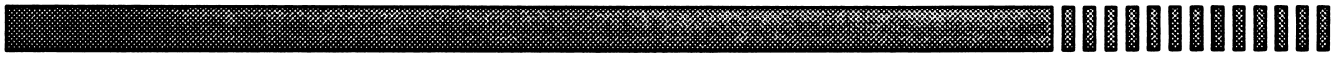
June, 1989
Order Number: 311570-002



iPSC®/2-VX

USER'S GUIDE

intel® Corporation

REV.	REVISION HISTORY	DATE
-001	Original issue	03/88
-002	Revision	06/89

RESTRICTED RIGHTS

PREFACE

PURPOSE

The purpose of this manual is to provide information for developing programs for the iPSC®/2-VX (vector processor system). This manual gives an overview of the hardware and software, outlines the program development steps, and describes the utility routines for the vector processor. It provides a sample makefile and several program examples.

SCOPE

This manual covers information specific to the iPSC/2-VX only. This manual does not detail all of the standard iPSC/2 commands and routines. Also, it does not explain parallelization and message-passing routines. This manual provides only a minimal explanation of the Green Hills Fortran and C compilers and other development tools. These topics are covered in the standard set of iPSC/2 manuals described in the section entitled "Applicable Documents." You should be familiar with the standard iPSC/2 commands and routines, and standard vector mathematics.

ORGANIZATION

Chapter 1 is an overview of the iPSC/2-VX system and a functional description of the relevant hardware and software.

Chapter 2 describes the program development process for the iPSC/2-VX, and describes how to vectorize your Fortran or C programs.

Chapter 3 describes how to link and compile your vectorized programs, with an illustrative sample makefile.

Chapter 4 contains a set of advanced programming topics, describing how to use static memory, the *vxld* command, and some special routines for advanced programming.

Chapter 5 provides several program examples illustrating applications for the iPSC/2-VX.

Chapter 6 contains a reference page (in alphabetical order) for each of the Fortran VX utility routines.

Chapter contains a reference page (in alphabetical order) for each of the C VX utility routines.

Appendix A consists of two tables that are summaries of the VecLib routines; the first shows the Fortran synopses, and the second shows the C synopses for the routines. Complete descriptions of these routines are in Chapters 4 and 5 of the *iPSC/2® Programmer's Reference Manual.*

APPLICABLE DOCUMENTS

For more information, refer to the other members of the iPSC/2 manual set.

iPSC®/2 User's Guide
This manual provides general information on how to use the iPSC/2, and is intended to provide you with enough detail to begin using the iPSC/2 system.

iPSC®/2 VAST2 User's Guide
Describes how to use the optional VAST-2 precompiler to vectorize DO and IF loops in Fortran programs.

iPSC®/2 System Administrator's Guide
This manual provides a detailed description of the system administration tasks related to operating and maintaining an iPSC/2 system.

iPSC®/2 C Language Reference Manual
This manual describes the C compiler for the iPSC/2 system.

iPSC®/2 Fortran Language Reference Manual
This manual describes the Fortran compiler for the iPSC/2 system.

iPSC®/2 DECON User's Guide
This manual describes how to use DECON, the iPSC/2 concurrent debugger.

iPSC®/2 Simulator
This manual describes how to use the iPSC/2 Simulator for software development.

UNIX System V Manual Set
These manuals provide a complete description of the UNIX System V operating system.

TABLE OF CONTENTS

CHAPTER 1 SYSTEM OVERVIEW

CHAPTER 2 PROGRAM DEVELOPMENT

CHAPTER 3 COMPILING AND LINKING YOUR PROGRAM

CHAPTER 4 ADVANCED PROGRAMMING TOPICS

CHAPTER 5 iPSC®/2-VX PROGRAM EXAMPLES

CHAPTER 6 iPSC®/2-VX FORTRAN UTILITY ROUTINES

CHAPTER 7 iPSC®/2-VX C UTILITY ROUTINES

APPENDIX A FORTRAN VECLIB ROUTINE SUMMARY

APPENDIX B C VECLIB ROUTINE SUMMARY

APPENDIX C ERROR MESSAGES

APPENDIX D DATA ALIGNMENT

APPENDIX E RESERVED WORDS

LIST OF TABLES

LIST OF FIGURES

SYSTEM OVERVIEW 1

INTRODUCTION

This chapter gives a functional overview of the iPSC/2-VX system, and describes the iPSC/2-VX hardware and software. This chapter assumes you have a basic understanding of the iPSC/2 system configuration, message-passing routines, the iPSC/2 Fortran and/or C compilers, and the optional VAST-2 vectorizing precompiler.

GENERAL SYSTEM DESCRIPTION

The iPSC/2-VX is an enhanced version of the standard iPSC/2 system in which vector processor boards are attached to each node processor in the system. The vector processor is capable of doing high-speed, floating-point arithmetic. A wide variety of scalar and vector operations are available for the vector processor under microcode control.

The easiest way to use the vector processor board when you are writing Fortran application programs is to use VAST-2, a pre-compiler that can read standard Fortran and substitute the appropriate vector instructions. The output from VAST-2 is modified source which can be fed directly into the Fortran compiler. (Refer to the *iPSC®/2 VAST2 User's Guide* for more information.) You must vectorize C programs by hand.

Code for the vector processor can be written either in Fortran or in C. The VecLib (vector library) is a library containing routines that interface to the microcoded vector operations. The following example, shows the original Fortran loop and its replacement, and the corresponding C loop and its replacement:

Fortran DO loop:

```
      q = 0.0
      do 3 k = 1, 1000
          q = q + z( k ) * x( k )
3     continue
```

The loop replaced with the DDOT function (automatic with the VAST-2 vectorizer):

```
q = 0.0
q = q + ddot( 1000, z, 1, x, 1 )
```

C for loop:

```
q = 0.0;
for(k=1,;k<1000;k++)
    q = q + z( k ) * x( k );
```

The loop replaced with the DDOT function:

```
q = ddot( 1000, z, 1, x, 1 );
```

The optional VAST-2 vectorizer automatically replaces vectorizable Fortran do loops with the equivalent VecLib statement. You must vectorize C programs by hand.

Chapter 2 describes the program development flow for the iPSC/2-VX, and contains information on vectorizing your code.

HARDWARE DESCRIPTION

The vector processor board is a high-speed arithmetic processor capable of performing integer and floating-point computations. It is used as an arithmetic accelerator to the node CPU.

The vector processor board is capable of reaching peak performance levels from static (fast) memory of 6.67 megaflops (64-bit) and 20 megaflops (32-bit). Dynamic memory produces peak performance levels of 3.3 megaflops (64-bit) and 10 megaflops (32-bit).

The vector processor board performs floating-point arithmetic IEEE-754 single and double-precision formats as well as integer arithmetic in two's complement format and logical operations on 32-bit operands. It does not do IEEE arithmetic in the following cases:

- It does not handle denormals. Denormals are treated as zeros.
- It does not perform IEEE rounding on divide and square root.

Figure 1-1 shows the basic components of the iPSC/2-VX. Functional groups are described following the figure.

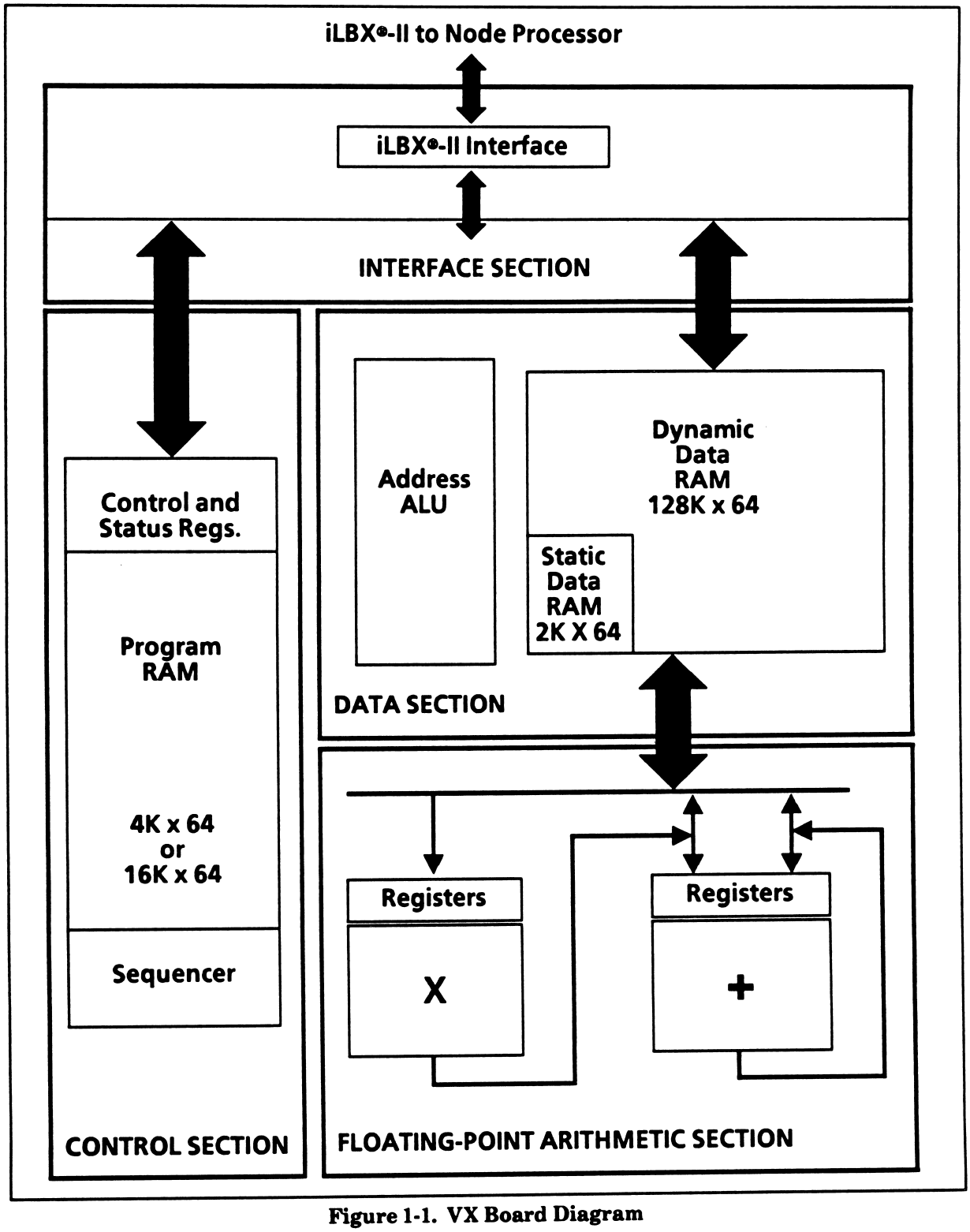


Figure 1-1. VX Board Diagram

Data Section

Static Data RAM — 16K bytes of fast memory, which contains tables of constants and supports high-performance access to variables declared within the VPFAST common block. Approx. 5K bytes are user-accessible (see Chapter 4).

Dynamic Data Ram — 1M-byte of dynamic RAM containing some tables and user data.

Address ALU — Generates the addresses for elements in the vector. It executes in parallel with the floating-point arithmetic chips.

Control Section

Control and Status registers — Registers that are memory-mapped into the iLBX®-II address space. These special registers provide miscellaneous control functions and information about the state of the machine (such as underflow and overflow conditions).

Program RAM — 4K or 16K-word writable control storage for the iPSC/2-VX. Contains microcode loaded with the application. Each 64-bit word is one microcode instruction including fields for controlling the sequencer, floating point adder and multiplier, address ALU, and miscellaneous control functions.

Sequencer — 16-bit microsequencer that executes microcode from program memory.

Interface Section

iLBX-II interface — The bridge between the node board and the vector processor board. iLBX-II is one of Intel's MULTIBUS®-II protocols.

Floating Point Arithmetic Section

Adder — Executes addition and subtraction operations on integer, single precision, and double precision data, and logical operations on Integers only. Executes independently under microcode control.

Multiplier — Executes multiplication operations on Integer, single precision and double precision data. Executes independently under microcode control.

Node/Vector Memory

Each VX node consists of two boards: the standard node board (with the 386™ processor and the 387™ co-processor) or the SX node board (with the Weitek 1167 co-processor) and the vector processor board. The node board contains from 1 to 8M bytes of memory, of which the node operating system uses at least 200K bytes. The remainder is available for the application program.

The vector processor board has 1Mbyte of memory. By default, code resides on the node board and data resides on the vector processor board. If you want data to be stored on the node board's memory, you can use the *vxld* command when linking.

Figure 1-2 shows a map of physical memory as seen by the main processor.

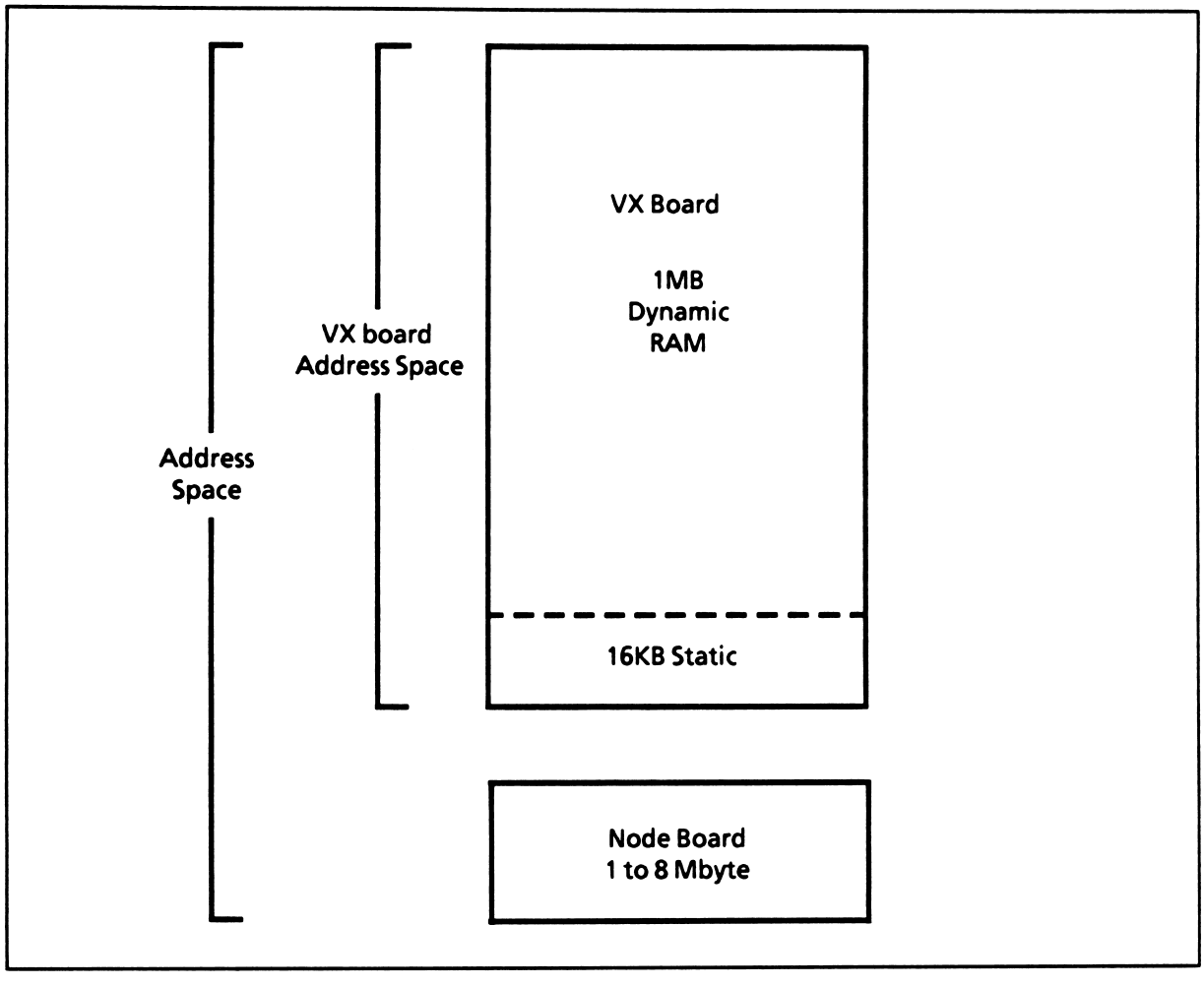


Figure 1-2. Node/Vector Memory Map

SOFTWARE DESCRIPTION

In addition to the standard iPSC/2 software, the iPSC/2-VX system has some additional directories and files as described below.

Utilities

The "/usr/bin" directory contains utilities for vector-specific operations. These include the VAST-2 precompiler (vast2) and several utilities which must be run to prepare code for execution.

vast2 — An optional Fortran vectorizer and pre-compiler for the iPSC/2-VX. It converts DO and IF loops to VecLib routines. Both input source and output are ANSI standard Fortran 77. The output is compiled by the Green Hills Fortran compiler for execution on the iPSC/2-VX.

vxld — Used by **f77** and **cc** to link VX executable files.

These are further explained in Chapter 3 of this manual.

Libraries and Object Files

The */usr/lib* directory contains the following libraries and files with vector routines, common blocks and other vector-specific files:

libvec.a — Executes on standard node board. Contains the VecLib routines which run on the standard node board or System Resource Manager.

libvxvec.a — Executes on a vector processor board. This library contains the VecLib routines. They consist of the Basic Linear Algebra Subroutines (BLAS) and a variety of other vector operation routines. Refer to Appendix A of this manual for a summary list of VecLib routines. For a detailed description of all the routines, refer to the *iPSC®/2 Programmer's Reference Manual.*

libvxdbvec.a — Executes on vector processor board. A version of VecLib that provides some parameter-checking capability. Refer to Chapter 3 of this manual for more information.

libsxvec.a — Executes on node boards with the SX option. Contains the VecLib routines which run on node boards with the SX option.

libvxsxvec.a	Executes on vector processor boards and nodes that have the SX option board (refer to the *iPSC®/2 User's Guide* for more information on the SX option). This library is identical to *libvxvec.a* except that it contains routines that execute on the SX option board rather than the standard node board.
libvxsxdbvec.a	Executes on vector processor board and nodes that have SX option. A version of VecLib that provides some parameter-checking capability. Refer to Chapter 3 of this manual for more information.
libvx.a	Contains the low-level interface routines for the vector processor board, and the vector memory allocation routines for C programmers. The other routines in this library are used by VecLib. Refer to Chapter 5 of this manual for more information.
libvxdb.a	Contains the parameter-checking version of the low-level interface routines found in *libvx.a*.
libvxsim.a	This library contains all of the routines found in *libvx.a* and is used when running the simulator. This library can be used in conjunction with *libvec.a* to run programs on the System Resource Manager with the hypercube simulator. For more information on the simulator, refer to the *iPSC®/2 Simulator Manual*. This library can also be used with *libvec.a* on standard node boards or *libsxvec.a* on node boards with the SX option.
h_single.o *l_single.o*	These files are linked with your application when you use the **-single** switch on the **f77** or **cc** command. These files should be linked with your application if it calls single precision or either of the two complex VecLib routines (forward and inverse FFT) or if you specify the **-single** switch in VAST-2.
h_double.o *l_double.o*	These files are linked with your application when you use the **-double** switch on the **f77** or **cc** command. These files should be linked with your application if it calls double precision VecLib routines or you specify the **-double** switch in VAST-2.
h_both.o *l_both.o*	These files are linked with your application when you use one of the **-both** , **-all** , or **-complex** switches in the **f77** or **cc** command. These files should be linked with your application if it calls both single and double precision, or any of the complex or double precision complex VecLib routines, or if you specify one of the **-both** , **-all** , or **-complex** switches in VAST-2.

All integer and logical VecLib routines, the conversion routines, the conditional store, mask, gather, scatter, and all swap and copy routines are found in both the double and single precision versions of the above files.

Include Files

The */usr/include* directory contain files with useful declarations and macro instructions for program development on the iPSC/2-VX. These files are helpful when writing your own vector instructions. You can insert these files into your source program with an **include** statement.

veclib.h	This file defines data types (single precision, double precision, complex, logical, or integer) for the VecLib routines.
cveclib.h	This file is the C version of *veclib.h*.
vpnode.h	This file defines data types for the utility routines that are functions found in *libvx.a*. It also contains a description of the VPFAST common block.
vpcmd.h	This file contains symbolic definitions of the microcode routines for the vector processor.

