March, 1988
Order Number: 311571-001

VAST-2

USER'S GUIDE

intel Corporation

VAST-2
User's Guide

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9 (a) (9).

Copyright © 1988 by Intel Scientifc Computers, Beaverton, Oregon. No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BITBUS	i_m	iRMX	OpenNET
COMMputer	iMDDX	iSBC	Plug-A-Bubble
CREDIT	iMMX	iSBX	PROMPT
Data Pipeline	Insite	iSDM	Promware
Genius	int_el	iSXM	QUEST
$\hat{1}$	int_elBOS	KEPROM	QueX
i	int_eligent Identifier	Library Manager	Ripplemode
I^2ICE	int_eligent Programming	MCS	RMX/80
ICE	Intellec	Megachassis	RUPI
iCS	Intellink	MICROMAINFRAME	Seamless
iDBP	iOSP	MULTIBUS	SLD
iDIS	iPDS	MULTICHANNEL	UPI
iLBX	iPSC	MULTIMODULE	

EXOS is a trademark or equipment designator of Excelan, Inc.
XENIX is a trademark of Microsoft Corp.
UNIX is a trademark of AT&T
VAST-2 is a registered trademark of Pacific-Sierra Research Corp.

REV.	REVISION HISTORY	DATE
-001	Original issue	03/15/88

VAST-2
User's Guide

RESTRICTED RIGHTS

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

CONTENTS

CHAPTER 1 - INTRODUCTION

Introduction	1-1
Purpose and Function	1-1
Code Generation For the iPSC-VX	1-1
Performance	1-3
User Interaction	1-3
Capabilities Overview	1-4
Example	1-5
How To Use This Guide	1-8

CHAPTER 2 - INVOKING VAST-2

Introduction	2-1
VAST2 Command	2-1
Examples	2-2

CHAPTER 3 - OPTIONS

Introduction	3-1
Switches	3-2
Type Control Options	3-3
OPTON/OPTOFF Option Switches	3-4
LSTON/LSTOFF Option Switches	3-5

CHAPTER 4 - DIAGNOSTIC MESSAGES

Introduction	4-1
Types of Diagnostic Messages	4-2

Contents

CHAPTER 5 - USER DIRECTIVES

Introduction	5-1
Directive Format	5-2
Directive Summary	5-3
Transformation Directives	5-4
Data Dependency Directives	5-6
Listing Directives	5-7
Other Directives	5-7

CHAPTER 6 - VECTORIZATION OVERVIEW

Introduction	6-1
VAST-2 Generated Names	6-2
Loop Types	6-2
Allowed Statements	6-3
VECTOR/NOVECTOR Directives	6-4
Understanding Loop Variables	6-4

CHAPTER 7 - INDEXING

Introduction	7-1
Constant Increment Integers	7-2
Index Expressions	7-3
Non-Linear Indexing	7-3
Last Value Saving	7-4
Indirect Addressing	7-5
Multiply Defined Indices	7-5
Index Expressions in Several Dimensions	7-6
Explicit Index Use	7-6

CHAPTER 8 - STORING INTO SCALARS

Introduction	8-1
Scalar Promotion	8-2
Carry Around Scalars	8-3
Reduction Functions	8-3

CHAPTER 9 - DATA DEPENDENCY ANALYSIS

Introduction 9-1
Data Dependency Examples 9-2
Reference Reordering 9-4
Ambiguous Subscript Resolution 9-4
Data Dependency Directives 9-5
Split Loop to Avoid Recursion 9-7

CHAPTER 10 - DECISION PROCESSES

Introduction 10-1
Allowable Conditional Constructs 10-2
Kinds of Conditional Statements 10-2
Two Types of Conditions 10-3
Conditional Indexing 10-4

CHAPTER 11 - FUNCTIONS & SUBROUTINES

Introduction 11-1
Statement Functions 11-2
Intrinsic Functions 11-2

CHAPTER 12 - OUTER LOOPS

Introduction 12-1
Choosing The Best Loop 12-2
Select Directive 12-2
Outer Loop Vectorization Inhibitors 12-3
Loop Nest Collapse 12-4

CHAPTER 13 - MISCELLANEOUS TRANSFORMATIONS

Introduction 13-1
Description 13-2

APPENDIX A - VECTORIZATION RULES

APPENDIX B - DIAGNOSTIC MESSAGES

Introduction B-1
Syntax Errors B-2
Data Dependency Conflicts B-3
Translation Diagnostics B-5
  Statement types B-5
  Branches B-7
  External references B-8
  DO statement B-9
  Scalars B-9
  Outer loops B-10
  Miscellaneous B-11
Internal Errors B-12
Directive Errors B-12
Notes B-13

APPENDIX C - GLOSSARY

APPENDIX D - VECTOR ROUTINES

Introduction D-1
Mathematical Primitives D-2
Triads D-3
Relational Primitives D-4
Logical Primitives D-5
Reduction Function Primitives D-6
Intrinsic Function Primitives D-7
Conversion Primitives D-8
Gather/Scatter Primitives D-8
Step Function Primitives D-9
Multi-Line Transformation Primitives D-9
Miscellaneous Routines of VecLib Not Used by VAST-2 D-10

CHAPTER 1

INTRODUCTION

INTRODUCTION

This manual is a guide to the use of the Intel Scientific Computer's iPSC-VX version of VAST-2. (VAST is an acronym for Vector and Array Syntax Translator). This manual is designed for FORTRAN programmers who want to understand VAST-2's capabilities and learn how to use it effectively.

PURPOSE AND FUNCTION

VAST-2 is a pre-compiler which generates code for the iPSC-VX vector processor (see Figure 1-1). It vectorizes DO and IF loops in FORTRAN programs.

When VAST-2 processes a FORTRAN program, it creates two files. One is a listing of the input program with diagnostic comments added to tell which loops were vectorized and which were not. If a loop was not vectorized, VAST-2 explains why it was not. VAST-2 also creates an enhanced version of the input FORTRAN program containing vector code in place of the original DO or IF loops (see Figure 1-2). This file can then be compiled using "rmfort".

CODE GENERATION FOR THE iPSC-VX

VAST-2/iPSC-VX gives assistance only with the vectorization of tasks on individual nodes, not with the partitioning of the whole program into parallel tasks.

For the first release of this product, the vector code produced is in the form of calls to a library of vector subroutines which execute on the vector processor. A later release will create larger blocks of code to execute on the vector processor, thereby reducing overhead significantly. Appendix D shows the vector routines for the iPSC-VX which are supported by VAST-2.

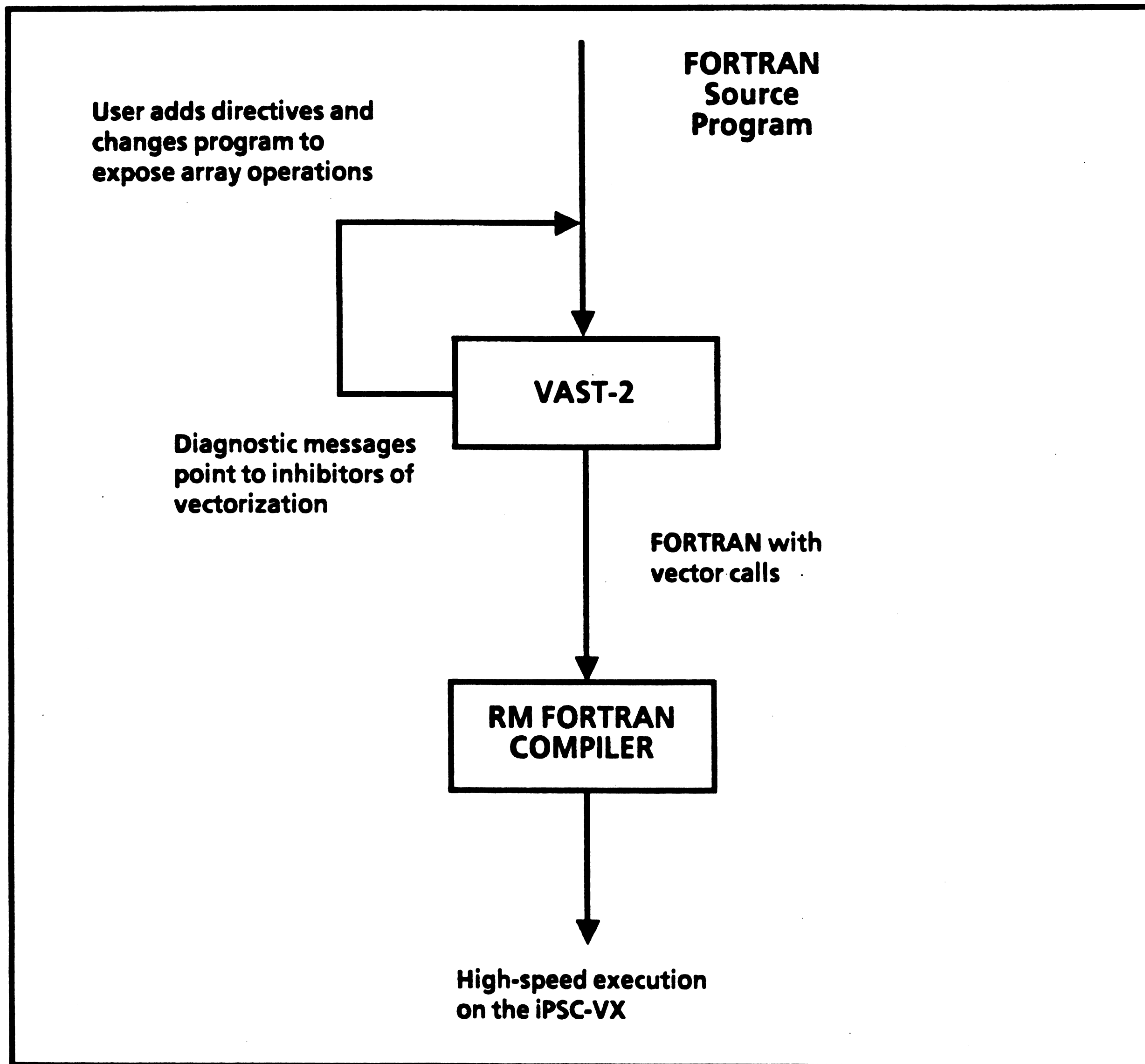

Figure 1-1
Iterating With VAST-2

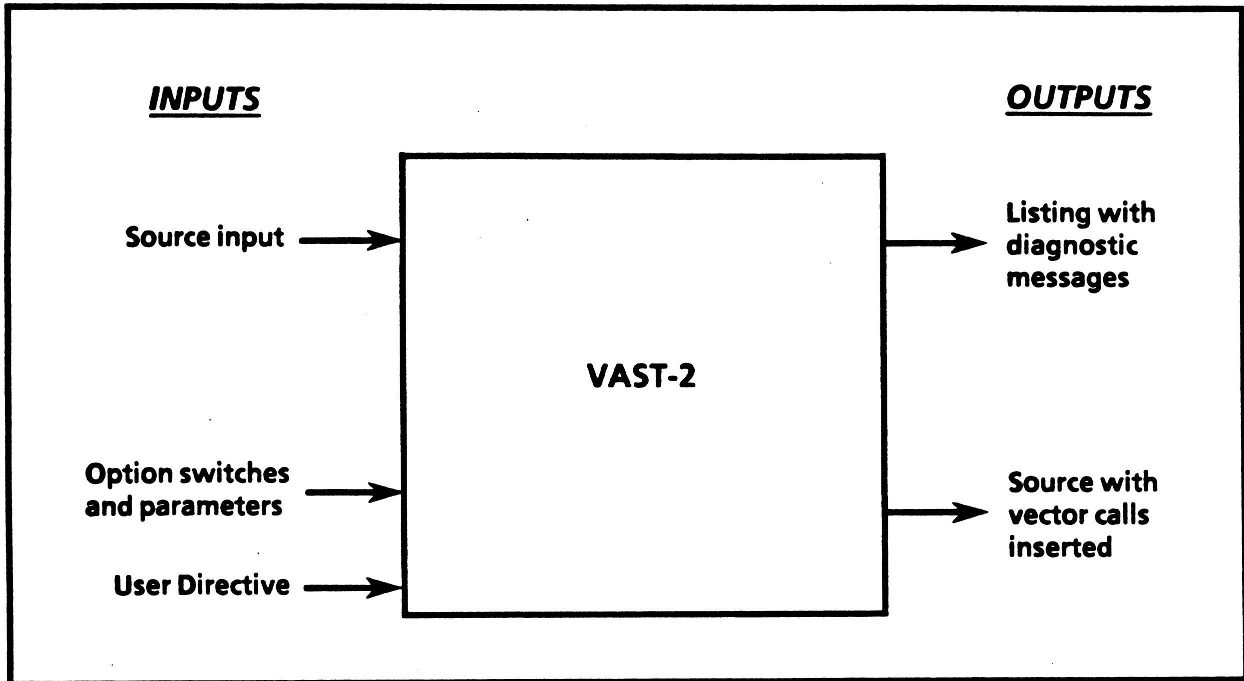

Figure 1-2
VAST-2 Inputs and Outputs

PERFORMANCE

Performance of VAST-generated code is highly program- and compiler-dependent; best results are usually obtained for programs which have been restructured to take advantage of VAST-2's abilities.

USER INTERACTION

It is important to realize that VAST-2 is only a tool; some programmer effort will ordinarily be required to produce an efficient array syntax translation. The degree of effort needed to expose the array operations in an existing program varies greatly, depending on factors such as the nature of the algorithms, the structure of the data, and the programming style.

Only those routines which take up a significant amount of computation time need to be run through VAST-2 and translated. Those loops that VAST-2 has not been able to vectorize can, in many cases, be restructured in a way that is logically equivalent but vectorizable. The diagnostics issued by VAST-2 are intended to aid you in restructuring the original source program to ensure maximum vectorization.

