MMS**, can be very helpful.

Some of the problems of preprocessor transparency have been attacked in the design of the system, primarily in the use of line numbers to relate Flecs source to Fortran source and the fact that there is a very close resemblance between the two languages. Other problems can be reduced by appropriate adaptation or modification of the system. Some problems and possibilities for their solutions are sketched below.

The extra processing step represented by the translator of Flecs to Fortran can be annoying if it requires much extra effort on the part of the programmer. Appropriate use of the job control language for the operating system may reduce this burden. In the best case the Flecs translator will appear to be just another compiler available in the operating system and will be invoked in a manner similar to invoking the Fortran compiler. Invoking the Flecs translator should initiate both the Flecs translation and the subsequent Fortran compilation. Such a change might be accomplished by simply establishing appropriate job control procedures in a procedure library or file. See Section 6.6 [I/O Interface], page 41, which contains many suggestions on how modifications of the I/O routines can be used to make the Flecs system blend more smoothly into the system.

The Flecs translator is able to adapt to a wide variety of Fortran dialects simply because it passes most statements through without alterations and for those it does alter, it produces ANSI standard equivalents. Nevertheless some features such as special conventions for columns 1 through 6 or for multiple statements per line will cause problems if used with Flecs constructs. Some of these problems may be circumvented by pre- and post-processing of the source file using **GET** and **PUT**. (See discussion of class *Flecs* in Section 6.6.2 [Classes of Input/Output], page 41.)

Many of the programming aids which are available for use by the Fortran programmer such as cross reference listings and special debugging techniques become clumsy when used with Flecs because of the need to work through the intermediate Fortran listing. No general solution to such problems can be offered. Frequently nice solutions can be obtained only by making modifications to some of the other programs involved. For example, persuading compilers, cross reference listers, and interactive debuggers that they should communicate in line numbers not internal or external statement numbers.

# Index

## A

Ascii Character Set ..... 32

## B

Blank Trimming ..... 40  
Blanks ..... 25  
Body ..... 21

## C

C Preprocessor ..... 32  
Case Conversion ..... 40  
CATNUM (conCATenate NUMber to string) ..... 35  
CATSTR (conCATenate STRing to string) ..... 35  
CATSUB (conCATenate SUBstring to string) ..... 36  
Character Processing Subroutines ..... 34  
Character String Conventions ..... 32  
Character Type Codes ..... 36  
CHSPAC ..... 33  
CHTYP (CHaracter TYPE) ..... 36  
CHZERO ..... 33  
Class *Err* ..... 43  
Class *Flecs* ..... 42  
Class *Fort* ..... 43  
Class *List* ..... 43  
Classes of I/O ..... 41  
CLOSEF (CLOSE Files) ..... 49  
COGOTO ..... 33  
Comments ..... 9, 22, 42  
Comparison of Strings ..... 39  
Concatenation of Strings ..... 35, 36  
CONDITIONAL ..... 14  
Context Errors ..... 27  
Continuation Lines ..... 25, 42  
control phrase ..... 6  
Control Structures ..... 5, 11  
*Control/in* ..... 41  
*Control/out* ..... 41  
Copying Strings ..... 37  
CPASS directive ..... 22  
CPYSTR (CoPY STRing) ..... 37  
CPYSUB (CoPY SUBstring) ..... 37

## D

Decision Structures ..... 11  
'*descrip.mms*' ..... 51  
Devices and Files ..... 41  
DFAKDO directive ..... 22  
Directives, Translator ..... 22  
DO ..... 16, 22  
DO Loops and Internal Procedures ..... 25  
Documentation ..... 53

## E

ELSE ..... 13  
Errors ..... 27, 42, 43  
Exiting ..... 49  
Extended Range DO Loops ..... 25