Examples

The *usr/ipsc/examples/vx/vx_vec_node* directory contains four example programs and a makefile for use as models when developing vector programs. The four example programs are:

timedaxpy.f	Fortran timing driver for DAXPY (constant times a vector plus a vector)
timesaxpy.c	C timing driver for SAXPY
dmain.f, dgefa.f, dgesl.f, seconds.f	double precision version of the standard linpack benchmark
smain.f, sgefa.f, sgesl.f, seconds.f	single precision version of the standard linpack benchmark

If the optional VAST-2 vectorizer is a part of your system, the *usr/ipsc/examples/vx/vast* directory contains the *loop7.v* example which shows how VAST-2 vectorizes Fortran loops.

Diagnostics

Diagnostics for the iPSC/2-VX are found in the */usr/ipsc/diag* directory. The Cube Diagnostic Program (CDP) provides an Optional Board Test that tests the functionality of the vector processor boards. For more information on CDP, refer to the *iPSC®/2 System Administrator's Guide*.

PROGRAM DEVELOPMENT 2

INTRODUCTION

This chapter provides, first, an overview of the steps you need to take to develop your Fortran or C program for execution on the iPSC/2-VX system. Next is information on how to vectorize your code. The next chapter describes how to compile and link your vectorized code.

OVERVIEW

The first step to take in preparing your software for the vector processor is to modify your source program so that it uses the Vector Library (VecLib). VecLib is a library of approximately 250 routines that perform vectorizable operations, taking advantage of the iPSC/2-VX capabilities. There are VecLib routines for both C and Fortran. These routines include not only the floating-point operations for 32-bit and 64-bit operands, but also vectorizable integer, logical, and complex operations. To use VecLib routines in either of your applications, you replace vectorizable loops with VecLib calls.

For Fortran code, you can use the VAST-2 precompiler, which recognizes opportunities for vector processing, and generates output containing Fortran statements that use the VecLib library of routines to perform vector processing. You must vectorize C code by hand. The VecLib routines are summarized in Appendix A of this manual and described in the *iPSC®/2 Programmer's Reference Manual*. Chapter 4 of this manual contains program examples that use VecLib calls.

After vectorizing your code, the next step is to verify correct data alignment. The iPSC/2-VX hardware requires that data elements be aligned on their natural boundaries to speed memory access. If data elements are not aligned properly, incorrect results are produced. Use the **-vx** switch when you compile your program to align data automatically and to report situations where misaligned data has been found.

Finally, you need to link your program with the VecLib and vector runtime libraries by using the **-vx** and **-vec** switches.

Figure 2-1 illustrates the program development steps for node processes to be executed on iPSC/2-VX nodes only. Host processes are not affected by the presence (or absence) of vector processor boards in the system. Development of host processes is described in the *iPSC®/2 User's Guide*.

NOTE

Only one process at a time can use a vector board. Structure your program so that each process is allocated to a different vector board.

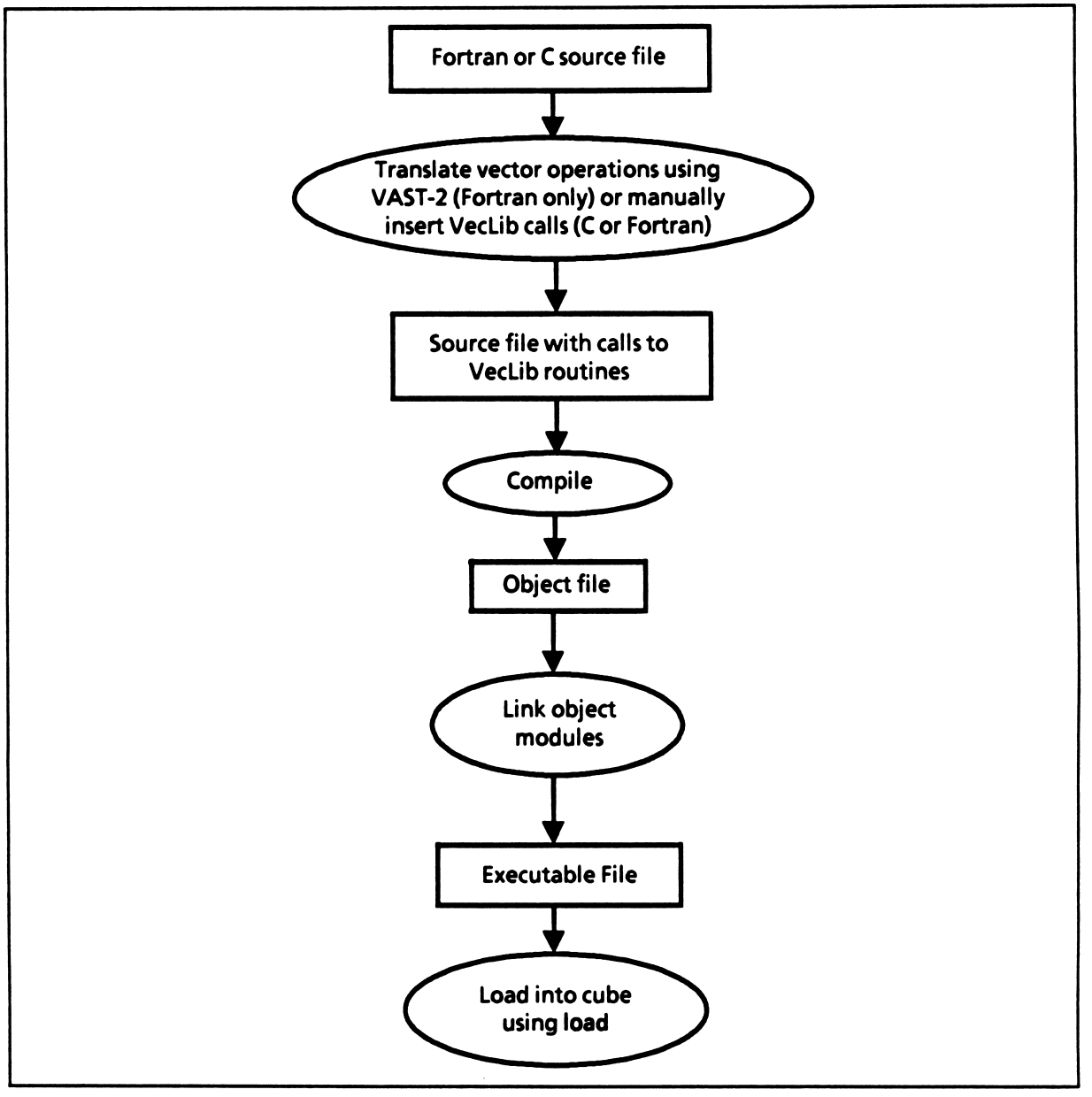


Figure 2-1. iPSC®/2-VX Program Development Steps

VECTORIZING FORTRAN CODE

If your application is written in Fortran, you can use the VAST-2 precompiler to analyze your code and insert appropriate calls to VecLib routines for vectorizable loops. VAST-2 takes Fortran source as its input, and produces source code that is ready to be compiled, linked, and executed. It does this in two steps:

1. The Fortran source is analyzed for sections that can be vectorized and executed on the vector processor board.

2. Those sections are replaced with equivalent calls to VecLib.

Because you know more about the application than is available in the Fortran listing, VAST-2 provides directives for you to convey this additional information to VAST-2. This information can be used by VAST-2 to improve the efficiency of its translation. The VAST-2 preprocessor is a relatively painless way to vectorize Fortran code and access the speed of the vector board. Chapter 4 contains an example of using VAST-2. The *iPSC®/2 VAST2 User's Guide* contains complete information and examples of how to use VAST-2.

After using VAST-2 to produce source code with VecLib calls, you must compile and link it to produce an executable node program. This process is described in the next chapter.

GUIDELINES FOR VECTORIZING C PROGRAMS

To vectorize C code, you must find vectorizable loops in your program, and replace them with appropriate VecLib calls. This section offers guidelines and examples to help you recognize vectorizable loops. The section describes the general rules, indexing, storing into scalars, conditional statements, use of functions, vectorizing outer loops, and miscellaneous transformations.

General Rules

Briefly, the following are the general rules for vectorization (these rules are described in more detail in following sections):

- **for**, **while**, and **do-while** statements may be vectorizable. Within vectorizable constructs, assignment, **if**, **if-else**, **break**, and **continue** statements are allowed. No switch-case statements are allowed.

- You cannot vectorize recursive statements that feed back results from a previous loop pass as input to a later pass unless you can use one of the first order or second order linear recurrence routines.

- Only math functions with vector versions are allowed in vectorizable loops.

- In general, if scalars are defined in a loop, they must be assigned a value before they are used. The exceptions are carry-around scalars, which must be processed in a scalar loop, or wrap-around scalars, which are vectorizable.

- The only reduction functions that are allowed are summation of elements, minimum or maximum element, index of minimum or maximum element, index of absolute minimum or maximum element, dot product, count of true values, and any true values. A reduction function scalar can be used only in the reduction statement within the loop.

- A gathered array cannot appear on the left-hand side in the loop. A scattered array cannot appear anywhere else in the loop.

- The following rules apply to outer loops:

  - You cannot vectorize an outer loop if it either sets the iteration count or conditionally executes any inner loop.

  - The outer loop must use some index at least once.

The following sections describe these rules in more detail and provide examples.

Vectorizable Statements

The following kinds of statements can be contained within vectorizable **for**, **while**, or **do-while** statements:

- Assignment — *a[i]=b[i] + c[i];*

- Conditional assignment — *if(d[i] < 0) a[i]=b[i] + c[i];* or *a[i] = d[i]<0?b[i]:c[i];*

- **goto** *label*

- **if** () {}

- **if**(){} **else**{}

- **break**

- Comments

- **continue**

Understanding Loop Variables

To understand how to vectorize your code, it is important to understand which uses of the variables in a loop are vectorizable and which are not. Variables in a loop can be put into one of three categories:

Scalar	A single value that does not change with each pass through the loop.
Index	An integer that is incremented by a constant amount for each pass through the loop.
Vector	A set of values (referred to by one variable name) located in a contiguous range of memory locations, with a constant increment or "stride" between consecutive elements.

Depending upon the use, any of these types may be vectorizable; the following sections offer guidelines on determining whether a given usage is vectorizable.

The following example of a **for** loop contains one of each type of variable:

```
for(i=0;i<n;i++)  {
      j = j+1
      a[j] = x
}
```

In this statement, the variables are classified as follows:

i, j	index variables
a[j]	vector, where $0 \leq j < n$
x	scalar

Indexing

This section describes the various ways that arrays can be indexed in loops, and describes how they could be vectorized. The categories that are covered are constant increment integers, index expressions, non-linear indexing, last-value saving, indirect addressing, multiply-defined indices, index expressions in several dimensions, and explicit use of indices.

CONSTANT INCREMENT INTEGERS

A constant increment integer (CII) is a variable integer that is incremented by a constant amount for each pass through the loop. It may be defined either in terms of a previously defined CII or its own previous value. It is important to recognize CIIs because they are easily vectorizable. A statement that defines a CII can have one of two forms:

cii1 = cii1 ± invariant_expression
*cii2 = cii1 * invariant_expression ± invariant_expression*

where *invariant_expression* is an expression whose value is constant for all passes of the loop.

For example, the following loop demonstrates some types of CII's that are easily vectorizable because they map directly into the syntax of VecLib calls. The CIIs in this loop are **i, j, k,** and **m**:

```
for(i=0;i<n;i++)  {                          /* this index is a CII */
     j = j + 1;                              /* recurrent CII definition */
     k = i*2 + 3;                 /* CII defined in terms of another CII */
     d[i] = b[k] * c[m] * a[j];
     m = m + 4;                              /* recurrent CII definition */
}
```

This is easily translated using the **svvtp()** call (vector times vector plus vector):

```
svvtvp ( n, &b[4],2, &c[m-1],4, &a[j],1, d,1 );
if (n > 0) {
     j = n+j;
     m = n*4+m;
}
```

In the following example, **j, k,** and **m** are not CIIs:

```
for(i=0;i<n;i++){
     k = i/4;
     m = i*i;
     j = j*2;
     a[j] = b[k] * c[m];
}
```

While it is possible to vectorize this loop, it requires gathering and scattering vectors, and defining intermediate variables as temporary storage for values, as shown later.

INDEX EXPRESSIONS

For an array to be accessed as a vector, at least one of its subscripts must be capable of being translated to the general form:

$$cii2 = cii1 * invariant\_expression \pm invariant\_expression$$

This would include scalar indices, such as i, (equivalent to i * 1 + 0). The following example shows more complicated index expressions for all vectors, all of which, nevertheless, are valid vectors, because they can be translated to the above form.

```
for(i=5;1<n;i++)  {
     a[5*i]= b[i+1] * b[(i-1)*n + m];
}
```

You can replace this with the **svadd()** call:

```
svadd ( n, &b[1], 1, &c[m-n], n, &a[0], 5);
```

NONLINEAR INDEXING

Nonlinear indexes are defined in two ways:

type 1	nonlinear functions of CIIs
type 2	nonlinear functions of themselves

In the following example, *k*,*m*, and *j* are nonlinear indexes:

```
for(i=0; 1<n; i++)  {
     j = j * 2;                    /* type 1 */
     k = i/4;                      /* type 1 */
     m = i * i;                    /* type 2 */
     a[j]= b[k] * c[m];
}
```

You can vectorize type 1 nonlinear indexes by indirectly addressing the array. Type 2, however, may be recursive, and it is best to "split it out", indirectly addressing it in a separate for loop. The following shows how to vectorize the above:

```
static int i_zero=0,i_one=1;
      int i,j,k,m,n;
      float *f1_tmp,*f2_tmp;
      int *i_tmp,*i_vec,*m_vec;

      j = 1; k = 1; m = 1;
```

```
/* dynamically allocate temporary variables with vx__malloc */
  if(n>0)  {
     i_vec = vx_malloc(n*(sizeof (long)));
     m_vec = vx_malloc(n*(sizeof (long)));
     f1_tmp = vx_malloc(n*(sizeof (float)));
     f2_tmp = vx_malloc(n*(sizeof (float)));
     if(i1_tmp == NULL || f1_tmp == NULL
     || i2_tmp == NULL || f2_tmp == NULL){
        perror("Allocation of temp vector(s) failed");
        exit(2);
     }
  }

  iramp ( n, &i_zero, &i_one, i_vec,1 );          /* generate i vector */
  ivmul ( n, i_vec,1, i_vec,1, m_vec,1 );     /* generate m vector (i*i) */
  ivdiv ( n, i_vec,1, &i_four,0, i_vec,1 );       /* generate k vector */
  sgathr ( n, b,1, i_vec,1, f1_tmp,1 );  /* gather b[k], store in f1__tmp*/
  sgathr ( n, c,1, m_vec,1, f2_tmp,1 );  /* gather c[m], store in f2__tmp*/
  svmul ( n, f1_tmp,1, f2_tmp,1, f2_tmp,1 );          /* b[k] * c[m] */
  {int i;
    for(i=0;i<n;i++){                                 /* split out j loop */
        j = j*2;
        i1_tmp[i] = j;
    }
  }
  sscatr ( n, f2_tmp,1, i1_tmp,1, a,1 ); /* scatter results back into a[j] */
  if(n>0){       /* dynamically free memory allocated for temporary variables */
        vx_free(i1_tmp); vx_free(i2_tmp);
        vx_free(f1_tmp); vx_free(f2_tmp);
  }
```

SAVING LAST VALUES

Where necessary, you should set final values of CIIs to ensure that all variables into which values are stored in the original code are given the same values in the vectorized code. Examine the flow of the program unit to see if the final value of an index variable is used outside the loop. If it is not, as is usually the case, the vectorized code will not store a value into the index variable. The following example illustrates this situation:

```
external int i, j;
     ...
for(i=0;1<n;i++)
     a[i] = b[j];
     j = j + 1;
}
```

This can be vectorized as follows:

```
external int i, j;
     ...
dcopy(n, &b[j], 1, &a[0], 1)                    /* vector copy */
if (n > 0) {                                /* must have > 0 iterations */
     i = n + 1;                                      /* save last value */
     j = n + j;                                      /* save last value */
}
```

INDIRECT ADDRESSING

When the subscript for an array is itself a vector, the array is said to be "indirectly addressed", as in the following example:

```
for(i=0;i<n;i++)
   a[ia[i]] = b[ib[i]*6-ABS(ic[i])];
```

To vectorize this, you can use the gather and scatter vector operations. Array references of this type can be vectorized easily if they obey these rules:

- An array into which values are indirectly stored (scattered) may not appear elsewhere in the loop.
- You cannot store values inside the loop in an array indirectly read (gathered).

The following example cannot be vectorized because the second occurrence of **b** violates the second rule.

```
for(i=0;1<n;i++)  {
     a(i) = b(ib(i));
     b(i) = 0;
}
```

If you know that the indexing is nonrecursive (that there are no repeated elements in the indexing array(s)), you can vectorize it despite its apparent recursion.

MULTIPLY-DEFINED INDICES

It is possible to vectorize loops that contain indices that are set more than once, as in the following example:

```
for(i=0; i<n; i++)  {                              /* vectorizable */
     j = j + 3;                                /* first definition of j */
     a[j] = 0;
     j = j + 2;                               /* second definition of j */
     b[j]= 1;
}
```

This is vectorized as follows:

```
sfill ( n, &f_zero, &a[j+3],5 );  /* assume f_zero defined as 0.0 */
sfill ( n, &f_one, &b[j+5],5 );   /* assume f_one defined as 1.0 */
```

INDEX EXPRESSIONS IN SEVERAL DIMENSIONS

An index expression used in more than one dimension of an array (diagonal indexing) is a vectorizable array reference. In the example below, the vector into which values are being stored starts at a[1,1], and skips by n + 1 for each element.

```
float a[n][n];
     .
     .
     .
for(i=0;1<n;i++)
     a[i][i] = b[i];
```

This translates to the simple vector copy statement. The increment determines the next vector element to be operated on in the matrix. In this example, the "n + 1" increment operates on the diagonal of the matrix:

```
scopy(n,b,1,a,n+1);
```

EXPLICIT INDEX USE

When a CII is used outside of array subscripts, you can translate it to a call to a vector ramp function. In the following example, using *i* inside of the **cos** function requires a vector of values from 1 to *n* to be multiplied by the array b, and the use of a temporary array:

```
for(i=0;1<n;i++) {
     a[i] = cos(b[i]*i);
}
```

You can vectorize this as follows:

```
t = vx_malloc(n * sizeof(double));  /* allocate temp storage */
sramp(n, &zero, &one, t, 1);        /* put flt. pt. version of i in t */
svmul(n, t, 1, b, 1, t, 1);         /* multiply b[i] by i, store in t */
svcos(n, t, 1, a, 1);               /* take cos of t, store in a */
vx_free(t);                         /* free temp storage */
```

Storing Into Scalars

Scalar variables have a single unchanging location in memory, as, for example, a simple variable **alpha**. Array reverences whose subscripts contain no CIIs (and thus represent a single location through all passes of the loop) are called "array constants". You can treat array constants similar to the way you treat simple scalar variables.

Scalar variables that are defined in a loop can sometimes be hard to vectorize. Scalar variables that are not redefined in the loop present no difficulty. This section describes how to vectorize nonindex scalars that are defined within a loop.

SCALAR PROMOTION

When a scalar is set to a vector expression, that scalar must be "promoted" to a vector. This requires that you introduce temporary vectors that replace the promoted scalars. Guidelines for dealing with promoted scalars follow:

Saving Last Values of Promoted Scalars

You should save the last value of a promoted scalar (the value the scalar would have had on exiting the original loop) when the value is needed following execution of the loop.