CAPABILITIES OVERVIEW

VAST-2 is a state-of-the-art vectorizing translator. Its features include:

- Vectorization of both DO loops and IF loops.
- Vectorization of outer loops.
- Sophisticated data dependency analysis to insure safe translation of loops.
- Examination of EQUIVALENCE statements to detect hidden recursion.
- Ability to reorder array references to avoid recursion, where possible.
- Use of program information from outside the loop to aid data dependency analysis (ambiguous subscript resolution).
- Ability to minimize data dependent sections of loops so that the maximum amount of computation is vectorized.
- Ability to handle all forward-branching conditional operations, even deeply nested or complicated branches.
- Use of program information from outside the loop to determine whether final values of indices and of scalar temporaries are required.
- Rerolling of "unrolled" loops.
- User directives and switches to control almost all aspects of the translation process.
- Vector common sub-expression elimination.
- Ability to select the best single dimension in a loop nest to vectorize on.
- Ability to collapse multiple dimension loops into one long loop.

In order to do the best job of vectorizing, VAST-2 must communicate with the user. Communication includes both the diagnostics by which VAST-2 sends messages to the programmer and the directives by which you can cause VAST-2 to take certain actions. These features allow you to know exactly what has occurred and to control the translation process.

EXAMPLE

Figure 1-3 shows a VAST-2 listing of a subroutine containing several loops. The input lines are shown numbered consecutively. A summary of the loops which appeared in the subroutine follows the source listing, and is in turn followed by counts of various events that occurred in translating this subroutine ("Event Summary"). The loop summary shows that the first two loops are vectorized, and the third loop is not vectorized because it has data dependency conflicts (explained in Chapter 9). In the loop summary, %CD stands for percent conditional and %DP for percent dependent.

```
                  VAST-2      Version 2.21A5      iPSC-VX

     1.       SUBROUTINE EXMPL(N,J,B,C)
     2.       REAL A(99),B(1),C(1)
     3.  C
     4.       DO 1 I=1,N
     5.     1 A(I)=B(I)+C(I)
     6.  C
     7.       DO 2 I=1,N
     8.     2 IF (C(I).GT.0.0) A(I)=B(I)*2.0
     9.  C
    10.       DO 3 I=1,N
    11.     3 A(I+J)=A(I)*B(I)+C(I)
    12.  C
    13.       RETURN
    14.       END
-------------------------------- EXMPL --------------------------------
 LINE TYPE MSG   ( T=TRANSLATION, D=DEPENDENCY, W=WARNING, S=SYNTAX )

 11  D  POTENTIAL FEEDBACK OF ARRAY ELEMENTS -- USE DIRECTIVE IF OK (A)
        CONFLICT ON LINE 11.  THE DO INDEX IS I, THE DO LABEL IS 3.
-----------------------------------------------------------------------

------------------- LOOP SUMMARY FOR ROUTINE EXMPL ----------------
 LABEL  INDEX  START  END NEST  COMMENT         %CD %DP  ITERATIONS

   1     I      4      5   1   VECTORIZED                    N
   2     I      7      8   1   VECTORIZED       100          N
   3     I     10     11   1   DATA DEPENDENT                N

-------------------- EVENT SUMMARY FOR ROUTINE EXMPL --------------

WARNING MESSAGES          --  0    SYNTAX ERRORS              --  0
TRANSLATION DIAGNOSTICS   --  0    DATA DEPENDENCY CONFLICTS  --  1
INNER LOOPS EXAMINED      --  3    INNER LOOPS VECTORIZED     --  2
```

Figure 1-3
Example VAST-2 Listing

Figure 1-4 (two parts) shows a VAST-2 listing of the same subroutine when LSTON = DN has been specified to show the generated declarations and code. First, the declarations added by VAST-2 are listed, followed by the executable portion of the subroutine, with vectorizable loops replaced by vector code. On the right of each line of generated code is shown the line number of the original line it comes from.

```
                VAST-2      Version 2.21A5      iPSC-VX

     1.         SUBROUTINE EXMPL(N,J,B,C)
     2.         REAL A(99),B(1),C(1)
     3.   C
     4.         DO 1 I=1,N
     5.     1   A(I)=B(I)+C(I)
     6.   C
     7.         DO 2 I=1,N
     8.     2   IF (C(I).GT.0.0) A(I)=B(I)*2.0
     9.   C
    10.         DO 3 I=1,N
    11.     3   A(I+J)=A(I)*B(I)+C(I)
    12.   C
    13.         RETURN
    14.         END

------------------------------ EXMPL  ------------------------------

 LINE TYPE MSG   ( T=TRANSLATION, D=DEPENDENCY, W=WARNING, S=SYNTAX )

 11  D  POTENTIAL FEEDBACK OF ARRAY ELEMENTS -- USE DIRECTIVE IF OK (A)
        CONFLICT ON LINE 11.   THE DO INDEX IS I, THE DO LABEL IS 3.

 ---------------------------------------------------------------------
```

Figure 1-4 - Part 1 of 2
Example VAST-2 Listing

```
C....TRANSLATED BY VAST-2 2.21A5  13:57:48   1/16/87   LSTON=DTN
     REAL R1V(1)
     LOGICAL L1V(1)
     REAL QQQ
     COMMON/VPFAST/ QQQ(4096)
     EQUIVALENCE (QQQ(1),L1V),(QQQ(100),R1V)
         .
         .
 C                                                           3
     CALL SVADD ( N, B(1),1, C(1),1, A(1),1 )                5

 C                                                           7
       CALL SSLT ( N, 0.0, C(1),1, L1V(1),1 )                8
       CALL SSMUL ( N, 2.0, B(1),1, R1V(1),1 )               8
       CALL SCNDST ( N, R1V(1),1, L1V(1),1, A(1),1 )         8

 C                                                           9
     DO 3 I=1,N                                              10
  3  A(I+J)=A(I)*B(I)+C(I)                                   11
 C                                                           12
     RETURN                                                  13
     END                                                     14


 ------------------- LOOP SUMMARY FOR ROUTINE EXMPL   -----------------
  LABEL  INDEX  START  END NEST   COMMENT         %CD %DP  ITERATIONS

     1     I      4     5   1   VECTORIZED                      N
     2     I      7     8   1   VECTORIZED        100           N
     3     I     10    11   1   DATA DEPENDENT                  N


 --------------------   EVENT SUMMARY FOR ROUTINE EXMPL   --------------
 WARNING MESSAGES            --   0    SYNTAX ERRORS              --     0
 TRANSLATION DIAGNOSTICS     --   0    DATA DEPENDENCY CONFLICTS  --     1
 INNER LOOPS EXAMINED        --   3    INNER LOOPS VECTORIZED     --     2
```

Figure 1-4 - Part 2 of 2
Example VAST-2 Listing

HOW TO USE THIS GUIDE

NOTE

For those unfamiliar with vectorization terminology, the Glossary in Appendix C provides definitions of some important terms.

• **Chapter 2**	Describes how to invoke VAST-2.
• **Chapter 3**	Presents the available switches and options that can be given to VAST-2.
• **Chapters 4 & 5**	Discuss communication between VAST-2 and the user: the way VAST-2 reports to the user (diagnostic messages), and ways to guide VAST-2's action (user directives).
• **Chapters 6-11**	Discuss the concepts and rules of vectorization. Examples in these chapters illustrate vectorizable and non-vectorizable loops.
• **Chapter 12**	Discusses the vectorization of outer loops and the choices involved in doing this.
• **Chapter 13**	Covers miscellaneous vectorization features.
• **Appendix A**	Summarizes some of the rules that loops must obey in order to be vectorized.
• **Appendix B**	Displays all the messages that VAST-2 can generate and gives examples of code that can cause each message.
• **Appendix C**	Defines vectorization terminology.
• **Appendix D**	Contains a list of vector routines generated by VAST-2. Use this if you want to understand the output of VAST-2 and the function of the vector routines.

CHAPTER 2

INVOKING VAST-2

INTRODUCTION

This chapter describes the method used to execute VAST-2 on XENIX. It contains a description of the vast2 command followed by three examples of using the command.

VAST2 COMMAND

VAST-2 is executed by the command:

vast2 [-options = string ...] [-o ofile.f] file [... filen]

where:

file	RM FORTRAN source input file	
options	(opton, optoff) (lston, lstoff) (single, double, both) space	(for explanation, see Chapter 3, "Options")
string	switch strings to set for the specified option	(for settings, see the Chapter 3, "Options")
ofile.f	vectorized source output file	if the "-o" option is not specified, a new output file name is created by prefixing the input file with the letter "V".

The listing file is written to the standard output file (normally the terminal.)

EXAMPLES

The following are examples of using the vast2 command:

Example 1

In this example, the FORTRAN program "crunch" is run through VAST-2. The output file will be "Vcrunch.f".

```
vast2 crunch.f
```

Example 2

In this example, it is desired to run "crunch" through VAST-2, save the VAST-2 listing in file "crunch.lst", put the translated code in "crunch.f", and compile the translated code.

```
vast2 -o crunch.f crunch.F > crunch.lst
rmfort crunch.f
```

Example 3

In this example, it is desired to run "myprog" through VAST-2, requesting single precision vector operations only (single), no associative transformations (optoff = a), and no listing of warnings (lstoff = w) or the event summary (lstoff = e). (These options and others are presented in Chapter 3, " Options".)

By default, the output file will be Vmyprog.f.

```
vast2 -single -optoff = a -lstoff = we myprog.f
```

After running a subroutine through VAST-2 and compiling, it must be linked with the appropriate vector library supplied by Intel Scientific Computers (ISC). (Refer to the VX User's Guide.)

CHAPTER 3

OPTIONS

INTRODUCTION

This chapter describes the options that can be used with the vast2 command. It consists of the following sections:

- Switches
- Type Control Options
- OPTON/OPTOFF Option Switches
- LSTON/LSTOFF Listing Option Switches

SWITCHES

Switches specifying global actions can be passed to VAST-2 on the vast2 command or via the "SWITCH" directive.

There are two kinds of switch control:

- control of optimizations (OPTON and OPTOFF keywords)
- control of listing (LSTON and LSTOFF keywords)

For XENIX, switch values are input on the command line in the form:

```
vast2   -opton = ssss  -optoff = tttt  -lston = lll  -lstoff = kkk
        -double  -space = nnnnn  -vpfast = mmmmm  file.f
```

where:

"ssss" and "tttt"	strings of letters defined in Table 3-1
"lll" and "kkk"	strings of letters defined in Table 3-2
"file.f"	the input file; it does not have to end in ".f"
"nnnnn"	an integer number which specifies the number of 32-bit words in the common block used by VAST-2 in each routine requiring temporaries. This space is used to hold the vectors of temporary values needed by the calculation. The same value for space should be used in all routines modified by VAST-2 in one program. Some vector operations need intermediate, temporary vectors. VAST-2 defines a labeled common block called /Q1VASTCM/ which it uses for these temporary vectors. nnnnn specifies the length of this common block in 32-bit words. The default length is 10000 words.
"mmmmm"	an integer number of 32-bit words out of the maximum 4096 words of static vector board memory to be used for VAST-2 generated vector temporaries. The -vpfast flag is exclusive to the -space flag. When used, VAST-2 uses the common block VPFAST to allocate vector temporaries instead of Q1VASTCM. The default length for -vpfast is 4096, for -space the default is 10000. (Refer to the iPSC VX User's Guide.)

In place of the type control option "double", the options "single" or "both" could be used as well (refer to the following section, "Type Control Options"). The "double" option is the default.

The keywords can appear in any order. Keywords and switches can be in upper or lower case. Conflicting options are resolved by taking the last referenced option.

When using the switch directive, the format is:

```
CVD$SWITCH,OPTON = ssss,OPTOFF = tttt,LSTON = lll,LSTOFF = kkk
```

Type control cannot be specified on the SWITCH directive. The SWITCH directive is placed in the FORTRAN source file. The "C" in column one tells the compiler to ignore the line. Only VAST-2 acts on the directive.

TYPE CONTROL OPTIONS

Not all of the microcode routines for both double and single precision floating point data types can fit into the instruction memory of the vector processor at the same time. Thus three options are provided:

"double" (the default)	Only double precision operations are vectorized while single precision operations are left in scalar mode.
"single"	The reverse of the above is true.
"both"	Vector operations are generated without regard to data type, but it is the user's responsibility to generate a new library of only the routines actually called that will fit into the instruction memory. (Refer to the iPSC VX User's Guide.)

Example:

Invoke vast2 and generate only single precision vector operations:

```
vast2 -single myprog.f
```

Request both single and double precision vector operations:

```
vast2 -both fullprog.f
```

OPTON/OPTOFF OPTION SWITCHES

Table 3-1 below shows the available switches which control options that effect the transformation of the input program.

Note that some switches duplicate or overlap the functions of directives. For example, the OPTOFF = D switch is equivalent to the NODEPCHK directive with global scope.

As an example, OPTOFF = EL would cause EQUIVALENCE statements not to be examined for data dependency analysis and IF loops not to be converted to DO loops.

Table 3-1

OPTON/OPTOFF Switches
Options

Switch	Description	Default
A	Permit Associative Transformations	ON
B	unused	
C	unused	
D	Don't ignore potential dependencies	ON
E	Examine EQUIVALENCE statements	ON
F	unused	
G	unused	
H	unused	
I	unused	
J	unused	
K	Treat D in Column 1 as a comment (OFF: D = blank)	ON
L	Transform IF loops to DO loops	ON
M	unused	
N	unused	
O	Minimum DO trip count is one	OFF
P	Allow loop collapse transformations	ON
Q	Take error exit if errors found	ON
R	Split out all user subroutines and functions	OFF
S	Permit partial vectorization of loops (scalar part)	ON
T	Permit outer loop vectorization	ON
U	unused	
V	Vectorize (like CVD$G VECTOR)	ON
W	unused	
X	Create vectorized source file	ON
Y	unused	
Z	Protect from count.lt.0 when saving scalar values	ON

LSTON/LSTOFF LISTING OPTION SWITCHES

Table 3-2 below shows the available switches that control the format of the listing file generated by VAST-2.

As an example, if you wanted to get a 132-column printer listing with no warning messages and no event summary, you would specify LSTOFF = TWE.