## F

FAKE ..... 33  
FAKEDO ..... 16  
FAKEDO directive ..... 22  
File Closing ..... 49  
File Input ..... 48  
File Names ..... 29  
File Opening ..... 45  
File Output ..... 48  
Files and Devices ..... 41  
Files in the Standard Version ..... 51  
FIN ..... 7  
'*flclean.com*' ..... 51  
Flecs Command ..... 29  
Flecs Implementation ..... 31  
Flecs Input File ..... 42  
Flecs Statements ..... 11  
'*flecs.f*' ..... 51  
'*flecs.flx*' ..... 51  
'*flecs.texinfo*' ..... 51  
'*flecs.txt*' ..... 51  
*Flecs/in* ..... 41  
'*flecsbsp.f*' ..... 52  
'*flecsdoc*' ..... 52  
'*flecsdoc.dvi*' ..... 52  
'*flecsdoc.ps*' ..... 52  
'*flecsdoc.texinfo*' ..... 52  
'*flecsp.f*' ..... 52  
'*flecsubs.f*' ..... 52  
'*flecsubs.flx*' ..... 52  
'*flifix.flx*' ..... 52  
FLSTOP (FLecs STOP) ..... 49  
FORMAT statements ..... 25  
*Fort/out* ..... 41  
Fortran Compatibility ..... 23  
Fortran Features ..... 5  
Fortran Line Numbers ..... 9, 25  
Fortran Output ..... 42, 43  
Fortran Statement Numbers ..... 34

## G

GET (GET input) ..... 48

## H

HASH (HASH function) ..... 38  
Hash Table ..... 34

**I**

I/O Classes .....	41
I/O Interface .....	41
I/O Subroutines .....	31, 44
IF .....	11
Implementation of Flecs .....	31
Indentation .....	8
Input from Files .....	48
Installation .....	51, 52
Integer Variable Names .....	25
Internal Procedure Table .....	34
Internal Procedures .....	20
Internal Procedures and DO Loops .....	25

**K**

keyword .....	6
---------------	---

**L**

Labels for Fortran Statements .....	34
Line Numbers .....	6, 9, 25, 43
List/out .....	41
Listing .....	8, 42, 43
LONG .....	33
Loop Structures .....	15
LWIDTH .....	33

**M**

'makefile.gen' .....	52
MAKEST (MAKE STring) .....	38
MAXSTK .....	33
Modifications .....	51, 53

**N**

Nesting .....	7
Number Conversion .....	35, 38
Numbers for Fortran Statements .....	34

**O**

OPENF (OPEN Files) .....	45
Organization of the Translator .....	31
OTHERWISE .....	14, 15
OUTCNT .....	34
Output to Files .....	48

**P**

Parenthesized Expressions .....	26
'pflecs.f' .....	52
Portability .....	5
PRIME .....	34
Procedure Parameters .....	22
Procedure Tables .....	34
Procedures, Internal .....	20

Process of Translation .....	5
PUT (PUT out strings) .....	48
PUTNUM (PUT NUMber) .....	38

**R**

'readme.txt' .....	52
Recursion .....	22
REPEAT UNTIL .....	20
REPEAT WHILE .....	18
Reserved Words .....	26

**S**

SAFETY .....	34
scope .....	6
SEEDNO .....	34
SELECT .....	15
Semantic Errors .....	27
SHORT .....	33
Source Preparation .....	29
specification .....	6
Stack Overflow .....	34
Standard Version .....	51
Statement Numbers .....	34
Stopping Execution .....	49
STREQ (STRing EQuality) .....	39
String Case Conversion .....	40
String Comparison .....	39
String Concatenation .....	35, 36
String Copying .....	37
String Processing Conventions .....	32
STRLT (STRing Less Than) .....	39
Structured Statement .....	6
Structures for Control .....	5
STRUP (STRing UPpercase) .....	40
Substring Concatenation .....	36
Substring Copying .....	37
Syntax Errors .....	27
System Structure .....	31

**T**

Testing the Translator .....	32
TO .....	20
TRANSLATE directive .....	23
Translation Parameters .....	32
Translation Process .....	5
Translator Directives .....	22
Translator Operation .....	29
Translator Parameters .....	32
TRIM (TRIM string of blanks) .....	40
Type Codes for Characters .....	36

**U**

UNLESS .....	12
UNTIL .....	19

Usage.....	29
Use of Fortran.....	5

## V

Variable Names.....	25
Varying Length Character Strings.....	32

## W

WHEN...ELSE.....	13
WHILE.....	16
Whitespace.....	25
Writing to Files.....	48