Conditionally Defined Promoted Scalars

If the last value of a conditionally defined promoted scalar is required, you must vectorize that operation as well. For example, to vectorize the following loop you need to use a temporary vector to "promote" the scalar **s**.

```
for(i=0;i<n;i++){
   if(a[i]<=0)continue;
   s = 1/a[i];
   b[i] = sqrt(s) + s;
}
```

In the following vectorization, the **sslt()** and the first **smask()** calls invert the conditional clause (a[i]< =0) because the VecLib scalar routines require the scalar value on the left-hand side of the statement. Notice that the vectorization of this kind of statement is long. To make the vectorization worthwhile, the condition must be such that the body of the loop is executed most of the time. If it is not, you probably should not bother to vectorize it.

```
/* allocate temporary storage space (f1__tmp, f2__tmp, if__vec) with vx__malloc */

                                        /* if !(a[i]<=0), store in if__vec*/
sslt ( n, &f_zero, a,1, if_vec,1 );
      /* invert true and false, store trues in f1__tmp, put 1's elsewhere for divide */
smask ( n, a,1, &f_one,0, if_vec,1, f1_tmp,1 );
                                          /* divide 1/a[i], put in f1__tmp*/
ssdiv ( n, &f_one, f1_tmp,1, f1_tmp,1 );
  /* conditional store; if if__vec true, store divide results in f2__tmp (the s vector) */
scndst ( n, f1_tmp,1, if_vec,1, f2_tmp,1 );
                              /* Save s vector, masking with if__vec, in f1__tmp */
smask ( n, f2_tmp,1, &f_one,0, if_vec,1, f1_tmp,1 );
     svsqrt ( n, f1_tmp,1, f1_tmp,1 );      /* Take square root of s */
svadd ( n, f1_tmp,1, f2_tmp,1, f1_tmp,1 );          /* ( sqrt(s) + s */
                              /* Conditional store; if if__vec is true, store in b */
scndst ( n, f1_tmp,1, if_vec,1, b,1 );

                                        /* free temporary storage space */
```

Scalar Folding

In certain cases, you can eliminate a promoted scalar by forward substitution. You should do this only when it results in the elimination of a vector temporary and does not add operations. For example, the following loop uses a temporary value t:

```
for(i=0;i<n;i++) {
   t = a[i];
   b[i] = t + 1.0/t;
}
```

When you vectorize it, you can eliminate **t** by storing the reciprocal of **a[i]** in **b[i]**, and then adding **a[i]** and **b[i]**, storing the final result back in **b[i]**. At the end, you save the last value of **t** as it would have been, had it been used, in case it is required later.

```
svrecp(n,a,1,b,1);                  /* take reciprocal of a, store in b */
svadd(n,a,1,b,1,b,1);                  /* add a and b, store in b */
if (n>0)t = a[n-1]                        /* save last value of t */
```

CARRY-AROUND SCALARS

Scalars that may be used before they are defined in a loop are called "carry-around scalars. They may or may not be recursive. Vectorizable scalars of this kind are "wrap-around" scalars; other types usually cannot be vectorized.

Wrap-around Scalars

When a carry-around scalar is used to save a calculation from a previous loop pass and does not create recursion, you can vectorize it. These are called "wrap-around" scalars. In the following example, t is a wrap-around scalar:

```
for(i=0;i<n;i++) {
  s = a[i] * a[i]
  b[i] = s + t
  t = s                              /* t holds the value for the next pass */
}
```

This can be vectorized as follows:

```
tmp[0] = t                    /* tmp[0...n-1] == -t; tmp[1...n] == s */
svmul(n,a, 1, a, 1, &tmp[1], 1);              /* square a[i] */
svadd(n,tmp, 1, &tmp[1], 1, b 1);           /* a[i] + a{i-1] */
if (n>0) t = tmp[n-1];                    /* save last value of t */
```

Nonvectorizable Carry-around Scalars

Carry-around scalars that are not wrap-arounds or which cause recursion generally cannot be vectorized. If possible, split out all references to these variables from the rest of the calculation and collect them in a scalar loop. For example, the following loop cannot be vectorized:

```
for(i=0;i<n;i++) {
  a[i] = s + 1/s;
  b[i] = c[i] - a[i] + s;
  s = b[i] + d[i];
}
```

REDUCTION FUNCTIONS

A reduction function is an operation that condenses array operands into one scalar value that characterizes some aspect of the input arrays. Reduction functions that you can vectorize with corresponding VecLib routines are as follows (routines listed with an x in the name indicate that there is a routine for more than one type; for example, **xsum** represents all of the sum routines, **dsum**, **ssum**, **csum**, and **zsum**)):

sum of elements	**xsum**(), **dzasum**(), **csasum**()
maximum value	**xvmax**()
minimum value	**xvmin**()
dot product	**xdot**(), **xdotu**(), **xdotc**()
index of maximum element	**ixmax**()
index of minimum element	**ixmin**()
count of true values	**icount**
any true values	**lany**
index of maximum abs. value	**ixamax**()
index of minimum abs. value	**ixamin**()

Vectorizing Conditional Statements

This section offers guidelines on vectorizing conditional statements in loops. You can vectorize most combinations of conditional assignments, conditional and unconditional forward branching (including arithmetic if's), and block if's.

Forward transfers, such as those using the continue statement, can be vectorized, as shown in the example earlier for a conditionally defined promoted scalar in the section "Storing into Scalars". Following are examples of conditional assignments and block-if statements:

Conditional assignment statements:

Form: **if** (*logical_expression*) *variable* = *expression*;

Example:
```
for(i=0;i<n;i++)  {
     if (b[i] < c[i]-x) a[i] = sqrt(d[i]);
}
```

Vectorized:
```
vx_malloc fl_tmp;          /* allocate temporary storage */
vx_malloc if_vec;
                    /* subtract x from c[i], store in fl_tmp */
svsub(n, c, 1, &x, 0, fl_tmp, 1);
                          /* if b<fl_tmp, store in if_vec */
slt(n, b, 1, fl_tmp, 1, if_vec, 1);
                     /* take square root of d[i], store in if_vec */
svsqrt(n, d, 1, fl_tmp, 1);
     /* for elements where if_vec true, store fl_tmp in a */
scndst(n, fl_tmp, 1, if_vec, 1, a, 1);
```

Block if-statements:

Form:
if (*logical_expression*) *expression*;
else *expression*;
else if (*expression*) *expression*;

Example:
```
for(i=0;i<n;i++) {
     if(a[i]==b[i]) {
          a[i] = 2.0*b[i];
          e[i] = pow(e[i],2.0);
     } else {
          a[i] = 0;
          e[i] = d[i];
     }
}
```

In the vectorization of the following example, it is of interest to note that the **smask()** call performs the **if** and **else** functions at once.

```
                                        /* allocate temporary storage with vx__malloc */

                                        /* if a[i] = =b[i]., store element in if__vec */
seq( n, a,1, b,1, if_vec,1 );
ssmul( n, &f_two, b,1, f_tmp,1 );                          /* 2 times b */
                                /* use if__vec to put multiply results in a, else put 0 in a*/
smask( n, f_tmp,1, &f_zero,0, if_vec,1, a,1 );
svmul( n, e,1, e,1, f_tmp,1 );              /* square e, store in f__tmp */
                                /* use if__vec to put multiply results in e, else put d in e*/
smask( n, f_tmp,1, d,1, if_vec,1, e,1 );
if(n>0){                                                   /* free temporary storage */
   vx_free(l_tmp); vx_free(f_tmp);
}
```

Other conditional statements are not easily handled:

- Switch statements
- Backward transfers
- Transfers out of the loop

The two basic types of conditions are "loop independent" and "loop dependent". Loop independent conditions stay the same for all iterations of the loop. Loop dependent conditions may change for each loop pass.

The following example shows a vectorizable loop independent conditional assignment:

```
for (i=0;i<100;i++) {
   b[i] = b[i]+1.0
   if (n==1) a[i] = b[i]+2.0;
}
```

Vectorized, this becomes two statements:

```
ssadd(100, &one, b,1,b,1);              /* add i (&one) to b, store in b */
                                        /* if n = =1, add 2 (&two) to b[i], store in a[i] */
if (n==1) ssadd(100, &two, b,1,a,1);a[i] = b[i]+2.0;
```

The **if** clause (n = = 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. You can 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.

A loop dependent condition (the if clause may change with each loop pass) is shown in the following loop:

```
for (i=0;i<n;i++) {
  if (c[i]<0) d[i] = e[i]+f[i];
}
```

To vectorize this, you need to perform the calculation across all elements and then store only the useful results into the result array. This can be vectorized as follows:

```
                                    /* allocate temporary storage (if_vec, f_tmp) with vx_malloc */

ssgt ( n, &f_zero, c, 1, if_vec, 1);        /* if c[i]>0, store in l_tmp */
svadd ( n, e,1, f,1, f_tmp,1 );          /* add e[i] to f[i], store in f_tmp */
                                    /* conditionally store f_tmp in d, for true elements of l_tmp */
scndst ( n, f_tmp,1, if_vec,1, d,1 );

                                    /* free temporary storage with vx_free */
```

Mathematical Functions

Only calls to mathematical functions can be vectorized. In general, mathematical functions can be applied to vector arguments as well as scalars.

Vectorizing Outer Loops

You should examine all possible loops where you have nested loops. It is possible to vectorize outer loops with multiple inner loops. It is important to keep track of data dependency when vectorizing nested loops.

CHOOSING THE BEST LOOP

Use the following criteria to pick the best loop in a nest:

- Vector Length
- Number of single-precision vectors accessed with a nonunit 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 the one to vectorize, leaving the inner loop scalar.

```
for(j=0;j<m;j++) {                    /* Outer loop is vectorizable */
  a[j] = b[j]*2.0;
  x = a[j]-c[j];
  for(i=0;i<n;i++) {                /* Inner loop is not vectorizable */
     d[i+1,j] = -d[i,j]/x*e[i,j];                    /* Recursive */
  }
  b[j] = e[j]-2.0;
}
```

NONVECTORIZABLE OUTER LOOPS

There are situations in which you cannot vectorize the outer loop. For example, when the iteration count of an inner loop is not constant for all passes of the outer loop, then the outer loop cannot be vectorized (you may still be able to vectorize the inner loops). Below, the iteration count for both of the inner loops change with each pass of the outer loop, thus making it impossible to vectorize the outer loop.

```
for(j=0;j<n;j++) {                  /* Outer loop is not vectorizable */
  for(i=0;i<j;i++) {
     a[i,j] = a[i,j]*b[i,j];  /* Inner loop not vectorizable */
   }
   for(i=0;i<n;i++) {                /* Inner loop is vectorizable */
     c[i,j] = 0;
   }
   for(i=0;i<n;i++) {
     d[i+1,j] = -d[i,j]/x*e[i,j];                    /* Recursive */
   }
  b[j] = e[j]-2.0;
}
```

Sometimes recursion along the inner loop also prevents vectorization of the outer loop. For example, the inner loop below is recursive, and so cannot be safely vectorized, preventing vectorization of the outer loop as well.

```
for(j=0;j<m;j++) {                                  /* Not vectorizable */
  for(i=0;i<n;i++) {                                /* Not vectorizable */
     a[i+1] = a[i] * b[i,j];
  }
}
```

When an inner loop is executed conditionally, the outer loop is not vectorized, as in the following example:

```
for(i=0;i<n;i++) {                          /* Not vectorizable */
  if (a[i] > 0) {
     for(j=0;j<n;j++) {                     /* Inner loop is vectorizable */
        b[i,j] = a[i]*j;
     }
  }
}
```

Finally, loops that 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.

```
for(i=0;i<n;i++) {                          /* Not vectorizable */
     for(j=0;j<m;j++) {                     /* Inner loop is vectorizable */
     a[j] = b[j] + c[j];
   }
}
```

Loop Nest Collapse

Under certain restrictive conditions, you can collapse loop nests 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, this is very easy to do. The collapse criteria are as follows:

- 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 with a stride of one. The declarations of these arrays must also conform; the first *n*-1 dimensions must be identical.

You can collapse this loop and vectorize it as a single vector operation of length 120:

```
float a[12][10], s_zero = 0;
     .
     .
for (j=0;i<10;j++) {
   for (i=0;i<12;i++) {
      a[i][j] = 0.0;
   }
 }
```

You can replace this with the single statement:

```
sfill(10*12, &s_zero, a, 1);
```

COMPILING AND LINKING YOUR PROGRAM 3

INTRODUCTION

This chapter describes how to compile and link your vectorized Fortran or C program, and provides sample makefiles.

COMPILING AND LINKING IN ONE STEP

You can compile and link your program with one command if you wish to. The **f77** command with the **-vx**, **-vec**, and **-node** switches will do this. By default, this links in the double precision routines. If you need the single precision routines only, you need to add the **-single** switch. If you need both single and double precision routines and/or the complex routines, you need to add one of the switches **-both**, **-all**, or **-complex** (all three have the same effect when linking). The following examples show these commands.

Fortran programs (assume the source file is named *node.f* and the resulting file is named *node*):

- For double precision vector operations:

```
f77 -o node  node.f  -vx  -vec  -node
```

- For single precision vector operations:

```
f77 -o node  node.f -single -vx  -vec  -node
```

- For both single and double precision and complex vector operations:

```
f77 -o node  node.f -all -vx  -vec  -node
```

C programs (assume the source file is named *node.c* and the resulting file is named *node*):

- For double precision vector operations:

```
cc -o node  node.f  -vx  -vec  -node
```

- For single precision vector operations:

```
cc -o node  node.f -single -vx  -vec  -node
```

- For both single and double precision and complex vector operations:

```
cc -o node  node.f -all -vx  -vec  -node
```

If you are using VAST-2 to vectorize your Fortran programs, the **f77** command recognizes .v extensions as source for VAST-2, so you can vectorize, compile, and link your program with the same Fortran commands as shown above, except the source file would be named *node.v* rather than *node.f*.

COMPILING AND LINKING IN SEPARATE STEPS

You may prefer to execute each of the steps individually. These steps are:

1. (Optional) Automatically vectorize Fortran programs only with the VAST-2 precompiler.
2. Compile the source module(s).
3. Link in the appropriate libraries.

The following examples all use *node* as the prefix of the program name and show how you would execute each of these steps.

1. For a Fortran program, either of the following two commands executes the optional VAST-2 precompiler to vectorize the source code (file named *node.v*):

```
vast2  -o node.f  node.v

f77  -v  node.v
```

   In both cases, the output file is a Fortran source file named *node.f*, in which which calls to VecLib functions are substituted for vector operations in the instruction stream. VAST-2 supports a variety of options defined by switches described in the *iPSC®/2 VAST2 User's Guide*.

2. Compile the source module. Both the **-c** and the **-vx** switches are required, and have the following functions:

-c	compiles source module only
-vx	aligns double precision and complex variables on 64-bit boundaries.

   The following Fortran example uses an input file with the **.f** extension. This assumes either that you have put in VecLib statements by hand or the file is the output of the VAST-2 precompiler:

   ```
   f77  -c  -vx  node.f
   ```

   To execute the VAST-2 vectorizer and then to compile in one command, you would issue the following command (notice that the file requires a **.v** extension to execute VAST-2):

   ```
   f77  -c  -vx  node.v
   ```

   The following command compiles a C source file (VecLib statements must be inserted by hand):

   ```
   cc  -c  -vx  node.c
   ```

3. Link in the required libraries. This command uses the same switches that you would use if you were linking and compiling in a single step, but the input file is an object module rather than a source file. The required switches are **-vx**, **-vec**, and **-node**. In addition, the routines you require from VecLib must be linked in by default, the double precision routines are linked. A switch is required to link either the single precision routines or all of the routines (including complex routines). The library switches are defined as follows:

   Required switches:

-vx	invokes the **vxld** command to place data in vector board memory and code in node memory.
-vec	links in the appropriate VecLib routines.
-node	links in the node communication and utility routines.

If you are using the optional Weitek coprocessor, you must use the **-sx** switch. If you would like runtime information about data alignment, you can use the **-vecdb** switch instead of the **-vec** switch.

VecLib switches (use one or default to double):

-double	(the default) links the double precision routines for vectorization.
-single	links the single precision routines for vectorization.
-both **-all** **-complex**	links all of the routines (double and single precision routines as well as complex routines) for vectorization. These switches require the 16K program memory option. These switches have the same effect (use only one).

The following examples (the first for Fortran, the second for C) use the **-both** switch to link in all of the VecLib routines, and in addition, use the compilers' **-o** switch to name the output file *node*:

```
f77  -o node node.o  -both  -vx  -vec  -node

cc  -o node node.o  -both  -vx  -vec  -node
```

NOTE

If your program is too big for vector memory the linker will report the message: "Can't allocate section *name* into owner vxram ld fatal: Error(s). No output written to *program_name*." Refer to the section on "Using the **vxld** Command" for more information.

NOTE

Use **-vecdb** instead of **-vec** to perform runtime alignment checks for vector data.

SAMPLE MAKEFILES

You can simplify and automate the development process by placing the commands described above in a makefile and using **make**, a UNIX tool. This ensures that the correct build steps are performed in the correct order. Additionally, whenever an input file is modified, **make** determines the minimum number of steps required to rebuild the application. For more information on make, see the *Unix System V Programmer's Reference Manual.*

The following is an example of a makefile for Fortran source code that can be used to generate both System Resource Manager (or host) and node object modules for a single precision application.

```
F77 = f77
VFLAGS = -single
F77FLAGS =  -vx
HLDFLAGS =  -host
NLDFLAGS =  -single  -vx  -vec  -node

.SUFFIXES: .v .f .o

.v.f:
   $(F77) -V $(VFLAGS) $*.v

.f.o:
   $(F77) -c $(F77FLAGS) $*.f

HOST_OBJECTS =  host.o

NODE_OBJECTS =  node.o

all:    host node

host:           $(HOST_OBJECTS)
                $(F77) -o $@ $(HOST_OBJECTS) $(HLDFLAGS)

node:           $(NODE_OBJECTS)
                $(F77) -o $@ $(NODE_OBJECTS) $(NLDFLAGS)
```

The following is an example of a makefile for C source code that can be used to generate both system resource manager (or host) and node object modules for a single precision application.

```
CC = cc
CFLAGS =  -vx
HLDFLAGS =  -host
NLDFLAGS =  -single  -vx  -vec  -node

.SUFFIXES: .c .o

.c.o:
   $(CC) -c $(CFLAGS) $*.c


HOST_OBJECTS =  host.o

NODE_OBJECTS =  node.o


all:    host node

host:           $(HOST_OBJECTS)
                $(CC) -o $@ $(HOST_OBJECTS) $(HLDFLAGS)

node:           $(NODE_OBJECTS)
                $(CC) -o $@ $(NODE_OBJECTS) $(NLDFLAGS)
```

In the Fortran example, the **VFLAGS** and **F77FLAGS** entries define the flags to be used for VAST-2 and for Fortran respectively. The VAST-2 flag, **-single**, is specified in this example. The specified Fortran flags are the same as shown in step 2.

In the C example, the **CFLAGS** entry define the flags to be used for **cc**. (VAST-2 cannot be used with C programs.)

The **SUFFIXES** entry defines the significant file suffixes. In the Fortran makefile **.v** is the suffix for the Fortran source file used as input to VAST-2, **.f** for the VAST-2 output file to be processed by the Fortran compiler, and **.o** for the object module produced by the Fortran compiler.

In the C makefile, **.c** is the suffix for the C source file used as input to the C compiler, and **.o** for the object module produced by the C compiler.

The **.v.f** entry for the Fortran makefile specifies that the make utility is to execute the **vast2** command on the next line whenever a **.f** file is to be produced from a **.v** file. Because VAST-2 cannot be used with C, the C makefile does not include this line.

The **.f.o** entry in the Fortran makefile specifies that the **f77** command is to be executed whenever a **.o** file is to be produced from a **.f** file. The **.c.o** entry in the C makefile specifies that the **cc** command is to be executed whenever a **.o** file is to be produced from a **.c** file.

The HOST_OBJECTS and NODE_OBJECTS entries list the object modules that are needed to build a complete program. The host program is *host.o*.

The **all:** entry specifies that the make utility is to build a new version of both the host and node programs (this is the default; it is possible to specify only one or the other).

The **host:** entry specifies that the host program is to be rebuilt if any of the HOST_OBJECTS have been modified more recently than the timestamp on the current version of host.

The **node:** entry specifies rebuilding the node program. The resulting commands are the same as the examples shown for separately linking the files. Note that in place of the file name *node.o*, the link command will link in all of the NODE_OBJECTS. In this example, the effects are the same. However, if the node program requires a separately compiled module in addition to *node.o*, its name need only be added to the list in NODE_OBJECTS to be included in the link.

ADVANCED PROGRAMMING TOPICS 4

INTRODUCTION

This chapter describes several features of the iPSC/2 system and software that allow the advanced programmer to improve the performance of certain kinds of applications. The following topics are covered:

- Using static (fast) memory to improve performance.
- Using the **vxld** command in large applications to specify how data is allocated to memory.
- Using the fast copy routines in Fortran programs to copy data between node and VX memory.
- Using the **vx_malloc** routines in C programs for dynamic allocation of VX memory.
- Using the asynchronous routines to improve performance.
- Improving the performance of vector programs processed with VAST-2.

USING STATIC (FAST) MEMORY

It may be possible to improve the performance of a program by allocating data to the static memory on the vector processor. The vector processor board has 16K bytes of static memory in addition to the 1M byte of dynamic memory. The vector operating system uses most of this space, but approximately 5K bytes of fast memory are available to you. This block of memory has been given the label **vpfast**. This is a Fortran common block label, the equivalent of a global variable in C.

Because the access time to static memory is less than half that of dynamic memory, any vector operation that uses operands coming from static memory will run significantly faster than if all the operands were in dynamic memory. Therefore, it might be advantageous to load repetitively-used vectors and temporary work vectors into **vpfast**.