Table 3-2

LSTON/LSTOFF Switches
Listing Control

Switch	Description	Default
A	unused	
B	List input line numbers in col 73-80 of out. list.	ON
C	List data dependency conflict messages	ON
D	List declarations added by VAST-2	OFF
E	List event summary at end of routine	ON
F	List fatal error messages	ON
G	List translation diagnostics	ON
H	unused	
I	unused	
J	unused	
K	unused	
L	List input lines	ON
M	unused	
N	List translated code	OFF
O	List optimization notes	OFF
P	List loop summary at end of routine	ON
Q	unused	
R	unused	
S	List only summary information	OFF
T	Terminal listing: format output for 80 columns	ON
	format output for 132 columns	OFF
U	unused	
V	unused	
W	List warning messages	ON
X	unused	
Y	List syntax errors	ON
Z	unused	

CHAPTER 4

DIAGNOSTIC MESSAGES

INTRODUCTION

Transforming a program from a scalar form into vector operations is a complex task, and to do the best possible job, VAST-2 requires your assistance. Two paths of communication are available for this purpose:

- VAST-2 informs you of the actions it takes on the program (which loops were vectorized, which loops were not vectorized, the reasons for their rejection, etc.)

- You can pass information and commands to VAST-2 via directives inserted into the program (refer to Chapter 5, "User Directives").

TYPES OF DIAGNOSTIC MESSAGES

VAST-2's diagnostic messages appear in a group at the end of the source listing and include the line number and (if possible) a relevant symbol. These messages are VAST-2's means of notifying the user of how well the source code has been vectorized. There are several different types of diagnostics:

- **Warning Message** — Some potentially troublesome input has been encountered.
- **Syntax Error** — A construct which is not legal FORTRAN has been encountered. No translation will be done on this program unit.
- **Internal Error** — An internal problem with VAST-2 has been detected. No further processing is done for this routine; translation is attempted again for the next routine in the input file. The error should be reported to ISC's Technical Marketing Support Group.
- **Translation Diagnostic** — Describes a problem in translating a loop into array syntax. Prevents a loop from being vectorized.
- **Data Dependency Conflict** — A real or potential feedback from one loop pass to the next prevents the safe use of vector operations. Prevents at least part of a loop from being vectorized.
- **Note** — A possible opportunity for further optimization. (Off by default.)

Each of these types of messages can be independently suppressed via the LSTOFF parameter on the VAST-2 command line or the SWITCH directive. Appendix B contains a full list of VAST-2's diagnostics with further explanation of the messages and tips on avoiding certain problems.

CHAPTER 5

USER DIRECTIVES

INTRODUCTION

You frequently have information about the overall structure of a program and about its data that is unavailable to VAST-2 through inspection of a single program unit. For this reason, a way for you to guide VAST-2 is by "user directives." User directives are treated as comments by FORTRAN compilers, thus preserving code transportability.

DIRECTIVE FORMAT

User directives have the following format:

Column: 1234567

```
    G
    R
CVD$L directive
     6
```

Column 1	The "C" in Column 1 makes the directive a comment for all other FORTRAN compilers. The "VD$" flags this line as a directive to VAST-2.
Column 5	"G" stands for global (meaning the directive applies until the end of the input file). "R" stands for routine (directive applies until the end of this routine). "L" for loop (directive applies to the next loop encountered). A blank in Column 5 is equivalent to "L." Some directives ignore Column 5. (Directives affecting IF loops must have the R or G option; L option applies only to DO loops.) Many directives can be preceded by "NO," thus effecting the reverse operation.
Column 7	The directive itself must start in column 7. The directive may also start in column 6 if column 5 is blank. (Directives are listed in Table 5-1.)

Examples:

CVD$ NODEPCHK	Turn off data dependency analysis for the next loop
CVD$R NOVECTOR	Turn off vectorization for the rest of this routine
CVD$G LIST	Turn on listing for the rest of the run

DIRECTIVE SUMMARY

The full set of directives is summarized in Table 5-1. The "scope" entry has either I for "Immediate", meaning that the directive applies immediately, or "L" meaning that it applies to the next loop, or "R" meaning that it applies to the whole routine, or "LRG" which means that the loop, routine, or global options can be used to control the scope.

A short description of each of these directives follows the table. In addition, the more important directives are discussed in detail at the appropriate point in the chapters on vectorization.

Table 5-1

VAST-2 Directives

Directive	Function	Default	Scope
NOVECTOR/ VECTOR	Disable/Enable vectorization	VECTOR	LRG
NOASSOC/ ASSOC	Don't/Do perform associative transformations	ASSOC	LRG
SELECT	Select which loop in nest of loops to vectorize on	--	L
NOALTCODE/ ALTCODE	Does not/does generate alternate (scalar) code as well as vector version; choose best at run time	ALTCODE	LRG
NODEPCHK/ DEPCHK	Do/Don't ignore potential data dependencies	DEPCHK	LRG
NOEQVCHK EQVCHK/	Does not/does check EQUIVALENCE statements to see if they cause data dependencies	EQVCHK	LRG
PERMUTATION	Pass list of integer arrays that have no repeated values	--	R
RELATION	Specify relationship between two simple variables	--	R
NOLIST/ LIST	Turn off/on listing of input lines	LIST	LRG
EJECT	Insert page eject into listing	--	I
SWITCH	Pass new global switches	--	I

TRANSFORMATION DIRECTIVES

The directives below are used to change the way VAST-2 transforms a loop.

- **NOVECTOR/VECTOR**

  In spite of VAST-2's best efforts to make the right choices, occasionally a loop may be less efficient after transformation. NOVECTOR is provided to disable transformation in such cases.

  VECTOR serves only to toggle back from NOVECTOR; it does not force vectorization. NOVECTOR inhibits all transformations.

- **NOASSOC/ASSOC**

  By default, VAST-2 transforms certain constructs into vector versions in which the order of operations may be different than the original (i.e., they have been associatively transformed). Because of the finite length of computer words, this may result in answers which differ from the scalar original. (An example of an associative transformation is changing (X + Y) + Z to X + (Y + Z) -- always mathematically correct but not always the same on a computer.)

  The NOASSOC directive disables any associative transformations, such as operation reordering to minimize dependent regions.

- **SELECT**

  SELECT advises VAST-2 to choose the next loop as the one to vectorize in the nest of loops.

- **NOALTCOD/ALTCOD**

By default, when the vector length is not known for a vectorized loop, VAST-2 generates both the vector version and the scalar (original) version, and picks the right one to execute at run time.

The vector length is tested against a value that can be supplied on the ALTCODE directive; if the vector length is greater than the value, the vector code is executed; otherwise, the scalar code is executed. The default value is 0.

Example:

```
CVD$ ALTCODE ( 16 )            (must have vector length greater than 16)
       DO 100 I = 1, N
          A(I) = B(I) + C(I)
100   CONTINUE
```

DATA DEPENDENCY DIRECTIVES

The directives in this section are used to help VAST-2 decide where data dependency conflicts actually exist in a loop. Often the user knows that a relationship is not recursive and can supply the appropriate directive from below to inform VAST-2 of this.

- **NODEPCHK/DEPCHK**

  When an array is stored into within a loop, VAST-2 must determine the exact relationship to all other references to the array in the loop, to ensure that these references can be safely vectorized.

  When the relationships cannot be determined, VAST-2 issues a "potential dependency" diagnostic message. The NODEPCHK directive asserts that all such potentially recursive relationships are in fact not recursive. It does not, however, force the vectorization of operations which are unambiguously recursive.

  The DEPCHK directive is used only to toggle back to the default state.

  The directive CDIR $IVDEP is also treated as equivalent to the CVD$L NODEPCHK directive.

- **NOEQVCHK/EQVCHK**

  NOEQVCHK tells VAST-2 to ignore EQUIVALENCE statements when examining the data dependencies in a loop.

- **PERMUTATION**

  The PERMUTATION directive declares an integer array to have no repeated values. This is useful when the integer array is used as a subscript for another array (as a "vector-valued" subscript).

  If it is known that the integer array merely serves to permute the elements of the subscripted array, then no feedback exists with that array reference.

- **RELATION**

  Advises VAST-2 that the specified relationship exists between two integer variables or between an integer variable and an integer constant.

  This information may be useful to VAST-2 in resolving otherwise ambiguous data dependency relationships.

LISTING DIRECTIVES

These directives are used to change the VAST-2 listing.

- **NOLIST/LIST** — Listing of the input source can be selectively suppressed with the NOLIST/LIST directive pair.

- **EJECT** — The EJECT directive causes a page eject in the input source listing.

OTHER DIRECTIVES

Directives that do not fit in any of the above categories are discussed below:

- **SWITCH** — Described in Chapter 3, "Options." Set (or change) global switches.

CHAPTER 6

VECTORIZATION OVERVIEW

INTRODUCTION

This chapter discusses VAST-2's abilities in the vectorization of scalar FORTRAN loops. It consists of the following sections:

- VAST-2 Generated Names
- Loop Types
- Allowed Statements
- VECTOR/NOVECTOR Directives
- Understanding Loop Variables

VAST-2 GENERATED NAMES

In some cases, VAST-2 must create and insert its own "temporary" arrays into the translated code. These temporary names are related to the variable names in the original program which the temporaries correspond to, if any.

VAST-2 examines the symbols defined in the program unit and creates names that do not conflict with any currently in use. The name of a VAST-2 created variable is made by adding a postfix to a root. The root for a variable which derives directly from a user variable is the first four characters of the name of the original variable. Otherwise the root is a letter indicating the type of the variable:

R = real
I = integer
C = complex
D = double
L = logical

The postfix consists of one or more digits followed by one letter:

U = user-derived variables
S = scalars
V = vectors

LOOP TYPES

Both DO and IF loops are considered for transformation.

- **IF loops**

  Innermost IF loops are converted into DO loops when possible. An innermost IF loop is one which contains no other loops, IF or DO. Criteria for conversion are:

  1. The loop must be unambiguous, that is, there must be a single exit and entrance.
  2. There must be a fixed iteration count.

  Directives affecting IF loops must use the Routine or Global scope options. (Refer to Chapter 5, "User Directives.")

- **DO loops**

  In a nest of DO's, VAST-2 chooses the best loop to vectorize.

ALLOWED STATEMENTS

Statements which may appear in a vectorizable loop are listed below.

- **Assignment** — Example: A(I) = B(I) + C(I)
- **Conditional assignment** — Example: IF (D(I).LT.0) A(I) = B(I) + C(I)
- **GO TO label** — (label must be forward)
- **IF ( ) GO TO label** — (label must be forward)
- **IF ( ) THEN**
- **ELSEIF ( ) THEN**
- **ELSE**
- **ENDIF**
- **Arithmetic IF** — (all labels must be forward) Example: IF (IJK) 10, 20, 30
- **Comment**
- **CONTINUE**
- **FORMAT**

The appearance of any other statement type in a loop causes that loop to be rejected for vectorization. A diagnostic message points out the offending statement in the listing. For example, the WRITE statement causes this loop to be left untranslated:

```
      DO 1 I = 1,N        (Not vectorized)
         A(I) = B(I)*C(I)
         WRITE (6) A(I)
1     CONTINUE
```

Statement labels may appear wherever legal in standard FORTRAN.

VECTOR/NOVECTOR DIRECTIVES

Vectorization may be turned on and off with the VECTOR and NOVECTOR directives. By default, vectorization is on. The following loop will not be vectorized:

```
CVD$  NOVECTOR
      DO 300 I = 1,N
         IF ( D(I).LT.0. ) A(I) = SQRT(B(I)+C(I))
 300  CONTINUE
```

NOTE

It is important to turn vectorization off when a loop is heavily conditional. Otherwise the vectorized code will run slower than the original.

UNDERSTANDING LOOP VARIABLES

The key to understanding vectorization lies in understanding the possible vectorizable and non-vectorizable uses of the variables in a loop. Every variable in a vectorizable loop can be categorized as one of three things: scalar, index, or vector.

- **Scalar** — A single value which does not change with each pass through the loop.
- **Index** — An integer quantity which is incremented by a constant amount each pass through the loop.
- **Vector** — A range of memory locations, with a constant increment or stride between consecutive elements.

Here is an example of each of these:

```
Loop:

      DO 10 I = 1, N
         J = J + 1
         A(J) = X
  10  CONTINUE

Classification:

      I,J:    index variables
      A(J):   vector
      X:      scalar
```

The following chapters examine each of these kinds of variables in detail, and explore how their different uses can make a loop vectorizable or not.

CHAPTER 7

INDEXING

INTRODUCTION

This chapter examines the various ways arrays can be indexed in loops, and describes how they are handled by VAST-2. It contains the following sections:

- Constant Increment Integers
- Index Expressions
- Non-Linear Indexing
- Last Value Saving
- Indirect Addressing
- Multiply Defined Indices
- Index Expressions in Several Dimensions
- Explicit Index Use

CONSTANT INCREMENT INTEGERS

A constant increment integer (CII) is an integer variable which is incremented by a constant amount each pass through a loop. A CII may be defined either in terms of a previously defined CII or in terms of its own previous value. The loop below demonstrates some types of CII's that are handled by VAST-2.

Example:

```
     DO 3210 I = 1,50           (loop index is always a CII)
     J = J+1                    (recurrent CII definition)
     K = I*2+3                  (CII defined in terms of another CII)
     R(I) = B(K)*C(M)+A(J)
3210 M = M-4                    (recurrent CII definition)

     CIIs in this loop: I, J, K, M
```

More specifically, a statement setting up a CII must be transformable into one of these two forms:

Form 1: CII1 = CII1 + or - (invariant expression)

Form 2: CII2 = CII1 * (invariant expression) + or - (invariant expression)

where an invariant expression is an expression whose value is constant for all passes of the DO loop.

The loop below contains examples of some statements that cannot be mapped into one of the forms just presented and thus do not define CII's.

Example:

```
     DO 3218 I = 1,N
     J = J*2              (not a CII)
     K = I/4              (not a CII)
     M = I*I              (not a CII)
3218 A(J) = B(K)*C(M)
```

Although the above example contains some indices that do not define CIIs, the loop is still vectorizable. VAST-2 processes these array references as non-linear indexing (refer to "Non-Linear Indexing" section which follows). The techniques used for handling non-linear indexing result in less efficient code than that generated for linear indexing with CIIs.