Using vpfast in Fortran Applications

If your application is written in Fortran, the **vpfast** common block is defined in the INCLUDE file *vpnode.h*. You must either include this file in your source, or write the definition into any source routine that uses the common block. The common block definition as follows:

```
INTEGER*4  VP$FAST  (4096)
COMMON  /VPFAST/  VP$FAST
```

NOTE

VP$FAST is defined as being 16K bytes long, but only about the first 5K bytes actually reside in static memory. The remainder spills over into dynamic memory. While it is the first 5K bytes that are most important, avoid loading more than 16K bytes into the array.

There are three ways to use **vpfast** with Fortran.

- Use VAST-2 with the **-vpfast** switch . In this case, VAST-2 assigns temporary vectors to the **vpfast** common block automatically. Following is an example of how to use the **-vpfast** switch:

```
vast2  -vpfast  -o file.f  file.v
```

- Copy the vectors into fast memory before using them. You should only copy a vector into the common block when it can be used several times. Following is an example of how copying vectors works:

```
      INCLUDE 'vpnode.h'
      DOUBLE PRECISION FAST (100)
      EQUIVALENCE (FAST(1), VP$FAST (1))
      DOUBLE PRECISION ALPHA (100)
            .
            .
      CALL DCOPY (100, ALPHA,1,FAST,1)
            .
C  References to ALPHA are replaced with references to FAST
            .
      CALL DCOPY (100,FAST,1,ALPHA,1)
            .
            .
```

- Equate a user vector with **vpfast** using an equivalence statement. Use this method only in applications with a single temporary work vector or only one subroutine that has vector commands where all the vectors can be identified. Following is an example that illustrates this method:

```
INCLUDE 'vpnode.h'
DOUBLE PRECISION WORK (1000)
EQUIVALENCE (VP$FAST(1), WORK(1))
    .
    .
```

Using vpfast in C Applications

If your application is written in C, you need to define **vpfast_** as a global variable in your program if you want to use the available static memory. For example, this statement defines **vpfast_** as an array of type float with a length of 16K bytes (or 4096 32-bit floats):

```
extern float vpfast_[4096];
```

The underscore is required; the type depends on the application, and the size depends on the type.

One way to use this memory in a C program is to copy the vectors into fast memory (the available static memory is approximately 5K bytes) before using them. You should only copy a vector into this memory when it can be used several times. The following example defines **vpfast** as a structure to allow two vectors to be copied into static memory, and then uses **dcopy()** to copy the vectors:

```
extern struct vpfast  {
     float x[500];
     float y[500];
     float pad[3096];          /* to make vpfast 16K bytes long */
     } vpfast_;
float alpha[500];
float beta[500];
     ...
dcopy(500, alpha, 1, vpfast_.x, 1);
dcopy(500, beta, 1, vpfast_.y, 1)
     ...
dcopy(500, vpfast_.x,1, alpha, 1);
dcopy(500, vpfast_.y, 1, beta, 1);
```

In this example, notice that the two useful arrays within the structure total just under the available memory size of 5K bytes. This ensures that all of the data in the arrays will be in fast memory.

USING THE "VXLD" COMMAND

If you compile and link your program as described in the previous chapter, iPSC/2-VX attempts to load all program data onto the vector board, which has 1Mbyte of memory. If program data occupies less than 1 Mbyte of memory, nothing additional need be done. If there is more than 1 Mbyte of program data, attempting to link the program returns an error. The **vxld** command, used before linking, allows you to specify which data blocks need to go on the vector board.

You can use the **vxld** command to make the most efficient use of available vector memory by placing only specified program sections on the vector board memory. The **vxld** command produces a link directive file to **ld** specifying where data will reside. Vector operands for the VX must reside in VX memory. The **vxld** switches that specify *arg* as a parameter take a quoted list of names of object modules (**.o** files) as input. Each object file can consist of up to four sections:

.text	executable code
.bss	unintialized data
.comm	Fortran common blocks or C external variables
.data	initialized data

Many of the following switches are used to place sections either in the VX memory or the node memory. (The **.text** sections always reside in node memory.) The available switches for the **vxld** command are as follows:

-b *arg*	places the **.bss** sections of the argument(s) **.o** files on the vector board
-c *arg*	places the .comm sections of the argument(s) **.o** files on the vector board
-d *arg*	places the .data sections of the argument(s) **.o** files on the vector board
-e *arg*	places the **.bss**, **.comm**, and **.data** sections of the argument(s) **.o** files on the vector board
-o *filename*	places the output into the file named in *filename*.
-single	links single precision microcode
-double	(the default) links double precision microcode
-all **-both** **-complex**	link all VecLib routines (single and double precision and complex microcode). These switches are equivalent and require 16K program memory.
[**--**] *args*	places the **.bss**, **.comm**, and **.data** sections of the argument(s) **.o** files on the vector board

Fortran Example

The following example demonstrates how to use **vxld** to place selected data sections of a Fortran program on the vector board memory.

The program is too large for the vector board memory and only some of the data needs to be on the vector board.

```
program big
double precision a(100,100,3),b(100,100,3),c(100,100,3)
double precision d(100,100,3),e(100,100,3),f(100,100,3)
....
end
```

You issue the command that compiles and links:

```
f77 -o big big.f -vx -vec -node
```

In this case, the program fails during the link because it is too big to fit on the vector board memory and it returns the error message:

```
Can't allocate section [name] into owner vxram ld fatal:
Error(s).  No output written to main.
```

Because all of the data will not fit on the vector board, you can decide which of the arrays it will be most effective to have on the vector board; in this case, assume you want arrays a, b, and c in vector memory. To do this, you can take the following steps:

1. First make the following changes to the program *big* by creating a common block to partition the data between the node board and vector board.

```
program big
double precision a(100,100,3),b(100,100,3),c(100,100,3)
double precision d(100,100,3),e(100,100,3),f(100,100,3)
common/bigvxarrays/a,b,c
....
end
```

2. Create a separate file (here named *blockdata*) for the common block *bigvxarrays*.

```
blockdata bigvxarrays
double precision a(100,100,3),b(100,100,3),c(100,100,3)
common/bigvxarrays/a,b,c
end
```

3. Use **vxld** to put just the common block *bigvxarrays* on the vector board. The file *big* is an output file for **vxld** and an input and output file for the **f77** command which links the program.

```
f77  -c -vx big.f bigvxarrays.f
vxld -o big bigvxarrays.o
f77  -o big big big.o  -lvxvec  -lvx  -node
```

Now, the only data on the VX board is the data in the file *bigvxarrays.o* (i.e. the common block *bigvxarrays*, arrays a, b, and c).

Notice that using **vxld** explicitly requires that you specifically link in the libraries otherwise loaded when you invoke the **-vx** and **-vec** switches of the **f77** command. The **-l** switch links in the library, and automatically prepends **lib** to the given name, and appends **.a** to the name. Thus, the switch **-lvxvec** links in the library **libvxvec.a** and the switch **-lvx** links in **libvx.a** (libraries are described in Chapter 1).

A makefile for the Fortran program follows:

```
VFLAGS=          -double
F77FLAGS=        -vx
LDFLAGS=         -double -lvxvec -lvx -node

.SUFFIXES: .v .f .o
.v.o:
        $(F77) $(VFLAGS) $(F77FLAGS) -c $<
.f.o:
        $(F77) $(F77FLAGS) -c $<

SOURCE=          big.f bigvxarrays.f

N_OBJECTS=       big.o
V_OBJECTS=       bigvxarrays.o

all:    big

big:    $(N_OBJECT) $(V_OBJECTS)
        vxld -o $@  $(V_OBJECTS)
        $(F77) -o $@ $@ $(N_OBJECTS) $(LDFLAGS)
```

C Example

The following example demonstrates how to use **vxld** to place selected data sections of a C program on the vector board memory.

The program is too large for the vector board memory and only some of the data needs to be on the vector board.

```
double a[100]100][3],b[100][100][3],c[100][100][3];
double d[100][100][3],e[100][100]3],f[100][100][3];
    ....
main() {
    ....
}
```

You issue the command that compiles and links:

```
cc -o big big.c -vx -vec -node
```

In this case, the program fails during the link because it does not fit on the vector board memory and it returns the error message:

```
Can't allocate section [name] into owner vxram ld fatal:
Error(s).  No output written to big.
```

Because all of the data will not fit on the vector board, you can decide which of the arrays it will be most effective to have on the vector board; in this case, assume you want arrays a, b, and c in vector memory. To do this, take the following steps:

1. first make the following changes to the program *big* to partition the data between the node board and vector board.

```
extern double a[100]100][3];
extern double b[100][100][3];
extern double c[100][100][3];
double d[100][100][3],e[100][100]3],f[100][100][3];
   ....
main()  {
   ...
}
```

2. Create a separate file (here named *bigarrays.c*) defining the data declared external above:

```
double a[100]100][3];
double b[100][100][3];
double c[100][100][3];
```

3. Use **vxld** to put the data in *bigvxarrays* on the vector board. The file *big* is an output file for **vxld** and an input and output file for the **cc** command which links the program.

```
cc -c -vx  big.c  bigarrays.c
vxld -o big  bigarrays.o
cc  -o big  big  big.o  -lvxvec  -lvx  -node
```

Now, the only data on the VX board is the data in the file *bigarrays.o*.

Notice that using **vxld** explicitly requires that you specifically link in the libraries otherwise loaded when you invoke the **-vx** and **-vec** switches of the **cc** command. The **-l** switch links in the library, and automatically prepends **lib** and appends **.a** to the given name. Thus, the switch **-lvxvec** links in the library **libvxvec.a** and the switch **-lvx** links in **libvx.a** (libraries are listed and described in Chapter 1).

A makefile for this program follows:

```
VFLAGS=         -double
CCFLAGS=        -vx
LDFLAGS=        -double -vx -vec -node

.SUFFIXES: .c .o
.c.o:
        $(CC) $(CCFLAGS) -c $<

SOURCE=         big.c bigarrays.c

N_OBJECTS=      big.o
V_OBJECTS=      bigarrays.o

all:    big

big:    $(N_OBJECT) $(V_OBJECTS)
        vxld -o $@  $(V_OBJECTS)
        $(CC) -o $@ $@ $(N_OBJECTS) $(LDFLAGS)
```

USING THE FORTRAN FAST COPY ROUTINES

There may be some large applications containing many large vectors that all need to be on the vector board at some point, but for which there is not enough vector memory at any one time. The **vxld** command allows you to place certain vectors in vx memory, but does not allow for transferring information between vx and node memory within the process. The fast copy routines allow you to do this. There are three sets of routines: vector copy, gather, and scatter. For each of these, there is one routine for double precision word length and one for single precision. The names of these routines are:

mscopy	msgathr	msscatr
mdcopy	mdgathr	mdscatr

The copy routines (**mxcopy**) use the same calling sequences as the **xcopy** VecLib routines documented in the *iPSC®/2 Programmer's Reference Manual* (a synopsis is shown in the table in Appendix A of this manual). Parameters are the length of the vector (*n*), the address of the source vector (*x*), the increment of the source vector (*incx*), the address of the destination vector (*y*), and the increment of the destination vector (*incy*).

The gather and scatter routines (**mxgathr** and **mxscatr**) differ from the **xgathr** and **xscatr** routines in the calling sequence. The **mxgathr** and **mxscatr** routines require as parameters the length of the vector (*n*), the address of the source vector (*x*), an integer vector of offsets in the source or destination vector (*iy*), and the address of the destination vector (*y*). These routines assume an increment of 1 in both source and destination vectors and the offset's vector.

The source for the fast copy routines is included in the node software for the system, and is automatically included when you when you link your application program with the **-node** switch.

These routines were designed to minimize the overhead of swapping data to and from the node memory. However, it is a time-consuming process, so if it is possible to partition your data so you do not need to swap, it is desirable. If you must swap data several times, however, another way to reduce overhead is to partition vx memory so that you can be operating one set of data while you are copying another set of data in or out.

Chapter 6 of this manual contains reference pages for these routines.

MEMORY MANAGEMENT SUPPORT FOR C ROUTINES

For C programmers accustomed to dynamic memory allocation, a set of calls that operate like the UNIX malloc(3C) calls are included for vx memory, and allow you to get memory on the vector board without statically allocating it with the **vxld** command. The names of these calls are as follows:

vx_malloc
vx_free
vx_realloc
vx_calloc
vx_cfree
vx_sbrk
vx_brk

These calls have the same semantics as the UNIX calls of the same names without the **vx_** prefix. Chapter 6 of this manual provides a reference page for **vx_malloc**, **vx_free**, **vx_realloc**, **vx_calloc**, and **vx_cfree**, and another for **vx_brk** and **vx_sbrk**.

All of the VX memory that is not taken up by data in the object files is in a pool of dynamically allocatable memory, which can be manipulated by the malloc calls. These calls guarantee that the data will be properly aligned.

ASYNCHRONOUS ROUTINES

For all subroutines (routines that do not return a value) in both Fortran and C, there is a corresponding asynchronous subroutine that has the same calling sequence as the original, but there is an underscore before the name of the routine. These routines reduce the call overhead for VecLib routines, particularly increasing the performance for small vectors. The asynchronous routines queue the vector operation on the vector board and return to the caller before the vector operation is complete.

If you are using VAST-2 to vectorize your Fortran program and you want it to use asynchronous routines where appropriate, when you invoke the **vast2** command, you must use the **-opton = w** switch on the command line.

If your program is Fortran, the easiest way to use these routines is to use the VAST-2 vectorizer, which will insert these calls automatically where appropriate and also insert calls to **vpwait()**.

If you are calling the asynchronous routines explicitly (without using VAST-2), you must use calls to the VX utility **vpwait()** to synchronize the VX board with the 386 microprocessor.

To increase the speed of the asynchronous routines, scalar arguments must reside on the vector board (unlike synchronous versions, where scalar arguments need not reside on the vector board). The consequences of this are that expressions are not allowed, because when expressions are evaluated, the results are put on the stack, and the stack is not on the vector board. In C, this means that the data and vectors must be static, not dynamic, because these also reside on the stack.

Because these routines are asynchronous, you must use **vpwait()** in the program when the results of the asynchronous routine are required by the program. This causes the program to pause in its execution until the asynchronous routine is done.

IMPROVING PERFORMANCE OF VAST-2 OUTPUT (FORTRAN ONLY)

If your application is written in Fortran and you vectorize with VAST-2, you can improve the performance of the output by inspecting it to see if any calls to **vpwait()** are unneeded and can be removed.

For example, for a DO loop containing several vector calls, VAST-2 puts a **vpwait()** at the bottom of the loop, requiring a wait at the end of each loop. By moving the **vpwait()** call outside the CONTINUE, you can reduce the number of waits. As another example, when VAST-2 sees a DO loop containing several calls followed by a second DO loop, also containing several calls, VAST-2 cannot look beyond the two blocks to optimize placement of the **vpwait()**. In some cases, you can remove all of the waits except the one at the end of the last loop.

It is possible to improve performance from 10-15% by doing this.

iPSC®/2-VX PROGRAM EXAMPLES 5

INTRODUCTION

This chapter contains program examples illustrating applications for the iPSC/2-VX. Similar examples can be found in on-line files located in the */usr/ipsc/examples* directory. You can read an explanation of these examples in their associated source files and README files.

PROGRAM EXAMPLE 1 – loop7.v

The following example shows how VAST-2 works. VAST-2 takes the source listing shown in Listing One and analyzes the Fortran DO loops. If the loops can be safely vectorized, VAST-2 translates the DO loop into calls to VecLib routines as shown in Listing Three. Listing Two shows the diagnostic listing provided by VAST-2. The three listings below were generated by executing the following command:

```
vast2 -o loop7.f loop7.v > loop7.L
```

Several features of VAST-2 can be seen in listing three of the VAST-2 translation:

1. Automatic generation of temporaries
2. Stripmining for loops of indeterminate length
3. Efficient generation of triads

See the *iPSC®/2 VAST2 User's Guide* for further information.

Listing One – loop7.v (source input)

```
      subroutine eqofstate( n, u, x, y, z, r, t )
      double precision u( * ), x( * ), y( * ), z( * ), r, t
      do 7 m = 1, n
           x( m ) = u( m ) + r * ( z( m ) + r * y( m ) )
     +     + t * (u(m + 3) + r * (u(m + 2) + r * u(m + 1))
     +     + t * (u(m + 6) + r * (u(m + 5) + r * u(m + 4))))
7     continue
      return
      end
```

Listing Two – loop7.L (VAST-2 diagnostic listing)

```
1.          subroutine eqofstate( n, u, x, y, z, r, t )
2.          double precision u( * ), x( * ), y( * ), z( * ), r, t
3.          do 7 m = 1, n
4.               x( m ) = u( m ) + r * ( z( m ) + r * y( m ) )
5.          +    + t * (u(m + 3) + r * (u(m + 2) + r * u(m + 1))
6.          +    + t * (u(m + 6) + r * (u(m + 5) + r * u(m + 4))))
7.     7    continue
8.          return
9.          end


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

     7   M            3      7  1  VECTORIZED                      N

--------------------  EVENT SUMMARY FOR ROUTINE EQOFSTAT------------------
WARNING MESSAGES          --      0       SYNTAX ERRORS              --      0
TRANSLATION DIAGNOSTICS   --      0       DATA DEPENDENCY CONFLICTS  --      0
LOOPS EXAMINED            --      1       LOOPS TRANSLATED           --      1
```

Listing Three – loop7.f (VAST-2 generated code)

```
      subroutine eqofstate( n, u, x, y, z, r, t )
C...TRANSLATED BY VAST-2 2.23A2  14:11:36   3/17/88
      INTEGER J1S,J2S
      DOUBLEPRECISION D1V(1),D2V(1),D3V(1)
      REAL QQQ
      COMMON/Q1VASTCM/ QQQ(10000)
      EQUIVALENCE (QQQ(1),D1V),(QQQ(3321),D2V),(QQQ(6641),D3V)
      double precision u( * ), x( * ), y( * ), z( * ), r, t
      DO 77001 J1S = 0, N-1, 1660
         J2S = MIN0 ( N-J1S,1660 )
         CALL DSVTVP ( J2S, R, Y(1+J1S),1, Z(1+J1S),1, D1V(1),1 )
         CALL DSVTVP ( J2S, R, D1V(1),1, U(1+J1S),1, D1V(1),1 )
         CALL DSVTVP ( J2S, R, U(2+J1S),1, U(3+J1S),1, D2V(1),1 )
         CALL DSVTVP ( J2S, R, D2V(1),1, U(4+J1S),1, D2V(1),1 )
         CALL DSVTVP ( J2S, R, U(5+J1S),1, U(6+J1S),1, D3V(1),1 )
         CALL DSVTVP ( J2S, R, D3V(1),1, U(7+J1S),1, D3V(1),1 )
         CALL DAXPY ( J2S, T, D3V(1),1, D2V(1),1 )
         CALL DSVTVP ( J2S, T, D2V(1),1, D1V(1),1, X(1+J1S),1 )
77001 CONTINUE
      return
      end
```

PROGRAM EXAMPLE 2 – timedaxpy.f

This program executes on a single vector processor board and writes its timing results to the standard output. It compares the timing of the synchronous DAXPY routine to the semi-synchronous version. Because of the coarse granularity of the timing routine "MCLOCK", many calls must be timed to get an accurate timing. By reporting the results in Kflops (thousands of floating point operations per second), this program illustrates the improved performance of the semi-synchronous version for short vectors.

```
      program timedaxpy
      double precision dx( 8192 ), dy( 8192 ), da, dzero
      integer nsize( 15 ), n
      integer itime, itemp, ilst
      include 'fcube.h'
      include 'veclib.h'
      data nsize
     x /0,1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192/
c
c setup dummy constants - this is a timing test
c
      da = 1.0d0
      dzero = 0.0d0
      call _dfill( 8192, dzero, dx, 1 )
      call dfill( 8192, dzero, dy, 1 )
      do 1000 j = 1, 15
         n = nsize( j )
c
c time the synchronous veclib version
c
         itime = mclock()
         do 100 i = 1, 1000
            call daxpy( n, da, dx, 1, dy, 1 )
 100     continue
         itime = mclock() - itime
         write( *, 2222 ) itime/1000.,n,n*2000./itime
c
c time the semi-synchronous version
c
         itime = mclock()
         do 200 i = 1, 1000
            call _daxpy( n, da, dx, 1, dy, 1 )
 200     continue
         call vpwait
         itime = mclock() - itime
         write( *, 3333 ) itime/1000.,n,n*2000./itime
 1000 continue
 2222 format('  VX milli = ',f8.4,' n = ',i6,' Kflops = ',f10.5 )
 3333 format(' _VX milli = ',f8.4,' n = ',i6,' Kflops = ',f10.5 )
      end
```

PROGRAM EXAMPLE 3 – timesaxpy.c

This example is a C program similar in function to the previous example, except that it compares the timing of the the single precision routine **saxpy**() to the semi-synchronous version of the same routine. Because of the coarse granularity of the timing routine "MCLOCK", many calls must be timed to get an accurate timing. By reporting the results in Kflops (thousands of floating point operations per second), this program illustrates the improved performance of the semi-synchronous version for short vectors.

timesaxpy.c

```
#include <stdio.h>
#include <cube.h>
#include <cveclib.h>

#define N 8192
float sx[N],sy[N],sa,szero;
int nsize[] = {
0,1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192 };

main(){
  int i,j,n,itime;

/* setup dummy constants - this is a timing test */

  sa = 1.0;
  szero = 0.0;
  _sfill( N, &szero, sx, 1 );
  sfill( N, &szero, sy, 1 );

  for(j = 0;j<15;j++){
     n = nsize[ j ];
/* time the synchronous veclib version */
     itime = mclock();
     for(i=0;i<1000;i++)
        saxpy( n, &sa, sx, 1, sy, 1 );
     itime = mclock() - itime;
     printf("  VX milli = %8.4f n = %6d Kflops = %10.5f\n",
           itime/1000.,n,n*2000./(double)itime );
/* time the semi-synchronous version */
     itime = mclock();
     for(i=0;i<1000;i++)
        _saxpy( n, &sa, sx, 1, sy, 1 );
     vpwait();
     itime = mclock() - itime;
     printf(" _VX milli = %8.4f n = %6d Kflops = %10.5f\n",
           itime/1000.,n,n*2000./(double)itime );
  }
}
```

PROGRAM EXAMPLE 4 – dlinpack

This example shows the modified portions of the standard LINPACK benchmark that run on the iPSC/2-VX. (For more information on the LINPACK benchmark, refer to the *LINPACK User's Guide*, 1979, Appendix A, by Bunch, J.R., Dongarra, J.J., Moler, C.B., Stewart, G.W.) Steps to prepare the program include:

1. Insert an INCLUDE statement for the "fcube.h" file.
2. Modify the program to run three cases; n=50, n=100, and n=150.

Additions to the code are on lines between the "c-new" comment lines. Lines are commented out using "c-old".

```
      double precision a(150,150),b(150)
      double precision time(6),vax,ops,total,en,dummy,seconds
      integer ipvt(150)
c-new
      include 'fcube.h'
c-new
c-old lda = 100
      write(*,10)
   10 format(3x,'n',6x,'dgefa',6x,'dgesl',6x,'total',5x,'mflops',
     >   7x,'unit',6x,'ratio')
c-new
      icount = 0
    9 if(icount.eq.0)n=50
      if(icount.eq.1)n=100
      if(icount.eq.2)n=150
      if(icount.eq.3)goto 12
      icount = icount + 1
      lda = n
c-new
      vax = 30.485
      en = n
      ops = (2.0d0*en*en*en)/3.0d0 + 2.0d0*en*en
c
c
         call matgen(a,lda,n,b)
         t1 = seconds(dummy)
         call dgefa(a,lda,n,ipvt,info)
         time(1) = seconds(dummy) - t1
         t1 = seconds(dummy)
         call dgesl(a,lda,n,ipvt,b,0)
         time(2) = seconds(dummy) - t1
         total = time(1) + time(2)
         if (total .eq. 0.0d0) total = 1.0d0
         time(3) = total
         time(4) = ops/(1.0d6*total)
         time(5) = 2.0d0/time(4)
         time(6) = vax/time(5)
```

```
         write(*,11) n,(time(i),i=1,6)
   11    format(i4,3f11.3,f11.6,2f11.3)
         go to 9
c-new
   12 print *,'END.'
c-new
      end
      subroutine matgen(a,lda,n,b)
      double precision a(lda,1),b(1)
c
      init = 1325
      do 30 j = 1,n
         do 20 i = 1,n
            init = mod(3125*init,65536)
            a(i,j) = (init - 32768.0d0)/16384.0d0
   20    continue
   30 continue
      do 35 i = 1,n
          b(i) = 0.0d0
   35 continue
      do 50 j = 1,n
         do 40 i = 1,n
            b(i) = b(i) + a(i,j)
   40    continue
   50 continue
      return
      end
```

PROGRAM EXAMPLE 5 – Makefile

The following sample makefile is a variation of the makefile found in the */usr/ipsc/examples/vx/vx_vec_node* directory. It can be used to create the executable files for the example programs shown in this section, provided all the source files reside in the same directory. It can also be used as a template for your own application. Simply copy the makefile to your directory and then substitute your file names for the ones given in the example makefile.

```
F77FLAGS= -OLM -vx -v
CCFLAGS = -OLM -vx -v
LDFLAGS = -vec -vx -node -v

.SUFFIXES: .c .f .o
.f.o:
        $(F77) $(F77FLAGS) -c $<
.c.o:
        $(CC) $(CCFLAGS) -c $<

all:    timedaxpy timedaxpydbg timesaxpy timesaxpydbg dlinpack
slinpack

timedaxpy:      timedaxpy.o
        $(F77) -o $@ timedaxpy.o -double -vec -vx -node

timedaxpydbg:   timedaxpy.o
        $(F77) -o $@ timedaxpy.o -double -vecdb -vx -node

timesaxpy:      timesaxpy.o
        $(CC) -o $@ timesaxpy.o -single -vec -vx -node

timesaxpydbg:   timesaxpy.o
        $(CC) -o $@ timesaxpy.o -single -vecdb -vx -node

dlinpack:       dmain.o dgefa.o dgesl.o seconds.o
        $(F77) -o $@ dmain.o dgefa.o dgesl.o seconds.o -double
$(LDFLAGS)

slinpack:       smain.o sgefa.o sgesl.o seconds.o
        $(F77) -o $@ smain.o sgefa.o sgesl.o seconds.o -single
$(LDFLAGS)

clean:
        -rm *.o
```

iPSC®/2-VX FORTRAN UTILITY ROUTINES 6

INTRODUCTION

This chapter describes the utility routines used when writing code for the vector processor board. These routines allow you to perform such functions as:

- Access the vector processor board.
- Modify the values of LEDs.
- Determine the contents of status registers.
- Perform fast copies of data between vector and node boards.

These routines are found in the utility library *libvx.a*. The data types for the functions declared in *libvx.a* are found in the include file *vpnode.h*.

Refer to Appendix A of this manual for a summary list of the vector routines found in VecLib. For detailed descriptions of all the routines found in VecLib, refer to the *iPSC®/2 Programmer's Reference Manual*.

NOTE

All undefined bits in the parameters for these routines are reserved and must be set to zero.

Table 6-1 summarizes the iPSC/2-VX utility routines. Following this table is a reference page for each routine, in alphabetical order.

Table 6-1. Summary of iPSC®/2-VX Utility Routines

Routine	Description
M*x*COPY	Fast copy routines for double and single precision data, primarily between node and VX memory.
M*x*GATHR	Fast gather routines for double and single precision vectors, primarily between node and VX memory.
M*x*SCATR	Fast scatter routines for double and single precision vectors, primarily between node and VX memory.
VPEXCEPT	Enables and disables vector processor exceptions.
VPRLEDS	Reads the value of the red and green LEDs on the vector processor board.
VPROUND	Alters the rounding mode.
VPRSTAT	Returns the contents of status register.
VPWAIT	Waits for completion of any asynchronous operations on the vector board.
VPWLEDS	Modifies the red and green LEDs on the vector processor board.

MxCOPY() MxCOPY()

Fast vector copy.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL MDCOPY(N, X, INCX, Y, INCY)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, INCX, INCY

CALL MSCOPY(N, X, INCX, Y, INCY)
```

Description

Fast copy is primarily intended to copy vectors between node and VX memories. This routine is part of the node library, and is linked with the **-node** switch. Parameters are as follows:

N	The number of double or single precision words to transfer.
X	Address of the source vector.
INCX	Increment of the source vector.
Y	Address of the destination vector.
INCY	Increment of the destination vector.

MxGATHR() MxGATHR()

Fast vector gather.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
INTEGER*4 N, IY(*)

CALL MDGATHR(N, X, IY, Z)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, IY

CALL MSGATHR(N, X, IY, Z)
```

Description

Fast vector gather is primarily intended to gather vectors between node and VX memories. This routine is part of the node library, and is linked with the **-node** switch. Parameters are as follows:

N	The number of double or single precision words to transfer.
X	Address of the source vector.
IY	The index, an integer vector of offsets in the source vector. IY must contain values between 1 and N.
Z	Address of the destination vector.

MxSCATR() MxSCATR()

Fast vector scatter.

Calling Sequence

Double precision:

```
DOUBLE PRECISION X(*), Z(*)
INTEGER*4 N, IY(*)

CALL MDSCATR(N, X, IY, Z)
```

Single precision:

```
REAL X(*), Y(*)
INTEGER*4 N, IY

CALL MSSCATR(N, X, IY, Z)
```

Description

Fast vector scatter is primarily intended to scatter vectors between node and VX memories. This routine is part of the node library, and is linked with the **-node** switch. Parameters are as follows:

N	The number of double or single precision words to transfer.
X	Address of the source vector.
IY	The index, an integer vector of offsets in the destination vector. IY must contain values between 1 and N.
Z	Address of the destination vector.

VPEXCEPT() VPEXCEPT()

Enables and disables vector processor exceptions.

Calling Sequence

```
INTEGER  VPEXCEPT,OLDEXCEPT,value$6

OLDEXCEPT = VPEXCEPT(value$6)
```

Input Parameters

value$6 0 in a bit position = disables exception handling
1 in a bit position = enables exception handling

mask (hex)	exception	operation	bit position
1	NAN (not a number)	*	0
2	Overflow	*	1
4	Underflow	*	2
8	NAN (not a number)	+	3
10	Overflow	+	4
20	Underflow	+	5

Return Values

OLDEXCEPT Returns the previous exception mask value as the low 6 bits of its 32-bit integer return value.

Description

The exception condition is checked at the end of each routine when you use the **-vecdb** switch. The process is aborted when an enabled exception occurs, with the name of the exception and the routine name given as part of the error message.

VPRLEDS() VPRLEDS()

Reads and returns the value of the red and green LEDs on the vector processor board.

Calling Sequence

```
INTEGER VALUE, VPRLEDS

VALUE = VPRLEDS()
```

Input Parameters

None

Return Values

VALUE Returns a 32-bit value where the low two bits are significant as described below.

Description

The low two bits of the returned 32-bit value correspond to the setting of the green LED (bit 0) and red LED (bit 1) A "1" in the bit position means ON, a "0" means OFF. Refer to Figure 6-1.

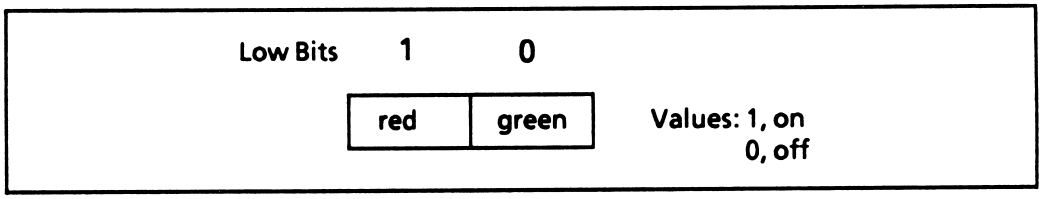


Figure 6-1. Return Value of 32 Bits

In decimal, the following numbers indicate the status of the LEDs.

0 both off
1 green on, red off
2 green off, red on
3 both on

Example

```
VALUE = VPLEDS()
```

VPROUND() VPROUND()

Alters the rounding mode. The default is to nearest number.

Calling Sequence

```
INTEGER OLDVALUE, VPROUND, value$4

OLDVALUE = VPROUND(value$4)
```

Input Parameters

value$4 Bits 0 and 1 control the adder rounding mode and bits 2 and 3 control the multiplier.

00	Set rounding to nearest number.
01	Set rounding to plus infinity.
10	Set rounding to zero.
11	Set rounding to minus infinity.

Return Values

OLDVALUE Returns the previous rounding mode setting as the low four bits of its 32-bit integer return value.

Description

The **vpround** function sets the rounding mode according to the input parameter. For example:

```
c  set rounding to zero for both adder and multiplier

   oldvalue = vpround(10)
```

This would result in setting the rounding modes of the adder and multiplier to zero. The **vpround** function returns the value *oldvalue*, the previous value. The decimal digit 10 is represented in binary as 1010, showing the set rounding to zero for both adder and multiplier.

VPROUND() *(cont.)*

VPROUND() *(cont.)*

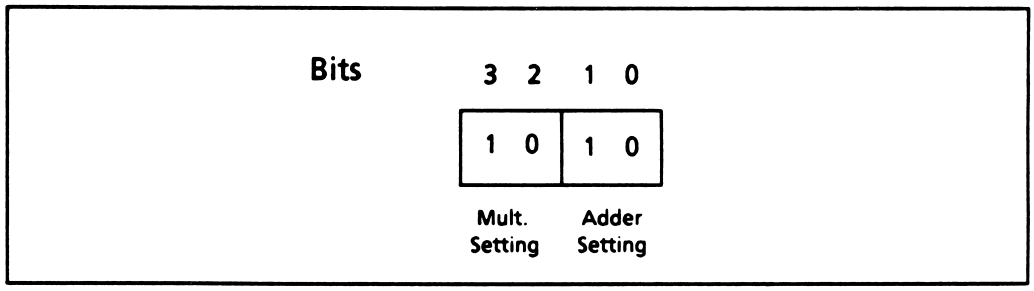


Figure 6-2. Setting Values on a Function

You can later reset the rounding mode to the previous mode with the statement:

```
oldvalue = vpround(oldvalue)
```

VPRSTAT() VPRSTAT()

Returns the contents of status register.

Calling Sequence

```
INTEGER  STATUS,VPRSTAT

STATUS = VPRSTAT()
```

Input Parameters

None

Return Values

STATUS Returns the contents of the status register(s), in 32-bit integer value.

Description

The status register (STSREG) indicates the state of the vector processor board by providing information on any arithmetic exceptions, pending interrupt requests, or memory parity errors.

Each of the 32 bits are described below starting with the 16, low-order bits. The 16 low-order bits indicate the current status of the vector processor board.

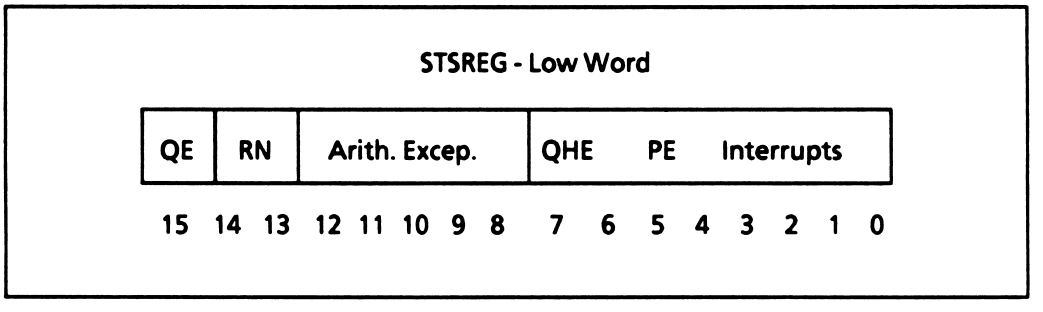


Figure 6-3. Low-order Bit Values

VPRSTAT() *(cont.)*

VPRSTAT() *(cont.)*

BIT 15:	QUEUE EMPTY	The command queue is empty and the vector processor board is idle.
BIT 14:	RUN	The vector processor board is in run state.
BIT 13:	ALU UNDERFLOW	The last operation or series of operations produced at least one underflow condition in the floating point ALU.
BIT 12:	ALU OVERFLOW	The last operation (or series of operations) produced at least one overflow condition in the floating point ALU.
BIT 11:	ALU INVALID OPERATION	The last operation or series of operations produced at least one invalid operation in the floating point ALU (for example, NAN + NORM)
BIT 10:	MULTIPLIER UNDERFLOW	The last operation or series of operations in the floating point MULT produced at least one underflow result and was forced to ZERO.
BIT 9:	MULTIPLIER OVERFLOW	The last operation or series of operations produced at least one overflow condition in the floating point MULT.
BIT 8:	MULTIPLIER INVALID OPERATION	Indicates the last operation or series of operations produced at least one invalid operation in the floating point MULT (for example, NAN x NORM)
BIT 7:		Command queue is half empty
BIT 6:		Reserved
BIT 5:	PARITY ERROR	A data parity error was encountered while accessing the dynamic RAM portion of local memory either during an arithmetic process or host CPU access.
BIT 4:	INT REQ	An arithmetic exception has occurred
BIT 3:	INT REQ	Command queue almost full
BIT 2:	INT REQ	Command queue half-empty
BIT 1:	INT REQ	Illegal microcode error interrupt
BIT 0:	INT REQ	Programmed interrupt

VPRSTAT() *(cont.)*

VPRSTAT() *(cont.)*

The 16, high-order bits indicate the setting of optional modes as shown in Figure 6-4:

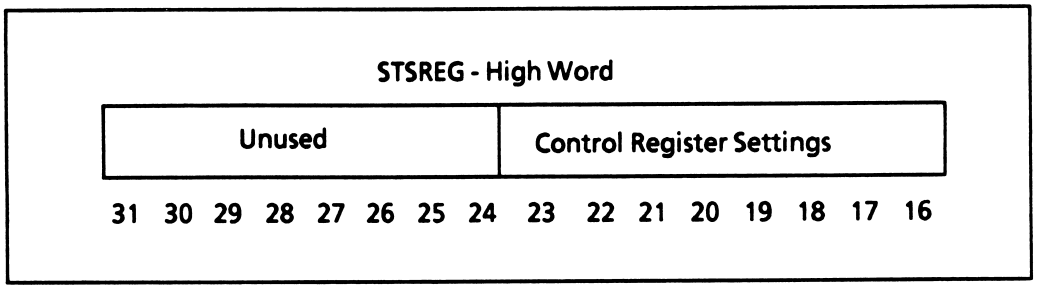


Figure 6-4. High-order Bit Values

BIT 24-31: Unused

BIT 23: Red LED (0=off, 1=on)

BIT 22: Green LED (0 = off, 1 = on)

BIT 21,20: ALU rounding mode
- 0, 0 nearest
- 0, 1 zero
- 1, 0 + Infinity
- 1, 1 - Infinity

BIT 19, 18: MULTIPLIER rounding mode
- 0, 0 nearest
- 0, 1 zero
- 1, 0 + Infinity
- 1, 1 - Infinity

BIT 17: Arithmetic exception mask

BIT 16: Queue half-empty interrupt mask

VPWAIT() VPWAIT()

Causes process to wait for completion of asynchronous operations.

Calling Sequence

```
CALL VPWAIT
```

Return Values

None

Description

Causes process to wait for the completion of asynchronous subroutines. These subroutines have the same names and functionality as VecLib subroutines, except the names are preceded by an underscore (_) The asynchronous routines queue the vector operation on the vector board and return to the caller before the vector operation is complete. If you are calling the asynchronous routines explicitly (without using VAST-2), you must use calls to **vpwait()** to synchronize the VX board with the 386 microprocessor.

VPWLEDS() VPWLEDS()

Writes the value to modify the red and green LEDs on the vector processor board.

Calling Sequence

```
INTEGER value$2

CALL VPWLEDS(value$2)
```

Input Parameters

`value$2` Low two bits of the integer value are interpreted as requests to modify the green LED (bit 0) and the red LED (bit 1). A "1" in the bit means ON, a "0"means OFF. (Refer to Figure 6-5) The low two bits of the integer value are written to the control register.

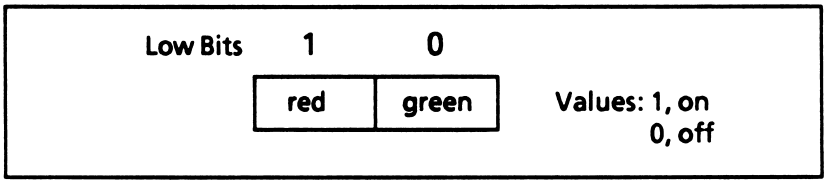


Figure 6-5. Control Register Bit Values

Return Values

None

Description

To turn on the green LED and turn off the red:

```
CALL VPWLEDS(1)
```

To turn off the red LED leaving the green alone:

```
CALL VPWLEDS (IAND(1,VPRLEDS()))
```

In decimal, the following numbers give the following results:

0 both off
1 green on, red off
2 green off, red on
3 both on

iPSC®/2-VX C UTILITY ROUTINES 7

INTRODUCTION

This chapter describes the utility routines used when writing code for the vector processor board. These routines allow you to perform such functions as:

- Access the vector processor board.
- Modify the values of LEDs.
- Determine the contents of status registers.
- Perform dynamic allocation of vector memory.