INDEX EXPRESSIONS

For an array to be directly accessed as a vector, it must have at least one subscript which is transformable into the form:

CII * (invariant expression) + or - (invariant expression)

Example:

```
     DO 3240 I = 1,N                    (Vectorized)
3240 A(5*I) = B(I-6)+C((I-1)*N+M)
```

NON-LINEAR INDEXING

Non-linear indices are defined in two ways:

- as non-linear functions of CIIs (type 1)
- as non-linear functions of themselves (type 2)

Example:

```
     DO 3218 I = 1,N
     K = I/4              (type 1)
     M = I*I              (type 1)
     J = J*2              (type 2)
3218 A(J) = B(K)*C(M)
```

Type 1 is vectorized by indirectly addressing the array.

Example:

```
     DO 3280 I = 1,N      (Vectorized)
3280 A(I) = B(I/2)
```

Type 2 may be recursive, and is left in scalar mode. If possible, the recursion is isolated and the remainder of the loop vectorized.

LAST VALUE SAVING

Where necessary, VAST-2 sets final values of CIIs to ensure that all variables stored into in the original code are given the same values in the vectorized code. VAST-2 examines the flow of the program unit to see if the final value of an index variable is used outside the loop; if not (as is usually the case) then the vectorized code will not store a value into the index variable.

Occasionally VAST-2 may save a last value unnecessarily. If this happens, the OFF = I switch may be used to suppress the generation of last values.

Example:

```
      COMMON /BLOCK/ I,J
      ...
      DO 3230 I = 1,N
      A(I) = B(J)
3230  J = J+1
```

Translation:

```
      COMMON /BLOCK/ I,J
      ...
      CALL DCOPY ( N, B(J),1, A(1),1 )    (vector copy)
      IF ( N.GT.0 ) THEN                  (must have > 0 iterations)
         I = N+1                          (save last value)
         J = N+J                          (save last value)
      ENDIF
```

INDIRECT ADDRESSING

When an array's subscript is itself vector-valued, the array is said to be "indirectly addressed". This is handled with the "gather" and "scatter" vector operations.

Example:

```
     DO 3260 I = 1,N                          (Vectorized)
3260 A(IA(I)) = B(IB(I)*6-IABS(IC(I)))
```

Such array references will be translated automatically only if they obey these rules:

- An array which is indirectly stored into (scattered) may not appear elsewhere in the loop.
- An array which is indirectly read from (gathered) may not be stored into within the loop.

In the example below, the second occurrence of B violates the second rule.

Example:

```
     DO 3265 I = 1,N
     A(I) = B(IB(I))
3265 B(I) = 0.
```

These rules may be overridden with the "NODEPCHK" user directive if you know that the indexing is non-recursive, i.e., that there are no repeated elements in the indexing array(s).

MULTIPLY DEFINED INDICES

VAST-2 handles loops containing indices which are set more than once.

Example:

```
     DO 3215 I = 1,20     (Vectorized)
     J = J+3              (first definition of J)
     A(J) = 0.
     J = J+2              (second definition of J)
3215 B(J) = 1.
```

INDEX EXPRESSIONS IN SEVERAL DIMENSIONS

VAST-2 recognizes the use of the same index expression in more than one dimension of an array (diagonal indexing) as a vectorizable array reference. In the example below, the vector of values being stored into starts at A(1,1) and skips by N + 1 for each element.

Example:

```
      REAL A(N,N)
      .
      .
      DO 3250 I = 1,N        (Vectorized)
 3250 A(I,I) = B(I)
```

EXPLICIT INDEX USE

The use of a CII outside of array subscripts is translated by VAST-2 into a call to a vector ramp function. In the example below, the use of i inside the COS function requires a vector of values from 1 to N to be multiplied by the array B.

Example:

```
      DO 3270 I = 1,N        (Vectorized)
 3270 A(I) = COS(B(I)*I)
```

CHAPTER 8

STORING INTO SCALARS

INTRODUCTION

Scalar variables are unchanging single locations in memory, such as a simple variable X. Array references whose subscripts contain no CIIs (and thus represent a single location through all passes of the loop) are called "array constants." Array constants are treated similarly to simple scalar variables by VAST-2.

Scalar variables which are defined (stored into) in a loop can sometimes inhibit vectorization. Scalar variables which are not redefined in the loop do not inhibit vectorization. This chapter discusses the transformations used to deal with the storing of values into non-index scalars within a loop.

SCALAR PROMOTION

When a scalar is set to a vector expression, that scalar must be "promoted" to a vector. This requires the introduction of temporary vectors which replace the promoted scalars.

- **Last Values of Promoted Scalars**

  VAST-2 saves the last value of a promoted scalar (the value the scalar would have had on exiting the original loop) only when it determines that the value may be needed following execution of the loop.

- **Conditionally Defined Promoted Scalars**

  If the last value of a conditionally defined promoted scalar is required, that operation is vectorized as well.

  Example:

```
          DO 3320 I = 1, N
             IF ( A(I) .LE. 0 )  GO TO 3320
             S = 1./A(I)            (S is set conditionally)
             B(I) = SQRT(S) + S
     3320 CONTINUE
          X = T * S                 (need last value of S)
```

- **Scalar Folding**

  In certain cases, VAST-2 will eliminate a promoted scalar by forward substitution. This is done only when it results in the elimination of a vector temporary and does not add operations.

  Example:

```
          DO 3330 I = 1, N
             T = A(I)             (T will be eliminated)
             B(I) = T + 1./T      (A(I) is folded in)
     3320 CONTINUE
```

CARRY AROUND SCALARS

Scalars which may be used before they are defined in a loop are called "carry-around" scalars. They may or may not be recursive.

- **Wrap-around Scalars**

  When a carry-around scalar is used to save a calculation from a previous loop pass, and does not create recursion, it may be vectorizable. In this case it is called a "wrap-around" scalar. In the example below, T is a wrap-around scalar.

  Example:

```
     DO 3312 I = 1, N  (Vectorized)
        S = A(I)**2
        B(I) =  S + T
        T = S           (T holds the value for the
3312 CONTINUE             next pass)
```

- **Other Carry-Around Scalars**

  Carry-around scalars which are not recognized as wrap-arounds or which cause recursion inhibit vectorization. All references to these variables are collected in a scalar loop and split out from the rest of the calculation if possible.

  Example:

```
     DO 3313 I = 1,N
     A(I) = S+1/S
     B(I) = C(I)-A(I)+S
3313 S = B(I)+D(I)          (S is recursive)
```

REDUCTION FUNCTIONS

A "reduction function" is an operation which condenses array operands into one scalar value which characterizes some aspect of the input arrays. The reduction functions translated by VAST-2 are:

Operation	Operation Name
S = S + A(I)	sum of elements
S = AMAX1(S,A(I))	maximum value
S = AMIN1(S,A(I))	minimum value
S = S + A(I)*B(I)	dot product
index of maximum element	location of maximum value
index of minimum element	location of minimum value
IF (L(I)) N = N + 1	count of true values
L = L .OR. LA(I)	any true values
L = L .AND. LA(I)	all true values
index of max. abs. value	location of maximum absolute value
index of min. abs. value	location of minimum absolute value

These operations are recognized in very general forms.

CHAPTER 9

DATA DEPENDENCY ANALYSIS

INTRODUCTION

VAST-2 ensures that the vectorized code gives the same answers as the original. For certain loops, straightforward vectorization would result (or could result) in incorrect answers. A loop in which results from one loop pass feed back into a future pass of the same loop is said to have a "data dependency conflict" and cannot be completely vectorized. (Such a loop is also said to be "recursive" or to contain "recurrences.") In these cases, VAST-2 detects the problem, reports it to you, and leaves the loop in its original form.

In certain cases, VAST-2 can determine that the problem is limited to a subset of the operations in the loop, and will cut the loop into vectorizable and non-vectorizable sub-loops. The "%DP" field in the loop summary measures how much of the loop is dependent (i.e., left unvectorized). If the "%DP" field is close to 100, then the loop is almost all dependent and not much vectorization was done.

VAST-2 examines EQUIVALENCE statements to see if they may be masking recursion, and suppresses any potentially unsafe transformations.

DATA DEPENDENCY EXAMPLES

Table 9-1 demonstrates the concepts of data dependency analysis. Four similar loops are shown, and for each loop the sequence of instructions that would be executed in scalar (one at a time) mode and vector (whole arrays at a time) mode is shown. Lower case variables (like "a") stand for new values set in the current loop, while upper case variables (such as "A") stand for old values that were set before the loop started.

Table 9-1

Data Dependency Analysis
(A = old value of A; a = new value of A)

Original Loop	Scalar Sequence	Vector Sequence	Vector Sequence Correct?
Example 1: DO 71 I = 2,N 71 A(I+1) = A(I)*B(I)+C(I)	a(3) = A(2)*B(2) + C(2) a(4) = a(3)*B(3) + C(3) a(5) = a(4)*B(4) + C(4) a(6) = a(5)*B(5) + C(5) . .	a(3) = A(2)*B(2) + C(2) a(4) = A(3)*B(3) + C(3) a(5) = A(4)*B(4) + C(4) a(6) = A(5)*B(5) + C(5) . .	No. We are not using updated values of A.
Example 2: DO 72 I = 2,N 72 A(I-1) = A(I)*B(I)+C(I)	a(1) = A(2)*B(2) + C(2) a(2) = A(3)*B(3) + C(3) a(3) = A(4)*B(4) + C(4) a(4) = A(5)*B(5) + C(5) . .	a(1) = A(2)*B(2) + C(2) a(2) = A(3)*B(3) + C(3) a(3) = A(4)*B(4) + C(4) a(4) = A(5)*B(5) + C(5) . .	Yes. Sequence is identical.
Example 3: DO 73 I = 2,N 73 A(I+K) = A(I)*B(I)+C(I)	a(2 + K) = A(2)*B(2) + C(2) a(3 + K) = ... (Can't show more of sequence because we don't know where A is being changed.)	a(2 + K) = A(2)*B(2) + C(2) a(3 + K) = A(3)*B(3) + C(3) a(4 + K) = A(4)*B(4) + C(4) a(5 + K) = A(5)*B(5) + C(5) . .	? Depends on K. Here, if 0<K<N, then vector is not correct.
Example 4: DO 74 I = 2,N,2 74 A(I+1) = A(I)*B(I)+C(I)	a(3) = A(2)*B(2) + C(2) a(5) = A(4)*B(4) + C(4) a(7) = A(6)*B(6) + C(6) a(9) = A(8)*B(8) + C(8) . .	a(3) = A(2)*B(2) + C(2) a(5) = A(4)*B(4) + C(4) a(7) = A(6)*B(6) + C(6) a(9) = A(8)*B(8) + C(8) . .	Yes, stride of 2 makes operation nonrecursive.

It is easy to see that the scalar and vector sequences for Example 1 in Table 9-1 are not the same. The vector version uses only old values of A, while the scalar version uses new ones. VAST-2 detects that this loop is not safe to vectorize, puts out a data dependency conflict message, and leaves the loop in its original form (the loop is "rejected"). Note that the scalar and vector sequences for Example 2 in Table 9-1 are identical. No feedback of results from one loop pass to another is occurring here. VAST-2 recognizes that this loop is safe to vectorize and does so.

The situation is less clear in Example 3 in Table 9-1; here the use of "K" in A's subscript makes the proper scalar sequence impossible to determine at compile time. If K is 1, the loop functions just like the recursive loop in Example 1; if K is -1, the loop is safe to vectorize (just as Example 2 was). When it is not possible for VAST-2 to tell if a loop is recursive or not, the loop is said to have "ambiguous subscripting." Frequently, you know that a loop is not recursive, but there is no way for VAST-2 to tell (as in Example 3). In these cases, you can assume responsibility and turn off data dependency checking with the NODEPCHK directive. Examples of this are found in Chapter 3.

Example 4 in Table 9-1 shows the same loop found in Example 1, but now it has an increment of 2. VAST-2 is able to figure out that the loop is now not recursive because no results feed back into the calculation.

These four similar examples point out the sensitivity of data dependency analysis to offset and stride values of arrays which appear on both the left and right sides of the equal sign within a loop.

Data dependency analysis extends to more than just single line loops, of course; in the loop shown below, the reference to A at the top of the loop conflicts with the store into A at the bottom. VAST-2 prints a message to this effect and does not vectorize the loop.

```
      DO 1 I = 2,N                (Not Vectorized)
     TEMP = A(I-1)+A(I-2)
     B(I) = TEMP+3.0+A(I)
 1   A(I) = SQRT(B(I))-5.0
```

VAST-2 uses information from other array dimensions as part of its analysis where possible. The loop in the following example is vectorized because VAST-2 can see that N can never equal N + 1 and thus there is no recursion (the references to A are two totally different column vectors -- they do not share data).

```
      DO 2 I = 2,N                (Vectorized)
 2    A(I,N) = A(I-1,N+1)*B(I)+C(I)
```

It is possible for array constants (array references with invariant subscripts) to cause dependency conflicts with vector array references. Because of this, the following cannot be safely converted into vector operations (J may be between 2 and N):

```
      SUBROUTINE UNSAFE ( A, B, N, J )
      REAL A(*), B(*)
      DO 3 I = 2,N                (Not Vectorized)
 3    A(I) = A(J)-B(I)
```

REFERENCE REORDERING

Sometimes a loop does not contain true "feedback" of information but still cannot be vectorized because of the order in which elements are referenced.

In cases like the example below, VAST-2 creates a temporary to hold the elements of an array so that they are still available when needed later. This allows the loop to be safely vectorized.

Example:

```
          DO 40 I=1,N              (Vectorized)
          A(I) = B(I)+C(I)+D(I)
          D(I) = E(I)+A(I+1)       (Create temp for A(I+1) at top of loop)
    40    CONTINUE
```

AMBIGUOUS SUBSCRIPT RESOLUTION

When it is not possible with information contained in the loop to determine the relationship between two references to an array, the situation is called "ambiguous subscripting" or "potential feedback".

In these situations, VAST-2 looks for statements outside of the loop that may provide information which clears up the ambiguity. In the loop below, VAST-2 finds the assignment to N2 which makes it clear that N1 is never equal to N2, and thus there is no feedback.

Example:

```
          N2=N1+1
          DO 100 I=1,N             (Vectorized)
    100   A(I+1,N1) = A(I,N2)*B(I)
```

DATA DEPENDENCY DIRECTIVES

- **NODEPCHK -- Turning Off Data Dependency Checking**

As mentioned previously, the "NODEPCHK" directive gives the user the ability to turn off data dependency checking when array subscripting is potentially recursive. This capability should be used only when it is clear that no real recursion exists. In cases of potential feedback, VAST-2 issues a message asking the user to use a directive (NODEPCHK) if the loop is in fact not recursive.

The following are several examples of situations where you may want to disable data dependency checking. In this loop, VAST-2 plays it safe and does not vectorize, but it is clear to the programmer that K will always be less than or equal to zero, and thus the loop is never recursive:

```
      K = -ABS(M)              (Not Vectorized)
      DO 1 I = 1,N
1     A(I+K) = A(I)+B(I)
```

In the example below, a "CVD$ NODEPCHK" directive is used to allow the loop to vectorize.

```
      K = -ABS(M)              (Vectorized)
CVD$  NODEPCHK
      DO 2 I = 1,N
    2 A(I+K) = A(I)+B(I)
```

In this loop, VAST-2 cannot be sure that N1 does not equal N2 and thus rejects the loop:

```
      SUBROUTINE MOVE ( A, B, N, N1, N2 )
      REAL A(N,*), B(*)
      DO 3 I = 1,N             (Not Vectorized)
    3 A(I+1,N1) = A(I,N2)+B(I)
```

If you know that N1 is never equal to N2, then a directive can be inserted as shown here:

```
CVD$  NODEPCHK
      DO 4 I = 1,N             (Vectorized)
4     A(I+1,N1) = A(I,N2)+B(I)
```

Finally, you can see that the array constant A(N*N) in this loop is not in the range of the vector A(I) (I = 1 to N) and thus a directive can be used:

```
CVD$  NODEPCHK                 (Vectorized)
      DO 6 I = 1,N
6     A(I) = B(I)-A(N*N)
```

- **NOEQVCHK -- Turning off EQUIVALENCE checking**

  It is very rare in real-world programs that recursion is hidden through the use of EQUIVALENCE statements. However, VAST-2 must assume the worst and not vectorize in situations where EQUIVALENCE statements could cause feedback. In programs where many of the variables are EQUIVALENCEd together, this can result in most of the loops being left unvectorized.

  The NOEQVCHK directive is provided so that EQUIVALENCE statements can be ignored for data dependency analysis. This means that variables with different names will not overlap. This is almost always the case, anyway. Equivalence checking can also be turned off for the whole input file on the VAST-2 command line via OPTOFF = E

  In the example below, several local arrays have been equivalenced to a large array in common (perhaps to save space). If the arrays could overlap (for instance, if the value of the variable N was 1500 in the DO 100 loop) then the DO 100 loop cannot be vectorized. However, because it is known that the arrays do not overlap, the NOEQVCHK directive is used for the whole routine.

```
       COMMON /BIG/ POOL(100000)
       DIMENSION A(1),B(1),C(1)
       EQUIVALENCE (POOL(1),A(1)),(POOL(1001),B(1)),(POOL(2001),C(1))
 CVD$R NOEQVCHK                        (don't worry about equivalence)
       DO 100 I = 1, N
          A(I) = B(I) + C(I)
  100  CONTINUE
```

- **RELATION - Giving more information**

  The relation directive is used to provide additional information to VAST-2 to help determine if a loop is safe to vectorize. The RELATION directive has the form:

  CVD$ RELATION (simple1 .rel. simple2)

  where "simple1" and "simple2" are simple integer variables (one of them can be an integer constant), and "rel" is one of GT, LT, GE, LE, EQ, NE with the normal FORTRAN meaning.

  When VAST-2 cannot otherwise figure out whether the relationship between two uses of an array is recursive, it searches the RELATION's you supplied for the current routine to see if they help.

  RELATION directives are only informative and do not force any action. RELATION directives apply for the whole routine they appear in. If conflicting relations are given, the result is unpredictable. It is up to you to insure that the relations specified are correct and consistent.

```
 CVD$  RELATION (J.GE.N)
          ...
       DO 100 I = 1, N                 (if j>n, no overlap)
          A(I+J) = A(I) + B(I)
  100  CONTINUE
```

- **PERMUTATION -- Safe Indirect Addressing**

  When an array with a vector-valued subscript appears on both sides of the equal sign in a loop, feedback is possible even if the subscript is identical. Feedback will occur if there are any repeated elements in the subscript.

  Sometimes, indirect addressing is used because the elements of interest in an array are sparsely distributed; in this case an integer array is used to point at the elements that are really wanted. Here, there are no repeated elements in the integer array.

  This information can be passed to VAST-2 through the PERMUTATION directive. PERMUTATION indicates that the list of integer arrays contain no repeated elements (the integer arrays serve merely to permute the elements of the arrays they indirectly address.)

  Syntax:

  ```
  CVD$ PERMUTATION ( ia1, ia2, ... , ian )
  ```

  PERMUTATION declares the integer arrays ("ia1", etc.) to have no repeated values for the entire routine.

  Example:

  ```
  CVD$  PERMUTATION ( IPNT )
        ...
        DO 100 I = 1, N
           A(IPNT(I)) = A(IPNT(I)) + B(I)
   100  CONTINUE
  ```

SPLIT LOOP TO AVOID RECURSION

When recursion or potential recursion exists in a loop and no other method can be applied to convert it, then VAST-2 will cut the recursive area out of the loop so that the rest of the loop can be vectorized. VAST-2 attempts to minimize the amount of code executed in scalar mode by moving operations out of the scalar section.

Example:

```
      DO 3440 I = 1,N
      X = A(I)+B(I)                (Vector)
      D(I+1) = (X*C(I)+2.)/D(I)    (Vector: X*C(I) + 2.)
 3440 B(I) = B(I)+D(I+1)*X         (Vector)
```

CHAPTER 10

DECISION PROCESSES

INTRODUCTION

This chapter describes the varieties of conditional statements in loops and how VAST-2 deals with them. It contains the following sections:

- Allowable Conditional Constructs
- Kinds of Conditional Statements
- Two Types of Conditions
- Conditional Indexing

ALLOWABLE CONDITIONAL CONSTRUCTS

VAST-2 can vectorize any combination of conditional assignments, conditional and unconditional forward branching (including arithmetic IF's), and block IF's. Because of compilation speed restrictions, there is a limit of six simultaneously active conditions.

KINDS OF CONDITIONAL STATEMENTS

VAST-2 can handle conditional assignment statements:

Form: IF (logical expression) variable = expression

Example:

```
      DO 10 I = 1,N
10    IF (B(I).LT.C(I)-X) A(I) = SQRT(D(I))
```

VAST-2 can also handle forward transfers:

Form: IF (logical expression) GO TO label

and

GO TO label
(where "label" is lower down in the same DO loop)

Example:

```
      DO 20 I = 1,N
      IF (A(I).GT.0) GO TO 15
      B(I) = C(I)
15    E(I) = 0.
20    D(I) = B(I)-6.1
```

and VAST-2 recognizes block IF statements:

Form:

```
IF (logical expression) THEN
.
.
ELSE                    (optional)
.
.
ELSE IF (logical expression) THEN
.                                        (optional)
.
ENDIF
```

Example:

```
   DO 30 I = 1,N
      IF (A(I).EQ.B(I)) THEN
         A(I) = 2.0*B(I)
         E(I) = E(I)**2
      ELSE
         A(I) = 0.
         E(I) = D(I)
      ENDIF
30 CONTINUE
```

VAST-2 can also handle arithmetic IF's. There is no nesting limit to block IF's. Forward transfers do not have to be properly nested; VAST-2 converts all forward transfers into structured IF statements.

These conditional statements are not handled:

- Computed GOTO
- Assigned GO TO
- Backward transfers
- Transfers out of the loop

TWO TYPES OF CONDITIONS

There are two basic types of conditions, as illustrated in the loops below. The first loop shows a "loop independent" conditional assignment:

```
      DO 1 I = 1,100              (Vectorized)
      B(I) = B(I)+1.
      IF (N.EQ.1) A(I) = B(I)+2.
    1 CONTINUE
```

The IF clause (N.EQ.1) does not depend on the loop index at all; it remains the same for all iterations of the loop. Thus, we know whether or not N is equal to 1 before the loop is executed. VAST-2 handles this situation by testing only once and conditionally executing vector operations. In this way, the loop is vectorized and many unnecessary tests that were done in the original code are avoided in the translated code.

The other kind of condition is shown in this loop:

```
      DO 2 I = 1,100              (Vectorized)
      IF (C(I).LT.0) D(I) = E(I)+F(I)
    2 CONTINUE
```

It is called "loop dependent" because the IF clause (C(I).LT.0) might change with each loop pass. Sometimes we want to do the calculation (add E(I) to F(I)) and sometimes we do not. This is vectorized in a "brute force" way by performing the calculation across all elements and then storing only those results that were really wanted into the result array.

The "%CD" field in the loop summary of the VAST-2 listing summarizes what percentage of the operations in the translated loop are executed under the control of loop dependent conditional statements. A value close to 100 in this field means that the loop is almost completely conditional, and may run slower than the scalar original if the condition is extremely sparse.

You should examine loops with large values in the %CD field to make sure that the calculation in those loops is usually executed. If the calculation is skipped over by conditional statements most of the time, the loop will probably run slower when vectorized, and a NOVECTOR directive should be used to turn off vectorization.

CONDITIONAL INDEXING

Indices that are set up in a non self-referent fashion and are set and used all within the same conditional branch are treated normally (the condition is ignored). If the last value of a conditionally defined index is required, operations referencing that index cannot be vectorized.

Example:

```
      DO 3670 I = 1,N             (Vectorized)
      IF (A(I).GT.0.) THEN
         J = I*3+2                (Last Value not needed)
         B(J) = 0.
      ENDIF
3670  CONTINUE
```

Indices set in a conditional branch and used outside are treated as carry-around scalars.

Example:

```
     DO 3677 I = 1,N             (Not vectorized)
     IF (C(I).GT.X) J = J+1      (J is conditionally set...)
3677 B(J) = B(J)+A(I)            (...and unconditionally used)
```

CHAPTER 11

FUNCTIONS & SUBROUTINES

INTRODUCTION

This chapter discusses the handling of loops containing function references or subroutine calls. Unless otherwise stated, calls to functions or subroutines inhibit vectorization.

The following are included in this chapter:

- Statement Functions
- Intrinsic Functions

STATEMENT FUNCTIONS

VAST-2 examines statement functions and where safe vectorizes loops that reference them by expanding the statement function in-line.

Example:

```
          STMFNC(X) = X+SQRT(X-1.)
                      ...
          DO 3510 I = 1,N                    (Vectorizable)
     3510 A(I) = B(I)+STMFNC(C(I))*.5        (statement function is expanded)
```

INTRINSIC FUNCTIONS

In general, intrinsic functions can be applied to array arguments as well as scalars. See Appendix D for the list of intrinsic functions available on the vector processor. For the most part, references to standard intrinsics do not inhibit vectorization.

Example:

```
          DO 3520 I = 1,N                                  (Vectorizable)
     3520 A(I) = AMAX1(SIN(SQRT(B(I)+C(I))),D(I))
```

CHAPTER 12

OUTER LOOPS

INTRODUCTION

All possible loops are examined in a nested loop situation. Loops do not need to be tightly nested, and vectorized outer loops can have multiple inner loops. When outer loop vectorization is being done, data dependency messages are tagged with the loop label and loop index which caused them; this makes it easier to determine where the dependency is coming from.

CHOOSING THE BEST LOOP

When VAST-2 tries to pick the best loop in a nest, it uses these criteria:

- Vector Length
- Number of single-precision vectors accessed with a non-unit stride
- Amount of dependent (scalar) code
- Amount of conditional code

For example, in the loop nest below the inner loop is recursive so the outer loop is chosen by VAST-2 and the inner loop is left scalar.

Example:

```
      DO 30 J = 1,M              (Outer loop is vectorizable)
         A(J) = B(J)*2.0
         X = A(J)-C(J)
         DO 20 I = 1,N           (Inner loop is not vectorizable)
            D(I+1,J) = -D(I,J)/X*E(I,J)          (Recursive)
20       CONTINUE
         B(J) = E(K,J)-2.0
30    CONTINUE
```

SELECT DIRECTIVE

The SELECT directive allows you to override VAST-2's choice of dimension on which to vectorize. The SELECT directive should be placed directly before the DO statement of the loop to be vectorized. Please note that the VECTOR and NOVECTOR directives are NOT used to control loop selection within a nest of loops. If a NOVECTOR directive is placed on a loop, VAST-2 does not examine that loop or any outer loops around it for vectorization.

For example, in the loop below VAST-2 would choose the inner loop as it has unit stride on the arrays and VAST-2 had no information about the relative values of the iteration counts N and M. However, if N were 11 and M were 11000, it may be better to vectorize on the outer loop by using the SELECT directive.