These routines are found in the utility library *libvx.a*.

Refer to Appendix B of this manual for a summary list of the vector routines found in VecLib. For detailed descriptions of all the routines found in VecLib, refer to the *iPSC®/2 Programmer's Reference Manual*.

NOTE

All undefined bits in the parameters for these routines are reserved and must be set to zero.

Table 7-1 summarizes the iPSC/2-VX utility routines. Following this table is a reference page for each routine, in alphabetical order.

Table 7-1. Summary of iPSC®/2-VX Utility Routines

Routine	Description
vpexcept	Enables and disables vector processor exceptions.
vprleds	Reads the value of the red and green LEDs on the vector processor board.
vpround	Alters the rounding mode.
vprstat	Returns the contents of status register.
vpwait	Waits for completion of any asynchronous operations on the vector board.
vpwleds	Modifies the red and green LEDs on the vector processor board.
vx_brk vx_sbrk	Change data segment space allocation.
vx_malloc vx_free vx_realloc vx_calloc vx_cfree	VX memory allocation calls.

VPEXCEPT() VPEXCEPT()

Enables and disables vector processor exceptions.

Calling Sequence

```
integer  vpexcept, oldexcept, value_6;

oldexcept = vpexcept(value_6);
```

Input Parameters

value$1 in a bit position = disables exception handling
1 in a bit position = enables exception handling

mask (hex)	exception	operation	bit position
1	NAN (not a number)	*	0
2	Overflow	*	1
4	Underflow	*	2
8	NAN (not a number)	+	3
10	Overflow	+	4
20	Underflow	+	5

Return Values

oldexcept Returns the previous exception mask value as the low bit of its 32-bit integer return value.

Description

The exception condition is checked at the end of each routine when you use the **-vecdb** switch. The process is aborted when an enabled exception occurs, with the name of the exception and the routine name given as part of the error message.

VPRLEDS() VPRLEDS()

Reads and returns the value of the red and green LEDs on the vector processor board.

Calling Sequence

```
int value, vprleds;

value = vprleds();
```

Input Parameters

None

Return Values

value	Returns a 32-bit value where the low two bits are significant as described below.

Description

The low two bits of the returned 32-bit value correspond to the setting of the green LED (bit 0) and red LED (bit 1). A "1" in the bit position means ON, a "0" means OFF. Refer to Figure 7-1.

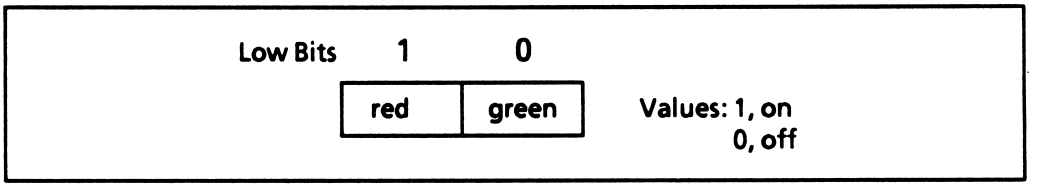


Figure 7-1. Return Value of 32 Bits

In decimal, the following numbers indicate the status of the LEDs.

0 both off
1 green on, red off
2 green off, red on
3 both on

Example

```
value = vpleds()
```

VPROUND() VPROUND()

Alters the rounding mode. The default is to nearest number.

Calling Sequence

```
int oldvalue, vpround, value_4;

oldvalue = vpround(value_4);
```

Input Parameters

value$4 Bits 0 and 1 control the adder rounding mode and bits 2 and 3 control the multiplier.

00	Set rounding to nearest number.
01	Set rounding to plus infinity.
10	Set rounding to zero.
11	Set rounding to minus infinity.

Return Values

oldvalue Returns the previous rounding mode setting as the low four bits of its 32-bit integer return value.

Description

The vpround function sets the rounding mode according to the input parameter. For example:

```
/* set rounding to zero for both adder and multiplier */

oldvalue = vpround(10)
```

This would result in setting the rounding modes of the adder and multiplier to zero. The **vpround** function returns the value *oldvalue*, the previous value. The decimal digit 10 is represented in binary as 1010, showing the set rounding to zero for both adder and multiplier.

VPROUND() *(cont.)*

VPROUND() *(cont.)*

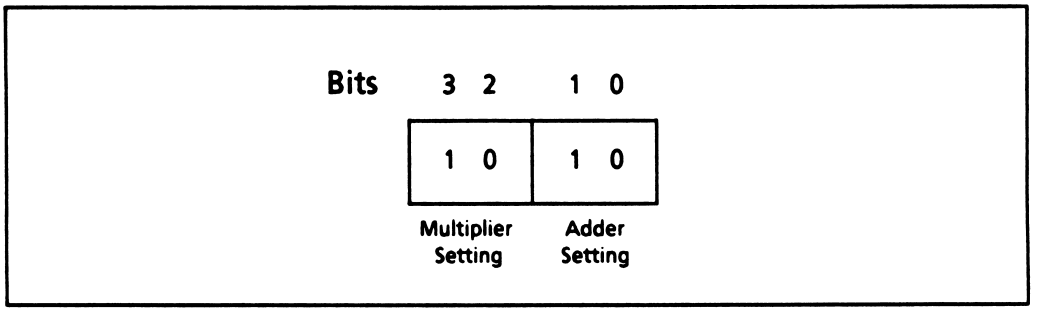


Figure 7-2. Setting Values on a Function

You can later reset the rounding mode to the previous mode with the statement:

```
iold = vpround(iold);
```

VPRSTAT() VPRSTAT()

Returns the contents of status register.

Calling Sequence

```
int  status,vprstat;

status = vprstat();
```

Input Parameters

None

Return Values

status Returns the contents of the status register(s), in 32-bit integer value.

Description

The status register (STSREG) indicates the state of the vector processor board by providing information on any arithmetic exceptions, pending interrupt requests, or memory parity errors.

Each of the 32 bits are described below starting with the 16 low-order bits. The 16 low-order bits indicate the current status of the vector processor board.

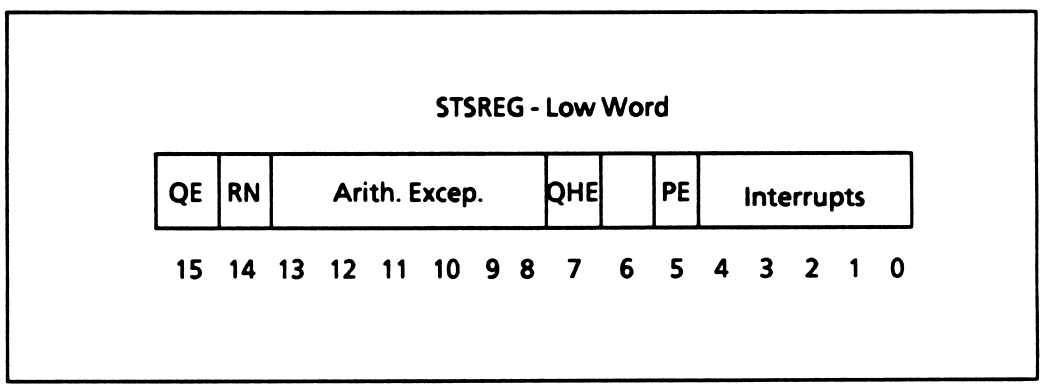


Figure 7-3. Low-order Bit Values

VPRSTAT() *(cont.)*

VPRSTAT() *(cont.)*

BIT 15:	QUEUE EMPTY	The command queue is empty and the vector processor board is idle.
BIT 14:	RUN	The vector processor board is in run state.
BIT 13:	ALU UNDERFLOW	The last operation or series of operations produced at least one underflow condition in the floating point ALU.
BIT 12:	ALU OVERFLOW	The last operation (or series of operations) produced at least one overflow condition in the floating point ALU.
BIT 11:	ALU INVALID OPERATION	The last operation or series of operations produced at least one invalid operation in the floating point ALU (for example, NAN + NORM).
BIT 10:	MULTIPLIER UNDERFLOW	The last operation or series of operations in the floating point MULT produced at least one underflow result and was forced to ZERO.
BIT 9:	MULTIPLIER OVERFLOW	The last operation or series of operations produced at least one overflow condition in the floating point MULT.
BIT 8:	MULTIPLIER INVALID OPERATION	Indicates the last operation or series of operations produced at least one invalid operation in the floating point MULT (for example, NAN * NORM).
BIT 7:		Command queue is half empty.
BIT 6:		Reserved
BIT 5:	PARITY ERROR	A data parity error was encountered while accessing the dynamic RAM portion of local memory either during an arithmetic process or host CPU access.
BIT 4:	INT REQ	An arithmetic exception has occurred.
BIT 3:	INT REQ	Command queue almost full.
BIT 2:	INT REQ	Command queue half-empty.
BIT 1:	INT REQ	Illegal microcode error interrupt.
BIT 0:	INT REQ	Programmed interrupt.

VPRSTAT() *(cont.)*

VPRSTAT() *(cont.)*

The 16 high-order bits indicate the setting of optional modes as shown in Figure 7-4:

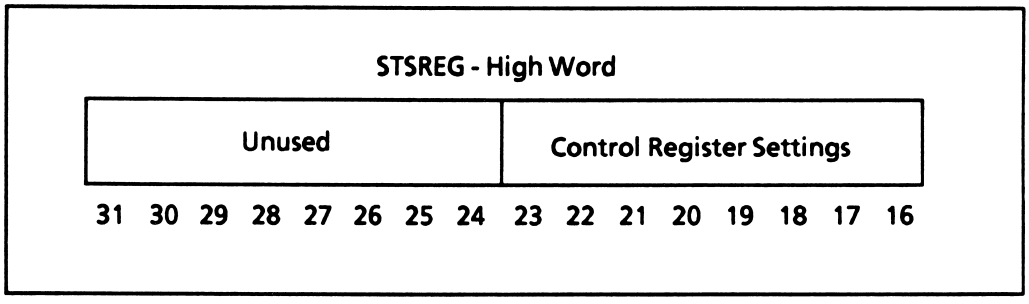


Figure 7-4. High-order Bit Values

BIT 24-31: Unused

BIT 23: Red LED (0=off, 1=on)

BIT 22: Green LED (0=off, 1=on)

BIT 21,20: ALU rounding mode
- 0, 0 nearest
- 0, 1 zero
- 1, 0 + Infinity
- 1, 1 - Infinity

BIT 19, 18: MULTIPLIER rounding mode
- 0, 0 nearest
- 0, 1 zero
- 1, 0 + Infinity
- 1, 1 - Infinity

BIT 17: Arithmetic exception mask

BIT 16: Queue half-empty interrupt mask

VPWAIT() VPWAIT()

Causes process to wait for completion of asynchronous operations.

Calling Sequence

```
vpwait()
```

Return Values

None

Description

Causes process to wait for the completion of asynchronous subroutines. These subroutines have the same names and functionality as VecLib subroutines, except the names are preceded by an underscore (__). The asynchronous routines queue the vector operation on the vector board and return to the caller before the vector operation is complete. You must use calls to **vpwait()** to synchronize the VX board with the 386 microprocessor.

VPWLEDS() VPWLEDS()

Writes the value to modify the red and green LEDs on the vector processor board.

Calling Sequence

```
int value_2;

vpwleds(value_2);
```

Input Parameters

value_2 Low two bits of the integer value are interpreted as requests to modify the green LED (bit 0) and the red LED (bit 1). A "1" in the bit means ON, a "0" means OFF. (Refer to Figure 7-5). The low two bits of the integer value are written to the control register.

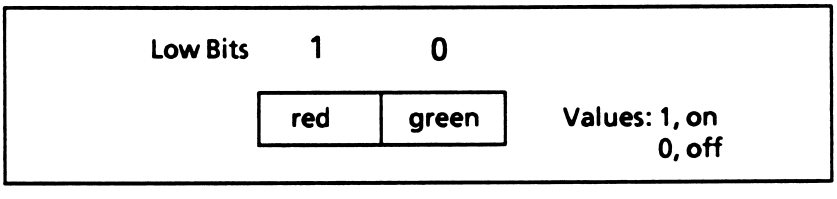


Figure 7-5. Control Register Bit Values

Return Values

None

Description

To turn on the green LED and turn off the red:

```
vpwleds(1)
```

To turn off the red LED leaving the green alone:

```
vpwleds (vprleds() & 1);
```

In decimal, the following numbers give the following results:

0 both off
1 green on, red off
2 green off, red on
3 both on

VX__BRK(), VX__SBRK() VX__BRK(), VX__SBRK

Change data segment space allocation.

Calling Sequence

```
int vx_brk(endds)
char *endd;

char *vx_sbrk(incr)
int incr;
```

Return Values

Upon successful completion, **vx__brk** returns a value of 0; **vx__sbrk** returns the old break value. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Description

NOTE

These are low-level routines, used primarily to support the other calls. They are not recommended for general use. The only available extra memory is that not already statically allocated to the program.

The **vx__brk** and **vx__sbrk** calls are used to change dynamically the amount of space allocated to the calling process's data segment. They make this change by resetting the break value of the process and allocating the appropriate memory space. The break value is the address of the first location beyond the end of the data segment. As the break value increases, the amount of allocated space also increases. All newly allocated space is assigned values of zero. If the same memory space is reallocated to the same process, the values are undefined.

For **vx__brk**, the parameter *endds* is a pointer to the address that is to be the new end of the data segment; it sets the break value to that address and changes the allocated space accordingly.

For **vx__brk**, the parameter *incr* is an integer that is the number of bytes that you want to add to the current data segment; it adds the specified number of bytes to the break value and changes the allocated space accordingly. If *incr* is negative, the amount of allocated space is decreased.

VX_MALLOC, VX_FREE, VX_REALLOC, VX_CALLOC, VX_CFREE

VX memory allocation calls

Calling Sequence

```
char *vx_malloc(size)
unsigned size

void vx_free(ptr)
char *ptr

char *vx_realloc(ptr, size)
char *ptr;
unsigned size;

char *vx_calloc(nelem, elsize)
unsigned nelem, elsize;

void vx_cfree(ptr, nelem, elsize)
char *ptr;
unsigned nelem, elsize);
```

Return Values

The calls **vx_malloc**, **vx_realloc**, and **vx_realloc** return a pointer to a space suitably aligned (after possible pointer coersion) for storage of any type of object. If there is no available memory, the allocation routines return a NULL pointer.

The calls **vx_free** and **vx_cfree** do not return a value.

Description

These calls provide a way to allocate VX memory dynamically.

The **vx_malloc** function returns a pointer to a block that is at least *size* bytes, suitably aligned for any use. It allocates the first contiguous block of space that is big enough found in a circular search from the last block allocated or freed. As it searches, it coalesces adjacent free blocks. It calls **vx_sbrk** to get more memory from the system when no suitable space is already free.

The **vx_free** function function frees for further allocation a block previously allocated by **vx_malloc**. The argument is a pointer to the block previously allocated. If the space assigned by **vx_malloc** is overrun, or a random number is used as the argument, undefined results will occur.

VX_MALLOC, VX_FREE, VX_REALLOC, VX_CALLOC, VX_CFREE *(cont.)*

The **vx_realloc** function changes the size of the block that the argument *ptr* points to, and returns a pointer to the block (which may have been moved). The contents of the block will be unchanged, up to the lesser of the new or old sizes. If no free block of *size* bytes is available in the storage area, **vx_realloc** asks **vx_malloc** to enlarge the area by *size* bytes, and then moves the data to the new space. The **vx_realloc** call also works if *ptr* points to a block freed since the last call of **vx_malloc**, **vx_realloc**, or **vx_calloc**. This allow you to use sequences of **vx_free**, **vx_malloc**, and **vx_realloc** to exploit the search strategy of **vx_malloc** for storage compaction.

The **vx_calloc** function allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeroes.

The **vx_cfree** function frees for further allocation a block previously allocated by **vx_calloc**. The arguments are a pointer (*ptr*) to that block, the number of elements (*nelem*), and the size of the elements (*elsize*).

FORTRAN VECLIB ROUTINE SUMMARY A

This appendix contains a set of tables that summarize the Fortran VecLib routines according to function. There are eight tables that categorize the routines as follows:

- Mathematical Primitives
- Other Mathematical Functions
- Triad Operations
- Relational Primitive Operations
- Logical Primitive Operations
- Reduction Functions
- Conversion Primitives
- Miscellaneous Operations

Each table describes the functions and then lists the Fortran versions of the routines. The *iPSC®/2 Programmer's Reference Manual* lists these routines alphabetically.

The routines are listed by root name. An italic *x* represents where the data type is specified in the routine name. The data type upon which a function operates is indicated by the first or second letter of the routine name; "d" for double, "s" for single, "i" for integer, "l" for logical, and "c" for complex, and "z" for double precision complex. For example, **dasum** is the double precision routine to calculate the sum of absolute values, while **sasum** is the single precision version.

The first three columns contain the routine's root name, data types and a brief description. The fourth column gives the calling sequence and equivalent in the language specified. A "d" is used in the examples to represent the double precision version of the routine. When "i" is used as an index in the Fortran equivalent statement, "i" ranges from 1 to n.

Table A-1. Mathematical Primitives

Root Name	Data Type	Description	Calling Sequence/Fortran Equivalent
*x*swap	d,s,i,c,z	Swap vectors	`call dswap(n,x,incx,y,incy)` `t = y(i)  y(i) = x(i)  x(i) = t`
*x*copy	d,s,i,l,c,z	Copy vector	`call dcopy(n,x,incx,y,incy)` `y(i) = x(i)`
*x*fill	d,s,i,l,c,z	Fill vector	`call dfill(n,alpha,x,incx)` `x(i) = alpha`
*x*neg	d,s,i,c,z	Change sign	`call dneg(n,x,incx)` `x(i) = -x(i)`
*x*vneg	d,s,i,c,z	Negate vector	`call dvneg(n,x,incx,y,incy)` `y(i) = -x(i)`
*x*sadd	d,s,i,c,z	Scalar plus vector	`call dsadd(n,alpha,x,incx,y,incy)` `y(i) = alpha + x(i)`
*x*vadd	d,s,i,c,z	Vector addition	`call dvadd(n,x,incx,y,incy,z,incz)` `z(i) = x(i) + y(i)`
*x*ssub	d,s,i,c,z	Scalar vector subtraction	`call dssub(n,alpha,x,incx,y,incy)` `y(i) = alpha - x(i)`
*x*vsub	d,s,i,c,z	Vector subtraction	`call dvsub(n,x,incx,y,incy,z,incz)` `z(i) = x(i) - y(i)`
*x*smul	d,s,i,c,z	Scalar times vector	`call dsmul(n,alpha,x,incx,y,incy)` `y(i) = alpha * x(i)`
*x*vmul	d,s,i,c,z	Vector element-wise multiplication	`call dvmul(n,x,incx,y,incy,z,incz)` `z(i) = x(i) * y(i)`
*x*scal	d,s,c,z	Scalar times a vector to itself	`call dscal(n,alpha,x,incx)` `x(i) = alpha * x(i)`
*x*sdiv	d,s,i,c,z	Scalar divided by vector	`call dsdiv(n,alpha,x,incx,y,incy)` `y(i) = alpha / x(i)`
*x*vrecp	d,s,c,z	Vector reciprocal	`call dvrecp(n,x,incx,y,incy)` `y(i) = 1.0 / x(i)`
*x*vdiv	d,s,i,c,z	Element-wise vector division	`call dvdiv(n,x,incx,y,incy,z,incz)` `z(i) = x(i) / y(i)`

Table A-2. Other Mathematical Functions

Root Name	Data Type	Description	Calling Sequence/Fortran Equivalent
*x*vabs	d,s,i,c,z	Element-wise absolute value	`call dvabs(n,x,incx,y,incy)` `y(i) = abs(x(i))`
*x*vmax	d,s	Vector element-wise maximum	`call dvmax(n,x,incx,y,incy,z,incz)` `z(i) = MAX(x(i),y(i))`
*x*vmin	d,s	Vector element-wise minimum	`call dvmin(n,x,incx,y,incy,z,incz)` `z(i) = MIN(x(i),y(i))`
*x*vamax	d,s	Vector element-wise maximum absolute value	`call dvamax(n,x,incx,y,incy,z,incz)` `z(i) = MAX(abs(x(i)),abs(y(i)))`
*x*vamin	d,s	Vector element-wise minimum absolute value	`call dvamin(n,x,incx,y,incy,z,incz)` `z(i) = MIN(abs(x(i)),abs(y(i)))`
*x*vpow	d,s	Element-wise power function	`call dvpow(n,x,incx,y,incy,z,incz)` `z(i) = x(i) ** y(i)`
*x*vexp	d,s	Element-wise exponential	`call dvexp(n,x,incx,y,incy)` `y(i) = exp(x(i))`
*x*vlg10	d,s	Element-wise base 10 logarithm.	`call dvlg10(n,x,incx,y,incy)` `y(i) = log10(x(i))`
*x*vlog	d,s	Element-wise natural logarithm	`call dvlog(n,x,incx,y,incy)` `y(i) = log(x(i))`
*x*vatan	d,s	Element-wise inverse-tangent	`call dvatan(n,x,incx,y,incy)` `y(i) = atan(x(i))`
*x*vatn2	d,s	Element-wise inverse-tangent of quotient	`call dvatn2(n,x,incx,y,incy,z,incz)` `z(i) = atan2(x(i),y(i))`
*x*vcos	d,s	Element-wise cosine	`call dvcos(n,x,incx,y,incy)` `y(i) = cos(x(i))`
*x*vsin	d,s	Element-wise sine	`call dvsin(n,x,incx,y,incy)` `y(i) = sin(x(i))`
*x*vsqrt	d,s	Element-wise square root	`call dvsqrt(n,x,incx,y,incy)` `y(i) = sqrt(x(i))`
*x*random	d,s	Generates pseudo-random number	`d = drandom()`
*x*vrandom	d,s	Pseudo-random vector generation	`call dvrandom(n,x,incx)`

Table A-3. Triad Operations

Base Name	Data Type	Description	Calling Sequence/Fortran Equivalent
*x*axpy	d,s,c,z	Scalar times vector plus a vector to itself	`call daxpy(n,alpha,x,incx,y,incy)` `y(i) = alpha*x(i) + y(i)`
*x*svmvt	d,s	Scalar minus vector quantity times vector	`call dsvmvt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = (alpha - x(i)) * y(i)`
*x*svpvt	d,s	Scalar plus vector quantity times vector	`call dsvpvt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = (alpha + x(i)) * y(i)`
*x*svtsp	d,s	Scalar times vector quantity plus scalar	`call dsvtsp(n,alpha,beta,x,incx,y,incy)` `y(i) = alpha * x(i) + beta`
*x*svtvm	d,s	Scalar times vector quantity minus vector	`call dsvtvm(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * x(i) - y(i)`
*x*svtvp	d,s	Scalar times quantity of vector plus vector	`call dsvtvp(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * x(i) + y(i)`
*x*svvmt	d,s	Scalar times quantity of vector minus vector	`call dsvvmt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * (x(i) - y(i))`
*x*svvpt	d,s	Scalar times quantity of vector plus vector	`call dsvvpt(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha * (x(i) + y(i))`
*x*svvtm	d,s	Scalar minus quantity of vector times vector	`call dsvvtm(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha - x(i) * y(i)`
*x*svvtp	d,s	Scalar plus quantity of vector times vector	`call dsvvtp(n,alpha,x,incx,y,incy,z,incz)` `z(i) = alpha + x(i) * y(i)`
*x*vvmvt	d,s	Vector minus vector quantity times vector	`call dvvmvt(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = (w(i) - x(i)) * y(i)`
*x*vvpvt	d,s	Vector plus vector quantity times vector	`call dvvpvt(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = (w(i) + x(i)) * y(i)`
*x*vvtvm	d,s	Vector times vector quantity minus vector	`call dvvtvm(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = w(i) * x(i) - y(i)`
*x*vvtvp	d,s	Vector times vector quantity plus vector	`call dvvtvp(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = w(i) * x(i) + y(i)`
*x*vvvtm	d,s	Vector minus quantity of vector times vector	`call dvvvtm(n,w,incw,x,incx,y,incy,z,incz)` `z(i) = w(i) - x(i) * y(i)`

Table A-4. Relational Primitive Operations

Root Name	Data Type	Description	Calling Sequence/Fortran Equivalent
*x*eq	d,s,i	Vector element equality	`call deq(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .eq. y(i)`
*x*seq	d,s,i	Vector equal to scalar	`call dseq(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .eq. x(i)`
*x*ge	d,s,i	Vector element greater than or equal	`call dge(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .ge. y(i)`
*x*sge	d,s,i	Scalar greater than or equal to vector	`call dsge(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .ge. x(i)`
*x*gt	d,s,i	Vector element greater than	`call dgt(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .gt. y(i)`
*x*sgt	d,s,i	Scalar greater than vector	`call dsgt(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .gt. x(i)`
*x*sle	d,s,i	Scalar less than or equal to vector	`call dsle(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .le. x(i)`
*x*slt	d,s,i	Scalar less than vector	`call dslt(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .lt. x(i)`
*x*ne	d,s,i	Vector element inequality	`call dne(n,x,incx,y,incy,lz,incz)` `lz(i) = x(i) .ne. y(i)`
*x*sne	d,s,i	Vector not equal to scalar	`call dsne(n,alpha,x,incx,ly,incy)` `ly(i) = alpha .ne. x(i)`

Table A-5. Logical Primitive Operations

Root Name	Data Type	Description	Calling Sequence/Fortran Equivalent
land	l	Vector logical AND with vector	`call land(n,x,incx,y,incy,z,incz)` `z(i) = x(i) .and. y(i)`
lnot	l	Vector logical negation	`call lnot(n,x,incx,y,incy)` `y(i) = .not. x(i)`
lor	l	Vector logical OR with vector	`call lor(n,x,incx,y,incy,z,incz)` `z(i) = x(i) .or. y(i)`
lsand	l	Scalar logical AND with scalar	`call lsand(n,alpha,x,incx,y,incy)` `y(i) = alpha .and. x(i)`
lsor	l	Scalar logical OR with vector	`call lsor(n,alpha,x,incx,y,incy)` `y(i) = alpha .or. x(i)`
lcopy	d,s,i,l,c,z	Copy vector	`call lcopy(n,x,incx,y,incy)` `y(i) = x(i)`
lfill	d,s,i,l,c,z	Fill vector	`call lfill(n,alpha,x,incx)` `x(i) = alpha`

Table A-6. Reduction Functions

Root Name	Data Type	Description	Calling Sequence/Fortran Equivalent
*x*asum	d,s	Sum of absolute values	`d = dasum(n,x,incx)` `dasum = dasum + abs(x(i))`

dzasum	d	Sum of absolute value of real and imaginary parts of the values	`d= dzasum(n,x,incx)`
			`dzasum = dzasum + abs(real(x(i))) +`
			`abs(imag(x(i)))`

scasum	s	Sum of absolute value of real and imaginary parts of the vector	`s = scasum(n,x,incx)`
			`scasum = scasum + abs(real(x(i))) +`
			`abs(imag(x(i)))`

*x*sum	d,s,c,z	Vector sum	`s = dsum(n,x,incx)` `dsum = dsum + x(i)`
*x*nrm2	d,s	Euclidean vector norm	`d = dnrm2(n,x,incx)` `d = sqrt(ddot(n,x,incx,x,incx))`
*x*dot	d,s	Dot product of two vectors	`d = ddot (n,x,incx,y,incy)` `ddot = ddot + x(i)*y(i)`
*x*dotc	c,z	Dot product of two complex vectors	`c = cdotc (n,x,incx,y,incy)` `cdotc = cdotc + CONJG(x(i))*y(i)`
*x*dotu	c,z	Dot product of two complex vectors	`c = cdotu (n,x,incx,y,incy)` `cdotu = cdotu + x(i)*y(i)`

i*x*amax	d,s	Index of maximum absolute value	`i = idamax(n,x,incx)`
			`idamax = MIN (1, max(0,n))`
			`if(abs(x(i)).gt.abs(x(idamax)))idamax=i`

i*x*amin	d,s	Index of minimum absolute value	`i = idamin(n,x,incx)`
			`idamin = MIN(1, max(0,n))`
			`if(abs(x(i)).lt.abs(x(idamin)))idamin=i`

i*x*max	d,s	Index of maximum value	`i = idmax(n,x,incx)`
			`idmax = min (1, max(0,n))`
			`if (x(i) .gt. x(idmax)) idmax = i`

i*x*min	d,s	Index of minimum value	`i = idmin(n,x,incx)`
			`idmin = min (1, max(0,n))`
			`if (x(i) .lt. x(idmin)) idmin = i`

icount	l	Number of logical true values	`i = icount(n,lx,incx)` `if (lx(i)) icount = icount + 1`
ifirst	i	Index of first logical true value	`i = ifirst(n,lx,incx)` `if (lx(i)) ifirst = i`
ilast	i	Index of last logical true value	`i = ilast(n,lx,incx)` `if (lx(i) .and. ifirst .eq. 1)ilast = i`

lany	l	Logical true if any of x are true	`l = lany(n,x,incx)`
			`lany = .false.`
			`if (x(i)) lany = .true.`

Table A-7. Conversion Primitives

Root Name	Data Type	Description	Calling Sequence/Fortran Equivalent
*x*vcmplx	c,z	real to complex	`call cvcmplx(n,sx,incx,sy,incy,z,incz)` `z(i) = CMPLX(sx(i),sy(i))`
*x*vconjg	c,z	Conjugate of a complex vector	`call cvconjg(n,x,incx,y,incy)` `y(i) = CONJG(x(i))`
vdble	d	Single to double precision	`call vdble(n,sx,incx,dy,incy)` `dy(i) = dble(sx(i))`
*x*vfix	d,s	Truncate elements to integer values	`call dvfix(n,x,incx,iy,incy)` `iy(i) = int(x(i))`
*x*vfloa	d,s	Convert integer to floating point	`call dvfloa(n,ix,incx,y,incy)` `y(i) = ix(i)`
*x*vimag	c,z	Extract imaginary part of complex vector	`call cvimag(n,x,incx,sy,incy)` `sy(i) = imag(x(i))`
*x*vreal	c,z	Extract real part of complex vector	`call cvreal(n,x,incx,sy,incy)` `sy(i)=real(x(i))`
vsngl	s	Double to single precision	`call vsngl(n,dx,incx,sy,incy)` `sy(i) = sngl(dx(i))`

Table A-8. Miscellaneous Functions

Root Name	Data Type	Description	Calling Sequence/Fortran Equivalent
*x*scatr	d,s,i	Vector scatter	call dscatr(n,x,incx,iy,incy,z,incz) z(iy(i)) = x(i)
*x*gathr	d,s,i	Vector gather	call dgathr(n,x,incx,iy,incy,z,incz) z(i) = x(iy(i))
*x*ramp	d,s,i	Ramp function	call dramp(n,alpha,beta,x,incx) x(i) = alpha + (i-1)*beta
*x*clip	d,s	Clip to interval [alpha,beta]	call dclip(n,x,incx,alpha,beta,y,incy) y(i) =MAX(MAX(x(i),alpha),beta)
*x*iclip	d,s	Inverted clip	call diclip(n,x,incx,alpha,beta,y,incy) if (x(i).lt.(alpha + beta)/2.0) then y(i) = MIN (x(i),alpha) else y(i) = MAX(x(i),beta)
*x*cndst	d,s	Conditional assignment	call dcndst(n,x,incx,ly,incy,z,incz) if (ly(i)) z(i) = x(i)
*x*mask	d,s	Conditional assignment	call dmask(n,w,incw,x,incx,ly,incy,z,incz) if (ly(i)) z(i) = w(i) else z(i) = x(i)
*x*fft	c,z	Fast Fourier Transform	call cfft(n,x,incx,y,incy) y = fft(x)
*x*ifft	c,z	Inverse Fast Fourier Transform	y = cifft(n,x,incx,y,incy) y = fft^{-1}(x)
*x*rot	d,s	Apply a plane rotation (t = x(i))	call drot(n,x,incx,y,incy,c,s) x(i)= t * c + y(i) * s y(i)= -t * s + y(i) * c
*x*rotg	d,s	Construct a Givens plane rotation	call drotg(a, b, c, s)
*x*folr	d,s	First order linear recurrence routine	call dfolr(n,x,incx,y,incy,z,incz) z(i+1)=y(i)+z(i)*x(i)
*x*solr	d,s	Second order recurrence routine	call dsolr(n,w,incw,x,incx,y,incy,z,incz) z(i+2)=w(i)+z(i+1)*x(i)+z(i)*y(i)
*x*lbidi	d,s	Solves lower bidiagonal	call dlbidi(n,l,incl,b,incb,x,incx) x(i)=b(i)-l(i)*x(i-1)
*x*trfac	d,s	LU factorization of matrix tri-diagonal	call dtrfac(n,l,incl,d,incd,u,incu) l(i) = l(i) * d(i-1) d(i) = 1.0/(d(i) - l(i) * u(i-1))
*x*ubidi	d,s	Solves upper bidiagonal	call dubidi(n,d,incd,u,incu,x,incx) u(i+2)=(u(i)-d(i))*u(i+1))*l(i)
*x*vpoly	d,s	Vector polynomial evaluation	call dvpoly(n,x,incx,m,c,incc,y,incy) y(i)=c(1)+c(2)*x(i)+...+c(m)*x(i)**(m-1)

C VECLIB ROUTINE SUMMARY B

This appendix contains a set of tables that summarize the C VecLib routines according to function. There are eight tables that categorize the routines as follows:

- Mathematical Primitives
- Other Mathematical Functions
- Triad Operations
- Relational Primitive Operations
- Logical Primitive Operations
- Reduction Functions
- Conversion Primitives
- Miscellaneous Operations

Each table describes the functions and then lists the C versions of the routines. The *iPSC®/2 Programmer's Reference Manual* lists these routines alphabetically.

The routines are listed by root name. An italic *x* represents where the data type is specified in the routine name. The data type upon which a function operates is indicated by the first or second letter of the routine name; "d" for double, "s" for single, "i" for integer, "l" for logical, and "c" for complex, and "z" for double precision complex. For example, **dasum** is the double precision routine to calculate the sum of absolute values, while **sasum** is the single precision version.

The first three columns contain the routine's root name, data types and a brief description. The fourth column gives the calling sequence and equivalent in the language specified. A "d" is used in most examples to represent the double precision version of the routine. When "i" is used as an index in the C equivalent statement, "i" ranges from 0 to n-1.

Table B-1. Mathematical Primitives

Root Name	Data Type	Description	Calling Sequence/C Equivalent
*x*swap	d,s,i,c,z	Swap vectors	`dswap(n,x,incx,y,incy);` `t = y[i];  y[i] = x[i];  x[i] = t;`
*x*copy	d,s,i,l,c,z	Copy vector	`dcopy(n,x,incx,y,incy);` `y[i] = x[i];`
*x*fill	d,s,i,l,c,z	Fill vector	`dfill(n,alpha,x,incx);` `x[i] = alpha;`
*x*neg	d,s,i,c,z	Change sign	`dneg(n,x,incx);` `x[i] = -x[i];`
*x*vneg	d,s,i,c,z	Negate vector	`dvneg(n,x,incx,y,incy);` `y[i] = -x[i];`
*x*sadd	d,s,i,c,z	Scalar plus vector	`dsadd(n,alpha,x,incx,y,incy);` `y[i] = alpha + x[i];`
*x*vadd	d,s,i,c,z	Vector addition	`dvadd(n,x,incx,y,incy,z,incz);` `z[i] = x[i] + y[i];`
*x*ssub	d,s,i,c,z	Scalar vector subtraction	`dssub(n,alpha,x,incx,y,incy);` `y[i] = alpha - x[i];`
*x*vsub	d,s,i,c,z	Vector subtraction	`dvsub(n,x,incx,y,incy,z,incz);` `z[i] = x[i] - y[i];`
*x*smul	d,s,i,c,z	Scalar times vector	`dsmul(n,alpha,x,incx,y,incy);` `y[i] = alpha * x[i];`
*x*vmul	d,s,i,c,z	Vector element-wise multiplication	`dvmul(n,x,incx,y,incy,z,incz);` `z[i] = x[i] * y[i];`
*x*scal	d,s,c,z	Scalar times a vector to itself	`dscal(n,alpha,x,incx);` `x[i] = alpha * x[i];`
*x*sdiv	d,s,i,c,z	Scalar divided by vector	`dsdiv(n,alpha,x,incx,y,incy);` `y[i] = alpha / x[i];`
*x*vrecp	d,s,c,z	Vector reciprocal	`dvrecp(n,x,incx,y,incy);` `y[i] = 1.0 / x[i];`
*x*vdiv	d,s,i,c,z	Element-wise vector division	`dvdiv(n,x,incx,y,incy,z,incz);` `z[i] = x[i] / y[i];`

Table B-2. Other Mathematical Functions

Root Name	Data Type	Description	Calling Sequence/C Equivalent
*x*vabs	d,s,i,c,z	Element-wise absolute value	`dvabs(n,x,incx,y,incy);` `y[i] = CABS(x[i]);`
*x*vmax	d,s	Vector element-wise maximum	`dvmax(n,x,incx,y,incy,z,incz);` `z[i] = MAX(x[i],y[i]);`
*x*vmin	d,s	Vector element-wise minimum	`dvmin(n,x,incx,y,incy,z,incz);` `z[i] = MIN(x[i],y[i]);`
*x*vamax	d,s	Vector element-wise maximum absolute value	`dvamax(n,x,incx,y,incy,z,incz);` `z[i] = MAX(ABS(x[i]),ABS(y[i]));`
*x*vamin	d,s	Vector element-wise minimum abs. value	`dvamin(n,x,incx,y,incy,z,incz);` `z[i] = MIN(ABS(x[i]),ABS(y[i]));`
*x*vpow	d,s	Element-wise power function	`dvpow(n,x,incx,y,incy,z,incz);` `z[i] = pow(x[i], y[i]);`
*x*vexp	d,s	Element-wise exponential	`dvexp(n,x,incx,y,incy);` `y[i] = exp(x[i]);`
*x*vlg10	d,s	Element-wise base 10 logarithm.	`dvlg10(n,x,incx,y,incy);` `y[i] = log10(x[i]);`
*x*vlog	d,s	Element-wise natural logarithm	`dvlog(n,x,incx,y,incy);` `y[i] = log(x[i]);`
*x*vatan	d,s	Element-wise inverse-tangent	`dvatan(n,x,incx,y,incy);` `y[i] = atan(x[i]);`
*x*vatn2	d,s	Element-wise inverse-tangent of quotient	`dvatn2(n,x,incx,y,incy,z,incz);` `z[i] = atan2(x[i],y[i]);`
*x*vcos	d,s	Element-wise cosine	`dvcos(n,x,incx,y,incy);` `y[i] = cos(x[i]);`
*x*vsin	d,s	Element-wise sine	`dvsin(n,x,incx,y,incy);` `y[i] = sin(x[i]));`
*x*vsqrt	d,s	Element-wise square root	`dvsqrt(n,x,incx,y,incy);` `y[i] = sqrt(x[i]);`
*x*random	d,s	Generates pseudo-random number	`d = drandom();`
*x*vrandom	d,s	Pseudo-random vector generation	`dvrandom(n,x,incx);`