Example:

```
CVD$ SELECT                 (select the J loop)
     DO 200 J = 1, M
     DO 200 I = 1, N
     A(I,J) = B(I,J) * .5
200  CONTINUE
```

OUTER LOOP VECTORIZATION INHIBITORS

There are several situations which can prevent an outer loop from being vectorized. For instance, when the iteration count of an inner loop is not constant for all passes of the outer loop, then the outer loop is not vectorized (the inner loops are still vectorized if possible). Below, the iteration count for both the 100 loop and the 150 loop change with each pass of the outer loop. Thus the 200 loop cannot be vectorized.

```
      DO 200 J = 1,N                    (Not Vectorized)
         DO 100 I = 1,J                 (Vectorized)
100         A(I,J) = A(I,J)*B(J,I)
         DO 150 I = 1,IN(J)             (Vectorized)
150         C(I,J) = 0.
200   CONTINUE
```

Sometimes recursion along the inner loop also prevents vectorization of the outer loop. For example, the DO 400 loop below has feedback on the array A along the inner loop, and also cannot safely be vectorized along the DO 500 loop.

```
      DO 500 J = 1,M                    (Not Vectorized)
         DO 400 I = 1,N                 (Not Vectorized)
400         A(I+1) = A(I) * B(I,J)
500   CONTINUE
```

When an inner loop is executed conditionally, the outer loop is not vectorized. In the example below, the DO 700 loop is not vectorized since the DO 600 loop inside is executed conditionally.

```
      DO 700 I = 1,N                (Not Vectorized)
         IF (A(I).GT.0) THEN
            DO 600 J = 1,N          (Vectorized)
               B(I,J) = A(I)*J
600         CONTINUE
         ENDIF
700   CONTINUE
```

Finally, loops which do no indexing are not vectorized. This sometimes occurs when an outer loop is being used for timing purposes. In the example below, no use is made of the index I or any other index defined for the outer loop DO 900.

```
      DO 900 I = 1,N                (Not Vectorized)
         DO 800 J = 1,M             (Vectorized)
            A(J) = B(J) + C(J)
800      CONTINUE
900   CONTINUE
```

LOOP NEST COLLAPSE

Under certain restrictive conditions, loop nests can be collapsed into a single loop whose iteration count is the product of the iteration counts of the uncollapsed loops. If the loop bounds are identical to the array bounds and the loops are tightly nested, the transformation is done automatically.

Collapse criteria:

- There must be no recursion in the loops.
- There must be no explicit use of CIIs.

All vector array references in the loop must conform: the first N subscripts of each reference must be identical, where N is the nesting depth. Each of the N subscripts must be indexed by one and only one of the loop indices, by a stride of one. The declarations of these arrays must also conform: the first N-1 dimensions must be identical.

Example:

```
          REAL A(12,10)
          .
          .
          DO 200 J = 1, 10        (Done as one vector operation
             DO 100 I = 1, 12       of length 120)
                A(I,J) = 0.
100          CONTINUE
200       CONTINUE
```

CHAPTER 13

MISCELLANEOUS TRANSFORMATIONS

INTRODUCTION

Optimizing compilers for scalar computers change the order of the instructions they produce in order to overlap operations as much as possible. A common technique used to aid such compilers is to "unroll" short loops to give the compiler more flexibility in reordering instructions within a single loop pass. This is generally undersirable on vector machines.

DESCRIPTION

The following is an example of loop re-rolling.

For example, a simple dot product

```
      DO 2 I = 1, N
      S = S + A(I)
  2   CONTINUE
```

might be unrolled like this:

```
      DO 2 I = 1, N-3, 4
      S = S + A(I) + A(I+1) + A(I+2) + A(I+3)
  2   CONTINUE
```

The second loop performs the same function as the first, but it does four additions per pass instead of one.

The unrolling technique is generally undesirable on vector machines, since it decreases vector length and increases indexing overhead. In some cases, VAST-2 can detect unrolled loops and "re-roll" them into their original form.

Re-rollable loops must have an explicit constant non-one increment specified on the DO statement. They must contain no data dependency problems. For an unrolled summation or dot product they must have each operand added to the reduction scalar in order. For an unrolled assignment, there must be the same number of assignments as the increment on the DO loop and each assignment must do the next computation in order.

Below are two examples of loops that VAST-2 can re-roll.

Example 1:

```
      DO 100 I = 1, 997, 3        (rerolled into one vector add)
         A(I) = B(I) + C(I)
         A(I+1) = B(I+1) + C(I+1)
         A(I+2) = B(I+2) + C(I+2)
100   CONTINUE
```

Example 2:

```
      DO 200 I = 1, N, 5          (rerolled into one dot product)
         S = S + A(I)*B(I) + A(I+1)*B(I+1) + A(I+2)*B(I+2)
     1         + A(I+3)*B(I+3) + A(I+4)*B(I+4)
200   CONTINUE
```

APPENDIX A

VECTORIZATION RULES

INTRODUCTION

This appendix provides a summary of some of the rules mentioned in this manual. It covers the following:

- General
- Data Dependencies
- Functions
- Scalars
- Reduction Functions
- Indirect Addressing
- Outer Loops

- **General**
  - Only assignment, logical IF, GO TO, arithmetic IF, block IF, comment, and CONTINUE statements can be used in a vectorizable loop (no computed GO TO's, I/O, etc.).

- **Data Dependencies**
  - Recursion (feedback of results from one loop pass as inputs to another) prevents vectorization. However, if the recursion is isolated to only part of a loop, the rest of the loop will be vectorized and the recursion left in a scalar loop.
  - Potential recursion, where VAST-2 cannot tell whether or not the loop is recursive (also called ambiguous subscripting), prevents vectorization. When the loop is safe to vectorize, the "NODEPCHK" directive should be used to allow vectorization.

- **Functions**
  - Only intrinsic functions with vector versions are allowed in vectorizable loops.

- **Scalars**
  - In general, scalars must be defined (appear on the left side of the equal sign) before they are used (appear on the right side of the equal sign) if they are defined in the loop. Otherwise they are carry-around scalars, which must be processed in a scalar loop, or wrap-around scalars, which are vectorizable.

- **Reduction Functions**
  - Only summation of elements, minimum or maximum element, index of minimum or maximum element, index of absolute maximum or minimum element, dot product, "any", "all", and "count" are recognized.
  - A reduction function scalar may not appear anywhere else in the loop.

- **Indirect Addressing**
  - A gathered array must not appear on the left-hand side in the loop.
  - A scattered array must not appear anywhere else in the loop.

- **Outer Loops**
  - An outer loop cannot be vectorized if it sets the iteration count of any inner loop.
  - An outer loop cannot be vectorized if it conditionally executes any inner loop.
  - The outer loop must use some index at least once.

APPENDIX B

DIAGNOSTIC MESSAGES

INTRODUCTION

This appendix shows all VAST-2 messages, grouped into the following categories:

- Syntax Errors
- Data Dependency Conflicts
- Translation Diagnostics
- Internal Errors
- Directive Errors
- Notes

SYNTAX ERRORS

These are listed here but not illustrated; they should be self-explanatory. VAST-2 does not systematically check the syntax of the entire input source. It is assumed that all VAST-2 input has previously been successfully compiled and executed without VAST-2.

ARRAY NOT DECLARED
BAD RECORD ON INPUT FILE
BRANCH INTO DO-LOOP
CANNOT EQUIVALENCE TWO VARIABLES IN COMMON
CONSTANT HAS TOO MANY DIGITS OR CHARACTERS
END OF LOOP NOT FOUND -- MISSING LABEL
EQUIVALENCE USES MORE DIMENSIONS THAN DECLARED
ERROR IN INPUT/OUTPUT STATEMENT
ERROR IN EQUIVALENCE
FORMAL PARAMETERS MAY NOT BE EQUIVALENCED
IDENTIFIER EXCEEDS MAXIMUM NUMBER OF CHARACTERS
ILLEGAL ARRAY DIMENSIONS
ILLEGAL BRANCH INTO BLOCK IF
ILLEGAL CHARACTER FOUND IN INPUT LINE
ILLEGAL COMPUTED GOTO STATEMENT
ILLEGAL HOLLERITH CONSTANT
ILLEGAL NUMBER OF ARGUMENTS TO INTRINSIC FUNCTION
ILLEGAL STATEMENT FUNCTION DEFINITION
ILLEGAL STATEMENT FUNCTION REFERENCE
ILLEGAL SUBROUTINE CALL
ILLEGAL SYNTAX IN SPECIFICATION STATEMENT
IMPROPER ELSEIF STATEMENT
IMPROPER NESTING OF DO LOOPS
MISMATCH OF OPERANDS AND OPERATORS
MISSING ENDIF(S)
MISSING LABEL
MORE THAN 7 SUBSCRIPTS
NON-BLANK CHARACTERS IN COLUMNS 1-5 OF CONTINUATION CARD
NUMBER OF SUBSCRIPTS DECLARED AND USED DO NOT MATCH
SYNTAX ERROR IN ASSIGNED GO TO
TERMINATOR FOR CHARACTER STRING CONSTANT NOT FOUND
UNBALANCED ENDIF
UNBALANCED PARENTHESES
UNKNOWN OPERATOR OR ILLEGAL USE OF DECIMAL POINT
WRONG TYPE FOR INTRINSIC FUNCTION ARGUMENT

DATA DEPENDENCY CONFLICTS

Data dependency messages are always followed by the name of the variable which is causing the problem. If outer loop vectorization is being attempted, the label and index of the loop causing the problem is given as well.

"FEEDBACK OF ARRAY ELEMENTS"

Feedback of results makes the loop recursive and thus unsafe to vectorize.

Example:

```
      DO 4 I=1,N
  4   A(I+1) = A(I) + B(I)
```

"FEEDBACK OF SCALAR VALUE FROM ONE LOOP PASS TO ANOTHER"

In the following example, the variable "SCA" is used in the first line of the DO loop to set A(I), and then is set to a function of A(I) in the last line. This creates feedback of elements of A from one loop pass to the next, which prevents vectorization. SCA is called a "carry-around" scalar, as it carries a value around to the next pass of the loop.

Example:

```
      DO 112 I = 1,N
        A(I) = A(I) + SCA
  112 SCA = A(I)/C(J)
```

"POTENTIAL FEEDBACK OF ARRAY ELEMENTS -- USE DIRECTIVE IF OK"

It is not clear whether there is feedback between the two uses of A in this loop or not (it depends on the value of J). Loops of this kind that the user is sure are safe can be vectorized by putting the directive "CVD$ NODEPCHK" in front of the loop.

Example:

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

Here is another case where this message would result:

Example:

```
      DO 1 I=1,N
  1   B(I)=B(IB(I))+A(I)
```

"B(IB(I))" is a gathered array, and as the values in IB(I) are unknown, it may conflict with the use of B on the left side of the equal sign. If the pattern of B(IB(I)) is known to not overlap B(I), then the NODEPCHK directive should be used.

"POTENTIAL MULTIPLE STORE CONFLICT -- USE DIRECTIVE IF OK"

The loop below has a potential overlap between the two stores into A; usually in these situations there is no real overlap between the two sections of A and the NODEPCHK directive should be used to get the loop to vectorize.

Example:

```
        DO 100 I=1,N
        A(I+J)=B(I)
  100   A(I+K)=C(I)
```

"TOO MANY DATA DEPENDENCY PROBLEMS"

The loop has exceeded the maximum number of 12 data dependency conflicts allowed. Vectorization of the loop is abandoned at this point to avoid printing out further messages.

TRANSLATION DIAGNOSTICS

Translation diagnostics point out items that prevent a loop from being translated.

Statement Types

This set of diagnostic messages cover types of statements that prevent vectorization.

"ASSIGN IS NOT VECTORIZABLE"

Example:

```
      DO 210 I = 1, N
        IF ( A(I) .GT. 0 ) ASSIGN 225 TO ITOGO
210   CONTINUE
```

"ASSIGNED GOTO IS NOT VECTORIZABLE"

Example:

```
      DO 220 I = 1, N
        GOTO     ITOGO
220   CONTINUE
```

"COMPUTED GOTO IS NOT VECTORIZABLE"

Example:

```
      DO 230 I = 1, N
        GO TO ( 231, 232, 233 ) IGO(I)
231     B(I) = 0
232     B(I) = B(I) + A(I)
233     B(I) = B(I) * C(I)
230   CONTINUE
```

"I/O STATEMENTS ARE NOT VECTORIZABLE"

Example:

```
      DO 240 I = 1, N
        WRITE ( IOUNIT ) A(I)*B(I)+C(I)
240   CONTINUE
```

"RETURN IS NOT VECTORIZABLE"

Example:

```
      DO 250 I = 1, N
        IF ( X(I) .LT. 0 ) RETURN
        Y(I) = SQRT(X(I))
250   CONTINUE
```

- **"STOP IS NOT VECTORIZABLE"**

Example:

```
     DO 260 I = 1, N
        IF ( X(I) .LT. 0 ) STOP 99
        Y(I) = ALOG ( X(I) )
260  CONTINUE
```

- **"NOT A VECTORIZABLE STATEMENT"**

This message identifies a non-vectorizable statement not covered by one of the preceding messages.

Example:

```
     DO 109 I=1,9
    IF (A(I).LT.B(I)) PAUSE 'FOR REFRESHMENT'
```

Branches

The messages below relate to handling of conditional operations.

"BACKWARD TRANSFERS ARE NOT VECTORIZABLE"

The branch to label 104 is backward, not forward, and prevents vectorization. In some cases however, backward transfers may be converted into separate DO loops.

Example:

```
      DO 107 I=1,N
104   A(J)=SQRT(B(J)+D(I))
      J=J+1
      IF (A(J).GT.LIMIT) GO TO 104
107   CONTINUE
```

"BRANCHES OUT OF THE LOOP PREVENT VECTORIZATION"

The label "777" is not in the loop.

Example:

```
      DO 108 I=1,N
      IF (A(I).LT.0) GO TO 777
108   B(I)=6.0
```

"BRANCHES ARE TOO COMPLEX TO VECTORIZE"

Because of limits on speed and table space, conditional constructs with more than six simultaneous conditions cannot be converted to vector syntax.

External References

The messages below deal with external references in loops.

"SUBROUTINE CALL PREVENTS VECTORIZATION"

Subroutine calls within loops prevent vectorization.

Example:

```
      DO 110 I=1,N
      A(I)=SQRT(B(I)/C(I))
      CALL REVAMP(A(I),D(I))
  110 D(I)=EXP(A(I)+X)
```

"REFERENCE TO FUNCTION THAT HAS NO VECTOR VERSION"

References to non-intrinsic functions in a loop prevent vectorization of the loop.

Example:

```
      DO 115 I=1,N
  115 A(I) = MYFUNC(B(I))
```

DO Statement

"DO STATEMENT PARAMETERS MUST BE INTEGER FOR ARRAY TRANSLT."

For example, type real variables or constants are not allowed as the loop index or in the start, end or increment fields.