Table B-3. Triad Operations

Base Name	Data Type	Description	Calling Sequence/C Equivalent
*x*axpy	d,s,c,z	Scalar times vector plus a vector to itself	`daxpy(n,alpha,x,incx,y,incy);` `y[i] = alpha*x[i] + y[i];`
*x*svmvt	d,s	Scalar minus vector quantity times vector	`dsvmvt(n,alpha,x,incx,y,incy,z,incz);` `z[i] = (alpha - x[i]) * y(i);`
*x*svpvt	d,s	Scalar plus vector quantity times vector	`dsvpvt(n,alpha,x,incx,y,incy,z,incz);` `z[i] = (alpha + x[i]) * y[i];`
*x*svtsp	d,s	Scalar times vector quantity plus scalar	`dsvtsp(n,alpha,beta,x,incx,y,incy);` `y[i] = alpha * x[i] + beta;`
*x*svtvm	d,s	Scalar times vector quantity minus vector	`dsvtvm(n,alpha,x,incx,y,incy,z,incz);` `z[i] = alpha * x[i] - y[i];`
*x*svtvp	d,s	Scalar times quantity of vector plus vector	`dsvtvp(n,alpha,x,incx,y,incy,z,incz);` `z[i] = alpha * x[i] + y[i];`
*x*svvmt	d,s	Scalar times quantity of vector minus vector	`dsvvmt(n,alpha,x,incx,y,incy,z,incz);` `z[i] = alpha * (x[i] - y[i]);`
*x*svvpt	d,s	Scalar times quantity of vector plus vector	`dsvvpt(n,alpha,x,incx,y,incy,z,incz);` `z[i] = alpha * (x[i] + y[i]);`
*x*svvtm	d,s	Scalar minus quantity of vector times vector	`dsvvtm(n,alpha,x,incx,y,incy,z,incz);` `z[i] = alpha - x[i] * y[i];`
*x*svvtp	d,s	Scalar plus quantity of vector times vector	`dsvvtp(n,alpha,x,incx,y,incy,z,incz);` `z[i] = alpha + x[i] * y[i];`
*x*vvmvt	d,s	Vector minus vector quantity times vector	`dvvmvt(n,w,incw,x,incx,y,incy,z,incz);` `z[i] = (w[i] - x[i]) * y[i];`
*x*vvpvt	d,s	Vector plus vector quantity times vector	`dvvpvt(n,w,incw,x,incx,y,incy,z,incz);` `z[i] = (w[i] + x[i]) * y[i];`
*x*vvtvm	d,s	Vector times vector quantity minus vector	`dvvtvm(n,w,incw,x,incx,y,incy,z,incz);` `z[i] = w(i) * x[i] - y[i];`
*x*vvtvp	d,s	Vector times vector quantity plus vector	`dvvtvp(n,w,incw,x,incx,y,incy,z,incz);` `z(i) = w[i] * x[i] + y[i];`
*x*vvvtm	d,s	Vector minus quantity of vector times vector	`dvvvtm(n,w,incw,x,incx,y,incy,z,incz);` `z[i] = w[i] - x[i] * y[i];`

Table B-4. Relational Primitive Operations

Root Name	Data Type	Description	Calling Sequence/C Equivalent
*x*eq	d,s,i	Vector element equality	`deq(n,x,incx,y,incy,lz,incz);` `lz[i] = x[i] == y[i];`
*x*seq	d,s,i	Vector equal to scalar	`dseq(n,alpha,x,incx,ly,incy);` `ly[i] = alpha == x[i];`
*x*ge	d,s,i	Vector element greater than or equal	`dge(n,x,incx,y,incy,lz,incz);` `lz[i] = x[i] >= y[i];`
*x*sge	d,s,i	Scalar greater than or equal to vector	`dsge(n,alpha,x,incx,ly,incy);` `ly[i] = alpha >= x[i];`
*x*gt	d,s,i	Vector element greater than	`dgt(n,x,incx,y,incy,lz,incz);` `lz[i] = x(i) > y[i];`
*x*sgt	d,s,i	Scalar greater than vector	`dsgt(n,alpha,x,incx,ly,incy);` `ly[i] = alpha > x[i];`
*x*sle	d,s,i	Scalar less than or equal to vector	`dsle(n,alpha,x,incx,ly,incy);` `ly[i] = alpha <= x[i];`
*x*slt	d,s,i	Scalar less than vector	`dslt(n,alpha,x,incx,ly,incy);` `ly[i] = alpha < x[i];`
*x*ne	d,s,i	Vector element inequality	`dne(n,x,incx,y,incy,lz,incz);` `lz[i] = x[i] != y[i];`
*x*sne	d,s,i	Vector not equal to scalar	`dsne(n,alpha,x,incx,ly,incy);` `ly[i] = alpha != x[i];`

Table B-5. Logical Primitive Operations

Root Name	Data Type	Description	Calling Sequence/C Equivalent
land	l	Vector logical AND with vector	`land(n,x,incx,y,incy,z,incz);` `z[i] = x[i] && y[i];`
lnot	l	Vector logical negation	`lnot(n,x,incx,y,incy);` `y[i] = !x[i];`
lor	l	Vector logical OR with vector	`lor(n,x,incx,y,incy,z,incz);` `z[i] = x[i] || y[i];`
lsand	l	Scalar logical AND with scalar	`lsand(n,alpha,x,incx,y,incy);` `y[i] = alpha && x[i];`
lsor	l	Scalar logical OR with vector	`lsor(n,alpha,x,incx,y,incy);` `y[i] = alpha || x[i];`
lcopy	d,s,i,l,c,z	Copy vector	`lcopy(n,x,incx,y,incy);` `y[i] = x[i];`
lfill	d,s,i,l,c,z	Fill vector	`lfill(n,alpha,x,incx);` `x[i] = alpha;`

Table B-6. Reduction Functions

Root Name	Data Type	Description	Calling Sequence/C Equivalent
*x*asum	d,s	Sum of absolute values	`d = dasum(n,x,incx);` `dasum = dasum + ABS(x[i]);`

dzasum	d	Sum of absolute value of real and imaginary parts of the values	`d = dzasum(n,x,incx);`
			`dzasum = 0.0;`
			`dzasum=dzasum+ABS(x[i].r)+ABS(x[i].i);`

scasum	s	Sum of absolute value of real and imaginary parts of the vector	`s = scasum(n,x,incx);`
			`scasum = 0.0;`
			`scasum=scasum+ABS(x[i].r)+ABS(x[i].i);`

*x*sum	d,s,c,z	Vector sum	`s = dsum(n,x,incx);` `dsum = 0.0; dsum = dsum + x[i];`
*x*nrm2	d,s	Euclidean vector norm	`d = dnrm2(n,x,incx);` `dnrm2=dnrm2+x[i]*x[i];dnrm2=sqrt(dnrm2);`
*x*dot	d,s	Dot product of two vectors	`d = ddot(n,x,incx,y,incy);` `ddot = ddot + x[i]*y[i];`
*x*dotc	c,z	Dot product of two complex vectors	`c = cdotc(n,x,incx,y,incy);` `cdotc = cdotc + (CONJG(x[i])*y[i]);`
*x*dotu	c,z	Dot product of two complex vectors	`c = cdotu(n,x,incx,y,incy);` `cdotu = cdotu + x[i]*y[i];`

i*x*amax	d,s	Index of maximum absolute value	`i = idamax(n,x,incx);`
			`if (n>0) idamax = 0;`
			`if (ABS(x[i])>ABS(x[idamax])) idamax=i;`

i*x*amin	d,s	Index of minimum absolute value	`i = idamin(n,x,incx);`
			`idamin = min (1, max(0,n));`
			`if (ABS(x[i])<ABS(x[idamin])) idamin=i;`

i*x*max	d,s	Index of maximum value	`i = idmax(n,x,incx);`
			`idmax = -1; if(n>0) idmax = 0;`
			`if (x[i] > x[idmax]) idmax = i;`

i*x*min	d,s	Index of minimum value	`i = idmin(n,x,incx);`
			`idmin = -1; if (n>0) idmin=0;`
			`if (x[i] < x[idmin]) idmin = i;`

icount	l	Number of logical true values	`i = icount(n, lx, incx);` `if (lx[i]) icount = icount + 1;`
ifirst	i	Index of first logical true value	`i = ifirst(n,lx,incx);` `ifirst=-1; if(lx[i]){ifirst=i;break;};`
ilast	i	Index of last logical true value	`i = ilast(n,lx,incx);` `ilast = -1; if (lx[i]) ilast = i;`
lany	l	Logical true if any of x are true	`l = lany(n,x,incx);` `lany = 0; if (x[i]) lany = 1;`

Table B-7. Conversion Primitives

Root Name	Data Type	Description	Calling Sequence/C Equivalent
*x*vcmplx	c,z	real to complex	`cvcmplx(n,sx,incx,sy,incy,z,incz);` `z[i] = CMPLX(sx[i],sy[i]);`
*x*vconjg	c,z	Conjugate of a complex vector	`cvconjg(n,x,incx,y,incy);` `y[i] = CONJG(x[i]);`
vdble	d	Single to double precision	`vdble(n,sx,incx,dy,incy);` `dy[i] = (double)sx[i];`
*x*vfix	d,s	Truncate elements to integer values	`dvfix(n,x,incx,iy,incy);` `iy[i] = (int)x[i];`
*x*vfloa	d,s	Convert integer to floating point	`dvfloa(n,ix,incx,y,incy);` `y[i] = ix[i];`
*x*vimag	c,z	Extract imaginary part of complex vector	`cvimag(n,x,incx,sy,incy);` `sy[i] = x[i].i;`
*x*vreal	c,z	Extract real part of complex vector	`cvreal(n,x,incx,sy,incy);` `sy[i]= x[i].r;`
vsngl	s	Double to single precision	`vsngl(n,dx,incx,sy,incy);` `sy[i] = (float)dx[i];`

Table B-8. Miscellaneous Functions

Root Name	Data Type	Description	Calling Sequence/C Equivalent
*x*scatr	d,s,i	Vector scatter	dscatr(n,x,incx,iy,incy,z,incz); z[iy[i]] = x[i];
*x*gathr	d,s,i	Vector gather	dgathr(n,x,incx,iy,incy,z,incz); z[i] = x[iy[i]];
*x*ramp	d,s,i	Ramp function	dramp(n,alpha,beta,x,incx); x[i] = alpha + (i-1)*beta;
*x*clip	d,s	Clip to interval [alpha,beta]	dclip(n,x,incx,alpha,beta,y,incy); y[i] = MIN(MAX(x[i],alpha),beta);

*x*iclip	d,s	Inverted clip	diclip(n,x,incx,al,bl,y,incy);
			if(x[i]<(al+bl)/2) y[i]=MIN(x[i],alpha);
			else {y[i] = MAX(x[i],beta)};

*x*cndst	d,s	Conditional assignment	dcndst(n,x,incx,ly,incy,z,incz); if (ly[i]) z[i] = x[i];
*x*mask	d,s	Conditional assignment	dmask(n,w,incw,x,incx,ly,incy,z,incz); if (ly[i]) z[i] = w[i]; else z[i] = x[i];
*x*fft	c,z	Fast Fourier Transform	cfft(n,x,incx,y,incy); y = fft(x);
*x*ifft	c,z	Inverse Fast Fourier Transform	y = cifft(n,x,incx,y,incy); y = ifft^{-1}(x);

*x*rot	d,s	Apply a plane rotation	drot(n,x,incx,y,incy,c,s);
			t=x[i]; x[i]= t * c + y[i] * s;
			y[i] = -t * s + y[i] * c;

*x*rotg	d,s	Construct a Givens plane rotation	drotg(a, b, c, s);
*x*folr	d,s	First order linear recurrence routine	dfolr(n,x,incx,y,incy,z,incz); z[i+1] = y[i] + z[i] * x[i];
*x*solr	d,s	Second order recurrence routine	dsolr(n,w,incw,x,incx,y,incy,z,incz); z[i+2]=w[i]+z[i+1]*x[i]+z[i]*y[i];
*x*lbidi	d,s	Solves lower bidiagonal	dlbidi(n,l,incl,b,incb,x,incx); x[i] = b[i] - l[i] * x[i-1];

*x*trfac	d,s	LU factorization of matrix tri-diagonal	dtrfac(n,l,incl,d,incd,u,incu);
			l[i] = l[i] * d[i-1];
			d[i] = 1.0/(d[i] - l[i] * u[i-1]);

*x*ubidi	d,s	Solves upper bidiagonal	dubidi(n,d,incd,u,incu,x,incx); u[i+2]=(u[i]-d[i])*u[i+1])*l[i];
*x*vpoly	d,s	Vector polynomial evaluation	dvpoly(n,x,incx,m,c,incc,y,incy); y[i] = x[i] * y[i] + c[j];

ERROR MESSAGES C

This appendix lists the VX error messages.

Error messages are generated only when running VecLib routines in *libvxdbvec.a* or *libvxsxdbvec.a*. They are listed below alphabetically with a description following each message. "<x>" represents vector or scalar parameter names and "<ddot>" is used to represent any of the VecLib routine names.

VX veclib: Error in length for argument <x> in call to <ddot>

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support Customer Support

VX veclib: Error in memory space for argument <x> in call to <ddot>

Description: The data is not on vector board memory, but is on the node board.

Cause: Incorrect use of **vxld**.
Action: Move the data section containing the vector to the VX memory using **vxld**.

VX veclib: Error in physical alignment for argument <x> in call to <ddot>

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support Customer Support

VX veclib: Error in physical bounds check for argument <x> in call to <ddot>

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support Customer Support

VX veclib: Error in type for check of argument <x> in call from <ddot>

Cause: Internal error message, *libvxdbvec.a* or *libvxsxdbvec.a* error.
Action: Call iSC Customer Support Customer Support

VX veclib: Error in virtual alignment for argument <x> in call to <ddot>

Description: The virtual address's offset modulo the data size in 8 bit bytes is not equal to zero.
Cause: Ignored warning from compiler about equivalence misalignment.
Action: Change equivalence statement.

VX veclib: Error in virtual bounds check for argument <x> in call to <ddot>

Description: The data is not contiguous in virtual memory.
Cause: Requesting vector operations outside the bounds of an array.
Action: Locate error in your Fortran program.

VX veclib: The following enabled exceptions have occurred: NaN (not-a-number), overflow, underflow.

Cause: Requested modification of exceptions using vpexcept and algorithm-produced exception.
Action: Modify algorithm used so exception does not occur.

Vector size error

Cause: Internal error
Action: Call iSC Customer Support

Not enough table space for vid/sid

Cause: Internal error
Action: Call iSC Customer Support

tried to dealloc below starting point

Cause: Internal error
Action: Call iSC Customer Support

no space in pool

Cause: Internal error
Action: Call iSC Customer Support

seq stack over/under flow interrupt

Cause: Internal error
Action: Call iSC Customer Support

VX illegal microcode

Cause: Internal error, or mismatched -single or -double usage.
Action: Call iSC Customer Support

VX AE - [+ under] [+ over] [+ nan] [*under] [* over] [* nan]

Cause: Internal error
Action: Call iSC Customer Support

VORTEX Parity Error

Cause: Software or hardware error
Action: Call iSC Customer Support

illegal address interrupt

Cause: Address far outside of an array
Action: Correct addressing error.

function is not loaded

Cause: Tried to use microcode routine which was not loaded. For example, **saxpy** with the -double switch.

Action: Correct program and/or the switches on the **vast2** or **f77** commands which build the program.

DATA ALIGNMENT D

iPSC/2-VX hardware requires that data elements be aligned on their natural boundaries to speed memory access. The iPSC/2 Fortran and C compilers do this automatically when you use the **-vx** switch with the **f77** or **cc** command, which instructs the compiler to attempt to align data correctly. This means that the starting address of a variable must be an integer multiple of the number of bytes occupied by that variable. For example, real numbers are four bytes long. Therefore, real numbers must have addresses that are integer multiples of four. Double precision numbers must have addresses which are multiples of eight. Integers follow the same rules as real numbers. Complex values are like double precision values and must be a multiples of eight; double precision complex numbers must be multiples of 16.

In some rare cases, the **-vx** switch may not align all variables on their natural boundaries, causing unpredictable results such as wrong answers or parity errors. In this case, you can use the tool **nm**, as described in the final section of this appendix to make sure that all variables used on the vector boards are aligned correctly.

Character strings must also start on 32-bit boundaries, which the compilers will not do automatically. When sending or receiving character strings, it is best to ensure that these strings are multiples of four bytes.

If data elements are not aligned properly, incorrect results are produced. You can use the *libvxdbvec.a* library to provide error messages when data alignment rules have been violated.

Data formats supported by the iPSC/2-VX are as follows:

Data Type	Data Format	Operand
Integer	Two's complement	32 bits
Single Precision Floating Point	IEEE-754	32 bits
Double Precision Floating Point	IEEE-754	64 bits
Single Precision Complex	IEEE-754	64 bits (Real, Imaginary)
Double Precision Complex	IEEE-754	128 bits (Real, Imaginary)

Areas where your actions (or inactions) can affect data alignment are Fortran common blocks and equivalence statements, and certain cases in C.

COMMON BLOCKS IN FORTRAN

The following example shows an incorrectly structured common block. Variables are placed in consecutive locations in memory and common blocks always begin on 8-byte boundaries. This common block has a 2-byte integer, followed by a 4-byte real, followed by an 8-byte double precision.

```
      real*8       d
      integer*2    i
      real         r
      common/block/i, r, d
c     i is at offset = 0
c     r is at offset = 2
c     d is at offset = 6
```

This ordering causes the real and double precision variables to be incorrectly aligned for the vector processor hardware. The offsets of "r" (2) and "d" (6) are not legal starting addresses for vector applications.

NOTE

The Green Hills Fortran compiler detects incorrectly structured common blocks, issues warnings and then pads the misaligned common block when programs are compiled with the **-vx** switch.

The easiest way to ensure correct alignment is to define common blocks with the longest variables first, followed by next longest, and so on. For example:

```
      common/block/d, r, i
c     d is at offset = 0
c     r is at offset = 8
c     i is at offset = 12
```

Here the offset of "r" is 8 which is an integer multiple of 8 and the offset of "i" is 12 which is an integer multiple of 4.

EQUIVALENCE STATEMENTS IN FORTRAN

If you use an equivalence statement in Fortran to change a real to a double precision or a double precision to an integer, data may not be aligned on the proper boundaries. For example, in the following sequence, either D1 or D2 will not be on a boundary that is a multiple of 8:

```
real R(5)
double precision D1, D2
equivalence (R(1), D1)
equivalence (R(4), D2)
```

The iPSC/2 Fortran compiler issues warning messages for misaligned equivalence statements when programs are compiled with the **-vx** switch. You should be careful when using the result of an equivalence statement as an argument in any vector routine.

The C compiler does not issue any such warnings, and so great care should be taken with assignments.

ALIGNMENT NOTES FOR C

In general, if you use the same compile switches when compiling different parts of your code, the C compiler will align the data correctly. However, different switches cause different data alignment. This can cause you particular difficulty if you have structures of this kind:

```
struct {
    int a;
    double b;
    }
```

If you compile this without the **-vx** switch, **a** has an offset of 0, and **b** has an offset of 4. If you compile this with the **-vx** switch, **a** has an offset of 0, and **b** has an offset of 8.

The result of this is that if you mix compile switches when compiling different parts of your code, and you have structures like this in include files, it is possible for different routines to look at the data alignment of the same structure differently, and the results will be unpredictable.

CHECKING DATA ALIGNMENT

One way to check for correct data alignment is to use the debug version of VecLib (*libvxdbvec.a*) when you link, using the switch **-vecdb**. If you do this, errors are returned at runtime if any data is misaligned. In addition, errors are reported if read/write protection is violated (General Protection (GP) faults occur).

Another way to check data alignment is to compile your program with the -g switch which places the symbol table information in the object file. Then, after linking the program, use the following command sequence to produce a listing of the VX symbols allocated on the vector board.

```
nm  -x  node | fgrep  0xc0
```

An example of a symbol table is shown below:

Variable Name	Variable Address		Variable Type	Allocation Data Type		Variable Section Name
vx_base	0xc08fc000	extern	int			data
s_mem_array	0xc08fc000	extern	Uint[1660]	0x19f0		data
dmscr	0xc08fd9f0	extern	Uint[1024]	0x1000		data
vpfast_	0xc08fe9f0	extern	Uint[4096]	0x4000		data
vpdesc_	0xc09029f0	extern	struct-vpdesc	0x0050		data
d_mem_array	0xc0902a40	extern	Uint[5636]	0x5810		data
random_seed_	0xc0908250	extern	Uint			data
tmp_tbl	0xc08fd950	extern	int			data
_vx_allocs	0xc08fd998	extern	int			data
vxram_start	0xc0908258	extern	int			data
sa	0xc0908310	extern				comm
szero	0xc0908314	extern				comm
sx	0xc0908318	extern				comm
sy	0xc0910318	extern				comm
nsize	0xc09082d0	extern				data

For real, logical, and integer variables, the least significant digit in column 2 should be 0, 4, 8, or C. For double precision and complex, the least significant digit in column 2 should be 0 or 8.

RESERVED WORDS E

INTRODUCTION

The following reserved words are names of subroutines and common blocks that are required by the system to execute vector routines. You should not use these words unless you are writing microcode for the iPSC/2-VX. They are listed here for informational purposes only.

RESERVED SUBROUTINE NAMES

The following subroutine names found in the *libvxvec.a* and *libvxsxvec.a* libraries are reserved to support the vector runtime routines:

C and Fortran Versions	C Versions Only
vpcqr	__vxchecks
vpwctrl	__vxcheckv
	__vxchecke

RESERVED COMMON BLOCKS IN FORTRAN

The following common blocks are reserved to support the vector runtime routines:

/vpfast/
/vpregs/
/vpdesc/

RESERVED C PUBLIC VARIABLES

The following public variables are reserved to support the vector runtime routines:

vpdesc__	tmp__tbl
vpfast__	s__mem__array
vpregs__	d__mem__array
	p__mem__array