Example:

```
      DO 100 W = 1.0001, 1000000.
         A(I) = W
100   CONTINUE
```

"USER FUNCTION REFERENCES NOT ALLOWED IN ITERATION COUNT"

In this example NLEN is a user function. User functions cannot appear in the vector length expression for a DO loop (they cannot be in the start, end or increment fields of the DO statement).

Example:

```
      DO 210 I = 1, NLEN(J,K)
210   A(I)=0.
```

"VECTOR LENGTH TOO SHORT"

When loops have an explicit vector length which is less than a set cut-off amount (5) they are not vectorized as they will probably run faster in scalar mode.

Example:

```
      DO 116 I=1,2
116   A(I) = B(I)
```

Scalars

"UNABLE TO DETERMINE LAST VALUE OF SCALAR TEMPORARY"

USE NOLSTVAL DIRECTIVE IF POSSIBLE

Can't determine which element of a vector temporary has the last value to be assigned to a scalar temporary. Asks user to use NOLSTVAL directive if possible.

Example:

```
      DO 85 I = 1, N
         IF ( A(I) .GT. 0 ) THEN
               S = B(I)**2
            A(I) = S + 1/S
         ENDIF
85    CONTINUE
      WRITE ( IOUNIT ) S
```

Outer loops

"OUTER LOOP MAY NOT SET INNER LOOP ITERATION COUNT"

The iteration count of an inner loop must not change between passes of an outer loop. Here the inner loop is vectorized but not the outer.

Example:

```
      DO 350 J = 1, N
         DO 350 I = 1, J, 2
            S = S + D(I,J)
  350 CONTINUE
```

"OUTER LOOP MAY NOT CONDITIONALLY EXECUTE INNER LOOP"

Again, the inner loop will be vectorized but not the outer.

Example:

```
      DO 361 J = 1, N
         IF ( A(J) .LE. 0 ) THEN
            DO 360 I = 1, M, 2
               D(I,J) = SQRT(A(J)*D(I,J))
  360       CONTINUE
         ENDIF
  361 CONTINUE
```

"NO INDEXING DONE ALONG THIS LOOP"

The outer loop does no useful work and is not vectorized.

Example:

```
      DO 371 MQ = 1, 1000
         DO 370 I = 1, N, 2
            S = S + A(I)
  370    CONTINUE
  371 CONTINUE
```

Miscellaneous

"NULL LOOP BODY"

Nothing is in the loop. (The loop is not eliminated.)

Example:

```
      DO 111 I=1,N
111   CONTINUE
```

"CHARACTER DATA TYPE NOT VECTORIZABLE"

Use of character substrings prevents a loop from being considered for vectorization.

Example:

```
      DO 200 I = 1, N
         P(I)(1:2) = Q(I)(2:3)
200   CONTINUE
```

"STATEMENT FUNCTION CONTAINS VECTORIZATION INHIBITORS"

Example:

```
      F(J) = J*J
      .
      .
      .
      DO 100 J=1,N
         B(F(J))=0.0
100   CONTINUE
```

INTERNAL ERRORS

"STATEMENT FUNCTION EXPANDS LINE PAST SIZE OF BUFFER"

A table overflow has resulted from internal expansion of a statement containing statement function references.

"INTERNAL ERROR DETECTED -- PLEASE REPORT"

Call Intel Scientific Computers (ISC).

DIRECTIVE ERRORS

"UNKNOWN DIRECTIVE -- IT IS IGNORED"

Example:

```
CVD$   SCALARIZE
```

NOTES

"IF LOOP CONVERTED TO DO-LOOP"	An IF loop has been converted to a DO loop. (The resulting DO loop may or may not be vectorizable.)
"VARIABLE USED BUT NEVER DEFINED"	A local variable is used in executable statements but is never defined.
"VARIABLE DEFINED BUT NEVER USED"	A local variable is defined in executable statements but is never used.
"VARIABLE USED BUT NOT DEFINED"	A local variable is used when it is undefined, although it is defined elsewhere in the module.
"VARIABLE DEFINED BUT NOT USED"	This definition of a local variable is not used, although the variable is used elsewhere in the module.
"DEAD CODE"	Flags a section of code which, because of the program's flow of control, can never be executed.

APPENDIX C

GLOSSARY

Ambiguous Subscripts	Array subscripts which may cause feedback from one loop pass to another (making it unclear whether it is safe to vectorize or not.)
Array	A contiguous set of memory locations.
Array Constant	An array reference whose subscript value stays the same through all passes of a loop (e.g., A(6)).
Array Constant Temporary	An array constant into which a vector expression is stored. (A form of Scalar Temporary.)
Array Operation	An operation across many memory locations.
Array Syntax	The array syntax extensions in the 8X FORTRAN standard.
Associative Transformation	A transformation which changes the order of evaluation of an expression. Example: a + (b + c) to (a + b) + c. May result in slightly different answers due to finite length of computer word.
Carry-Around Scalar	A scalar that is stored into for the first time in a loop after its first use in a loop.
CII	See "Constant Increment Integer."
Conditional Assignment Statement	Statement of the form "IF (expr1) variable = expr2" (example: IF (X.LT.0) A = B + C).
Constant	An explicit number (like 100). See also "Invariant".

Constant Increment Integer

An integer variable which is incremented by a constant amount each pass through a loop. Used interchangeably with "Index". (The abbreviated form, CII, is used in this manual.)

Data Dependency Analysis

Process of examining the body of a loop to see if there are any unsafe or potentially unsafe relationships between uses of the same array (or EQUIVALENCED arrays) on both sides of the equal sign. (An "unsafe" relationship means the loop may give wrong answers if vectorized.)

Data Dependency Conflict

The use of a variable on the left side of the equal sign "conflicts" with a use of the same variable on the right hand side of the equal sign; a recursive or potentially recursive relationship exists. VAST-2 issues Data Dependency Conflict messages and does not vectorize loops which contain recursion.

Definition (of a Variable)

Appearance of a variable on the left hand side of the equal sign (place where variable is stored into).

Diagnostic

A message printed by VAST-2 describing some problem with or comment about the program being translated. These are not "errors".

Directive

See "User Directive".

Dot Product

The summation of the element-by-element multiplication of two vectors. Also called the "inner product". As it takes vector inputs and produces a single scalar result, it is a Reduction Function.

Explicit Index Use

Use of an index as part of the calculation, rather than in setting up another index or as part of an array subscript.

Forward Transfer

A branch to a label which has a line number greater than the line number of the branch.

Gather

Reorder an indirectly addressed array into a linearly addressed vector (T(I) = A(IA(I))). See also "Scatter".

Index Expression	An expression involving one index variable which can be multiplied by and/or added to invariants (a linear combination of one index variable).
Index Variable	A variable which is used to select array elements in a loop. Must have a constant stride. Same meaning as "Constant Increment Integer".
Indirectly Addressed Array	An array whose subscript is a vector expression rather than an index expression. (Example: A(IA(I))).
Invariant	A quantity which is the same for all loop passes. An "invariant expression" involves only invariants.
Iteration Count	Number of times a loop is executed. Also called the "vector length."
Last Value Saving	In the process of vectorization, the definitions within the loop of certain scalars (indices and scalar temporaries) disappear. If the actual final loop value of such a scalar is required elsewhere in the program unit, VAST-2 generates an operation to explicitly store the correct value into the scalar variable.
Loop	Refers to DO loops and to IF loops that can be transformed into DO loops.
Loop-Dependent Condition	The condition may have a different value for each iteration of the loop.
Loop-Independent Condition	The condition may have only one value for all iterations of the loop.
Loop Index	I in "DO 75 I = 1,N". The variable in the DO statement. The loop index is always a constant increment integer.
Outer Loop	Loop containing at least one other loop.

Potential Dependency	The relationship between two uses of an array cannot be precisely determined and therefore there is a possible data dependency conflict between the uses.
Promoted Scalar	A scalar that has been promoted to an array. A temporary array is created to hold all the values that the scalar had held in the original loop.
Nested Loops	One loop inside another loop.
NODEPCHK	User directive which turns off data dependency checking; useful in cases of ambiguous subscripting.
Non-Linear Indexing	Indexing that cannot be translated into (Linear Index) * (Invariant Expression) + (Invariant Expression).
Recursion	The use in one loop pass of operands which were computed in a previous loop pass ("feedback").
Recurrence	An instance of recursion (e.g., A(I) = A(I-1)*B(I)).
Reduction Function	An operation which maps vector inputs into a scalar result (reduces a whole array of values into a single result). The most common examples are the summation of a vector and the dot product of two vectors.
Reduction Function Scalar	The single result of a reduction function. For example, in the vector summation TOTAL = TOTAL + A(I), TOTAL is the reduction function scalar.
Restructuring	The process of altering a program to expose more vector operations. The user "restructures" the program so that more of it will be vectorized.
Scalar	A simple variable. One memory location.
Scalar mode	Operation on single scalar operands or pairs of operands.

Scalar Temporary — A scalar which is used to hold a different value for each loop pass (a scalar which is set to a vector expression). Also called a "Promoted Scalar". Example: X = A(I) + B(I).

Scatter — Scramble a linearly addressed array into an indirectly stored array. (A(IA(I)) = T(I)) See also "Gather".

Split-out — Something that is "split-out" of a loop is isolated in a scalar loop while the rest of the loop is vectorized.

Stride — A constant skip or increment through memory.

Translation Diagnostic — A message which describes what prevented a loop from vectorizing. The loop will be left in its original form.

User Directive — An input line which begins with "CVD$". The way for the user to pass information and commands to VAST-2.

VAST-2 — An acronym for Vector and Array Syntax Translator. A precompiler which translates standard FORTRAN to vector or array syntax.

Vector — A set of memory locations which are related by a constant stride (constant number of words between each element of the vector).

Vector Length — Number of elements in a vector.

Vector Operation — Operation involving at least one vector; more than one element is processed at the same time.

Vectorization — The process of converting a program to use vector operations.

APPENDIX D

VECTOR ROUTINES

INTRODUCTION

In the following argument lists, *n* is always an integer representing the vector length, and is always the first argument to the function or subroutine call. Also, *incw, incx, incy* and *incz* are always integers. The result is generally the last argument.

MATHEMATICAL PRIMITIVES

Double precision: *a*, *x*, *y*, and *z* are double precision

```
z(*) = x(*) + y(*)          call dvadd ( n, x,incx, y,incy, z,incz )
z(*) = a + x(*)             call dsadd ( n, a, x,incx, z,incz )
z(*) = x(*) - y(*)          call dvsub ( n, x,incx, y,incy, z,incz )
z(*) = a - x(*)             call dssub ( n, a, x,incx, z,incz )
z(*) = x(*) * y(*)          call dvmul ( n, x,incx, y,incy, z,incz )
z(*) = a * x(*)             call dsmul ( n, a, x,incx, z,incz )
x(*) = alpha * x(*)         call dscal ( n, alpha, x,incx )
z(*) = x(*) / y(*)          call dvdiv ( n, x,incx, y,incy, z,incz )
z(*) = a / x(*)             call dsdiv ( n, a, x,incx, z,incz )
z(*) = x(*) ** y(*)         call dvpow ( n, x,incx, y,incy, z,incz )
y(*) = x(*)                 call dcopy ( n, x,incx, y,incy )
z(*) = alpha                call dfill ( n, alpha, z,incz )
y(*) = -x(*)                call dvneg ( n, x,incx, y,incy )
x(*) = -x(*)                call dneg ( n, x,incx )
```

Single precision: *a*, *x*, *y*, and *z* are double precision

```
z(*) = x(*) + y(*)          call svadd ( n, x,incx, y,incy, z,incz )
z(*) = a + x(*)             call ssadd ( n, a, x,incx, z,incz )
z(*) = x(*) - y(*)          call svsub ( n, x,incx, y,incy, z,incz )
z(*) = a - x(*)             call sssub ( n, a, x,incx, z,incz )
z(*) = x(*) * y(*)          call svmul ( n, x,incx, y,incy, z,incz )
z(*) = a * x(*)             call ssmul ( n, a, x,incx, z,incz )
x(*) = alpha * x(*)         call sscal ( n, alpha, x,incx )
z(*) = x(*) / y(*)          call svdiv ( n, x,incx, y,incy, z,incz )
z(*) = a / x(*)             call ssdiv ( n, a, x,incx, z,incz )
z(*) = x(*) ** y(*)         call svpow ( n, x,incx, y,incy, z,incz )
y(*) = x(*)                 call scopy ( n, x,incx, y,incy )
z(*) = alpha                call sfill ( n, alpha, z,incz )
y(*) = -x(*)                call svneg ( n, x,incx, y,incy )
x(*) = -x(*)                call sneg ( n, x,incx )
```

Integer: *a*, *x*, *y*, and *z* are integers

```
z(*) = x(*) + y(*)          call ivadd ( n, x,incx, y,incy, z,incz )
z(*) = a + x(*)             call isadd ( n, a, x,incx, z,incz )
z(*) = x(*) - y(*)          call ivsub ( n, x,incx, y,incy, z,incz )
z(*) = a - x(*)             call issub ( n, a, x,incx, z,incz )
z(*) = x(*) * y(*)          call ivmul ( n, x,incx, y,incy, z,incz )
z(*) = a * x(*)             call ismul ( n, a, x,incx, z,incz )
z(*) = x(*) / y(*)          call ivdiv ( n, x,incx, y,incy, z,incz )
z(*) = a / x(*)             call isdiv ( n, a, x,incx, z,incz )
y(*) = x(*)                 call icopy ( n, x,incx, y,incy )
z(*) = alpha                call ifill ( n, alpha, z,incz )
y(*) = -x(*)                call ivneg ( n, x,incx, y,incy )
x(*) = -x(*)                call ineg ( n, x,incx )
```

RELATIONAL PRIMITIVES

In this section, z is always of data type logical.

Double precision: *a, x,* and *y* are double precision and z is logical

```
z(*) = x(*).eq.y(*)       call deq ( n, x,incx, y,incy, z,incz )
z(*) = a.eq.x(*)          call dseq ( n, a, x,incx, z,incz )
z(*) = x(*).ne.y(*)       call dne ( n, x,incx, y,incy, z,incz )
z(*) = a.ne.x(*)          call dsne ( n, a, x,incx, z,incz )
z(*) = x(*).gt.y(*)       call dgt ( n, x,incx, y,incy, z,incz )
z(*) = a.gt.x(*)          call dsgt ( n, a, x,incx, z,incz )
z(*) = a.lt.x(*)          call dslt ( n, a, x,incx, z,incz )
z(*) = x(*).ge.y(*)       call dge ( n, x,incx, y,incy, z,incz )
z(*) = a.ge.x(*)          call dsge ( n, a, x,incx, z,incz )
z(*) = a.le.x(*)          call dsle ( n, a, x,incx, z,incz )
```

Single precision: *a, x,* and *y* are single precision and z is logical

```
z(*) = x(*).eq.y(*)       call seq ( n, x,incx, y,incy, z,incz )
z(*) = a.eq.x(*)          call sseq ( n, a, x,incx, z,incz )
z(*) = x(*).ne.y(*)       call sne ( n, x,incx, y,incy, z,incz )
z(*) = a.ne.x(*)          call ssne ( n, a, x,incx, z,incz )
z(*) = x(*).gt.y(*)       call sgt ( n, x,incx, y,incy, z,incz )
z(*) = a.gt.x(*)          call ssgt ( n, a, x,incx, z,incz )
z(*) = a.lt.x(*)          call sslt ( n, a, x,incx, z,incz )
z(*) = x(*).ge.y(*)       call sge ( n, x,incx, y,incy, z,incz )
z(*) = a.ge.x(*)          call ssge ( n, a, x,incx, z,incz )
z(*) = a.le.x(*)          call ssle ( n, a, x,incx, z,incz )
```

Integer: *a, x,* and *y* are integers and z is logical

```
z(*) = x(*).eq.y(*)       call ieq ( n, x,incx, y,incy, z,incz )
z(*) = a.eq.x(*)          call iseq ( n, a, x,incx, z,incz )
z(*) = x(*).ne.y(*)       call ine ( n, x,incx, y,incy, z,incz )
z(*) = a.ne.x(*)          call isne ( n, a, x,incx, z,incz )
z(*) = x(*).gt.y(*)       call igt ( n, x,incx, y,incy, z,incz )
z(*) = a.gt.x(*)          call isgt ( n, a, x,incx, z,incz )
z(*) = a.lt.x(*)          call islt ( n, a, x,incx, z,incz )
z(*) = x(*).ge.y(*)       call ige ( n, x,incx, y,incy, z,incz )
z(*) = a.ge.x(*)          call isge ( n, a, x,incx, z,incz )
z(*) = a.le.x(*)          call isle ( n, a, x,incx, z,incz )
```

TRIADS

Double precision: *a, b, w, x, y,* and *z* are double precision

```
z(*) = (w(*)-x(*))*y(*)      call dvvmvt ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = (a-x(*))*y(*)         call dsvmvt ( n, a, x,incx, y,incy, z,incz )
z(*) = a*(x(*)-y(*))         call dsvvmt ( n, a, x,incx, y,incy, z,incz )
z(*) = (w(*)+x(*))*y(*)      call dvvpvt ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = (a+x(*))*y(*)         call dsvpvt ( n, a, x,incx, y,incy, z,incz )
z(*) = a*(x(*)+y(*))         call dsvvpt ( n, a, x,incx, y,incy, z,incz )
z(*) = w(*)*x(*)-y(*)        call dvvtvm ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = a*x(*)-y(*)           call dsvtvm ( n, a, x,incx, y,incy, z,incz )
z(*) = w(*)*x(*)+y(*)        call dvvtvp ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = a*x(*)+y(*)           call dsvtvp ( n, a, x,incx, y,incy, z,incz )
z(*) = a+x(*)*y(*)           call dsvvtp ( n, a, x,incx, y,incy, z,incz )
z(*) = a*x(*)+b              call dsvpst( n, a, b, x,incx, z,incz )
y(*) = alpha*x(*)+y(*)       call daxpy ( n, alpha, x,incx, y,incy )
z(*) = w(*)-x(*)*y(*)        call dvvvtm ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = a-x(*)*y(*)           call dsvvtm ( n, a, x,incx, y,incy, z,incz )
```

Single precision: *a, b, w, x, y,* and *z* are single precision

```
z(*) = (w(*)-x(*))*y(*)      call svvmvt ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = (a-x(*))*y(*)         call ssvmvt ( n, a, x,incx, y,incy, z,incz )
z(*) = a*(x(*)-y(*))         call ssvvmt ( n, a, x,incx, y,incy, z,incz )
z(*) = (w(*)+x(*))*y(*)      call svvpvt ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = (a+x(*))*y(*)         call ssvpvt ( n, a, x,incx, y,incy, z,incz )
z(*) = a*(x(*)+y(*))         call ssvvpt ( n, a, x,incx, y,incy, z,incz )
z(*) = w(*)*x(*)-y(*)        call svvtvm ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = a*x(*)-y(*)           call ssvtvm ( n, a, x,incx, y,incy, z,incz )
z(*) = w(*)*x(*)+y(*)        call svvtvp ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = a*x(*)+y(*)           call ssvtvp ( n, a, x,incx, y,incy, z,incz )
z(*) = a+x(*)*y(*)           call ssvvtp ( n, a, x,incx, y,incy, z,incz )
z(*) = a*x(*)+b              call ssvpst ( n, a, b, x,incx, z,incz )
y(*) = alpha*x(*)+y(*)       call saxpy ( n, alpha, x,incx, y,incy )
z(*) = w(*)-x(*)*y(*)        call svvvtm ( n, w,incw, x,incx, y,incy, z,incz )
z(*) = a-x(*)*y(*)           call ssvvtm ( n, a, x,incx, y,incy, z,incz )
```

LOGICAL PRIMITIVES

Logical: *a, x, y,* and *z* are logical

```
z(*) = x(*) .and. y(*)        call land ( n, x,incx, y,incy, z,incz )
z(*) = a .and. x(*)           call lsand ( n, a, x,incy, z,incz )
z(*) = x(*) .or. y(*)         call lor ( n, x,incx, y,incy, z,incz )
z(*) = a .or. y(*)            call lsor ( n, a, x,incy, z,incz )
y(*) = .not. x(*)             call lnot ( n, x,incx, y,incy )
z(*) = x(*)                   call lcopy (n, x,incx, z,incz)
                              call lfill (n, a, x, incx)
```

Double precision: *w, x,* and *z* are double precision; y is logical

```
if (y(*)) z(*) = x(*)        call dcndst ( n, x,incx, y,incy, z,incz )
if (y(*)) z(*) = w(*) then
     else z(*) = x(*)        call dmask ( n, w,incw, x,incx, y,incy,z, incz )
```

Single precision: *w, x,* and *z* are single precision; y is logical

```
if (y(*)) z(*) = x(*)        call scndst ( n, x,incx, y,incy, z,incz )

if (y(*)) z(*) = w(*)        call smask ( n, w,incx, x,incy, y,incy, z,incz )
     else z(*) = x(*)
```

REDUCTION FUNCTION PRIMITIVES

Double precision: *s, x,* and *y* are double precision; *j* is integer

```
s = s + x(*)                    s = s + dsum ( n, x,incx )
s = s + dabs(x(*))              s = s + dasum ( n, x,incx )
s = s + x(*)*y(*)               s = s + ddot ( n, x,incx, y,incy )
index of max value of x(*)      j = idmax ( n, x,incx )
index of min value of x(*)      j = idmin ( n, x,incx )
index max value of abs(x(*))    j = idamax ( n, x,incx )
index min value of abs(x(*))    j = idamin ( n, x,incx )
```

Single precision: *s, x,* and *y* are single precision; *j* is integer

```
s = s + x(*)                    s = s + ssum ( n, x,incx )
s = s + abs(x(*))               s = s + sasum ( n, x,incx )
s = s + x(*)*y(*)               s = s + sdot ( n, x,incx, y,incy )
index of max value of x(*)      j = ismax ( n, x,incx )
index of min value of x(*)      j = ismin ( n, x,incx )
index max value of abs(x(*))    j = isamax ( n, x,incx )
index min value of abs(x(*))    j = isamin ( n, x,incx )
```

Logical: *l* and *x* are logical; *m* and *j* are integer

```
number of trues in x(*)         m = icount ( n, x,incx )
index of last true in x(*)      j = ilast ( n, x,incx )
true if any trues in x(*)       l = lany ( n, x,incx )
```

INTRINSIC FUNCTION PRIMITIVES

Double precision: *x*, *y*, and *z* are double precision

```
z(*) = dabs(x(*))                  call dvabs ( n, x,incx, z,incz )
z(*) = atan2(x(*),y(*),z(*))       call dvatn2 ( n, x,incx, y,incy, z,incz )
z(*) = atan(x(*))                  call dvatn (n, x,incx, y, incy)
z(*) = cos(x(*))                   call dvcos ( n, x,incx, z,incz )
z(*) = exp(x(*))                   call dvexp ( n, x,incx, z,incz )
z(*) = log10(x(*))                 call dvlg10 ( n, x,incx, z,incz )
z(*) = log(x(*))                   call dvlog ( n, x,incx, z,incz )
z(*) = sqrt(x(*))                  call dvsqrt ( n, x,incx, z,incz )
z(*) = sin(x(*))                   call dvsin ( n, x,incx, z,incz )
z(*) = max(x(*),y(*))              call dvmax ( n, x,incx, y,incy, z,incz )
z(*) = min(x(*),y(*))              call dvmin ( n, x,incx, y,incy, z,incz )
z(*)=max(abs(x(*)),abs(y(*)))      call dvamax ( n, x,incx, y,incy, z,incz )
z(*)=min(abs(x(*)),abs(y(*)))      call dvamin ( n, x,incx, y,incy, z,incz )
```

Single precision: *x*, *y*, and *z* are single precision

```
z(*) = abs(x(*))                   call svabs ( n, x,incx, z,incz )
z(*) = atan2(x(*),y(*),z(*))       call svatn2 ( n, x,incx, y,incy, z,incz )
z(*) = atan(x(*))                  call svatn (n, x,incx, y, incy)
z(*) = cos(x(*))                   call svcos ( n, x,incx, z,incz )
z(*) = exp(x(*))                   call svexp ( n, x,incx, z,incz )
z(*) = log10(x(*))                 call svlg10 ( n, x,incx, z,incz )
z(*) = log(x(*))                   call svlog ( n, x,incx, z,incz )
z(*) = sin(x(*))                   call svsin ( n, x,incx, z,incz )
z(*) = sqrt(x(*))                  call svsqrt ( n, x,incx, z,incz )
z(*) = max(x(*),y(*))              call svmax ( n, x,incx, y,incy, z,incz )
z(*) = min(x(*),y(*))              call svmin ( n, x,incx, y,incy, z,incz )
z(*)=max(abs(x(*)),abs(y(*)))      call svamax ( n, x,incx, y,incy, z,incz )
z(*)=min(abs(x(*)),abs(y(*)))      call svamin ( n, x,incx, y,incy, z,incz )
```

Integer: *x*, and *z* are integers.

```
z(*) = iabs(x(*))                  call ivabs ( n, x,incx, z,incz )
```

CONVERSION PRIMITIVES

k is integer, *x* is double precision:

```
k(*) = idint(x(*))         call dvfix ( n, x,incx, k,inck )
```

k is integer, *y* is double precision:

```
y(*) = dfloat(k(*))        call dvfloa ( n, k,inck, y,incy )
```

k is integer, *x* is single precision:

```
k(*) = ifix(x(*))          call svfix ( n, x,incx, k,inck )
```

k is integer, *y* is single precision:

```
y(*) = float(k(*))         call svfloa ( n, k,inck, y,incy )
```

x is single precision, *z* is double precision:

```
z(*) = dble(x(*))          call vdble ( n, x,incx, z,incz )
```

x is double precision, *z* is single precision:

```
z(*) = sngl(x(*))          call vsngl ( n, x,incx, z,incz )
```

GATHER/SCATTER PRIMITIVES

In this section, *k* is always of data type integer

Double precision: *x* and *z* are double precision

```
z(*) = x(k(*))             call dgathr ( n, x,1, k,inck, z,incz )
z(k(*)) = x(*)             call dscatr ( n, x,incx, k,inck, z,1 )
```

Single precision: *x* and *z* are single precision

```
z(*) = x(k(*))             call sgathr ( n, x,1, k,inck, z,incz )
z(k(*)) = x(*)             call sscatr ( n, x,incx, k,inck, z,1 )
```

STEP FUNCTION PRIMITIVES

Double precision: *alpha*, *beta*, and *z* are double precision

```
z(*) = alpha + (i-1)*beta       call dramp ( n, alpha, beta, z,incz )
```

Single precision: *alpha*, *beta*, and *z* are single precision

```
z(*) = alpha + (i-1)*beta       call sramp ( n, alpha, beta, z,incz )
```

Integer: *alpha*, *beta*, and *z* are integers

```
z(i) = alpha + (i-1)*beta       call iramp ( n, alpha, beta, z,incz )
```

MULTI-LINE TRANSFORMATION PRIMITIVES

Double precision: *x* and *y* are double precision

```
Swap              call dswap ( n, x,incx, y,incy )
```

Single precision: *x* and *y* are single precision

```
Swap              call sswap ( n, x,incx, y,incy )
```

MISCELLANEOUS VECLIB ROUTINES NOT USED BY VAST-2

Complex to complex FFT routines:

```
call cfft( n, x, incx, y, incy)
call cifft( n, x, incx, y, incy)
```

Clip and inverse clip routines:

Double precision:

```
dclip(n, x, incx, alpha, beta, z, incz )
diclip(n, x, incx, alpha, beta, z, incz )
```

Single precision:

```
sclip(n, x, incx, alpha, beta, z, incz )
siclip(n, x, incx, alpha, beta, z, incz )
```

Euclidean vector norm:

Double precision:

```
dnrm2( n, x, incx )
```

Single precision:

```
snrm2 (n, x, incx )
```

Polynomial evaluation:

Double precision:

```
dvpoly( n, x, incx, m, c, incc, z, incz )
```

Single precision:

```
svpoly( n, x, incx, m, c, incc, z, incz )
```