The
Connection Machine
System

pndbx Release Notes

Version 1.2
March 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, March 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

Internet Electronic Mail:	`customer-support@think.com`
uucp Electronic Mail:	`ames!think!customer-support`
U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1264
Telephone:	(617) 234-4000

pndbx Version 1.2 Release Notes

1 About Version 1.2

Version 1.2 of **pndbx** works on CM-5 systems (with or without vector units), and runs under either CMOST 7.1.5 or CMOST 7.2. New features include:

- support for debugging of message-passing CM Fortran and C* programs
- support for debugging DPEAC code

In addition, **pndbx** Version 1.2 contains a number of bug fixes.

You can find out information about **pndbx** bugs by consulting its on-line bug-update file, **pndbx-1.2.bugupdate**. By default, this file is in the directory **/usr/doc**; if it isn't there, ask your system administrator for its location at your site. The file will be updated monthly.

2 Debugging Message-Passing CM Fortran Code

Starting with CMMD Version 3.0, you will be able to write programs consisting of CM Fortran code running on each node, passing messages between the nodes using CMMD. This capability provides one way of making use of the vector units from message-passing code.

To build such node-level CM Fortran programs, specify the **-node** option to the CM Fortran compiler. To build a debuggable version of a message-passing CM Fortran program, be sure to also specify the **-g** option to the CM Fortran compiler, on both the compile and the link steps.

Here is a sample compilation:

```
% cmf -g -node -o samp.x samp.fcm
```

And here is how you would start a **pndbx** session for the resulting executable program:

```
[In one window:]
% prism -C samp.x
(prism) stop in cmmd_debug
(prism) run

[In another window:]
% cmps
% pndbx samp.x pid
(pndbx 0)
```

pndbx understands all CM Fortran data types: integer, real, double, complex, double complex, and character. Arrays are printed in their entirety, one element per line. (The built-in variable **$print_width** can be used to change this default.) You can specify array sections using CM Fortran syntax. Arbitrary expressions can be evaluated, with some restrictions. You can use **assign** to modify variables.

The following example illustrates these features:

```
(pndbx 0) whatis u
(CM based) double precision U(1:10)
(pndbx 0) print u
(1)     1.1
(2)     1.1
(3)     1.1
(4)     1.1
(5)     1.1
(6)     1.1
(7)     1.1
(8)     1.1
(9)     1.1
(10)    1.1
(pndbx 0) print u(1:4)
```

```
(1)      1.1
(2)      1.1
(3)      1.1
(4)      1.1
(pndbx 0) set $print_width = 2
(pndbx 0) print u
(1:2)    1.1      1.1
(3:4)    1.1      1.1
(5:6)    1.1      1.1
(7:8)    1.1      1.1
(9:10)   1.1      1.1
(pndbx 0) print u(1:4)+1
(1:2)    2.1      2.1
(3:4)    2.1      2.1
(pndbx 0) assign u = 2.2
(pndbx 0) print u
(1:2)    2.2      2.2
(3:4)    2.2      2.2
(5:6)    2.2      2.2
(7:8)    2.2      2.2
(9:10)   2.2      2.2
```

If you need to get at the lower-level details of CM Fortran array descriptors, this feature may be helpful:

```
(pndbx 0) print &u
CM array, descriptor address = 0xb8aa4 (print *&U to see
the entire descriptor)
(pndbx 0) print *&u
(desc_or_object_kind = 1025, debug_info_ptr = 0xb8a98,
element_type = 5, spare1 = 0, spare2 = 0, cm_location =
1342187272, user_rank = 1, spare4 = 757192, spare5 =
757084, home = 3, initial_data = -1, is_modified = 0,
array_geometry = 1468752, spare6 = -1, spare7 = 1,
spare8 = 757080, spare9 = -1, is_slicewise = 1, ele-
ment_size = 8)
```

Because of the way that **pndbx** accesses data in CM Fortran programs, you may notice that printing expressions involving CM Fortran arrays is slow. You can work around this, to some extent, by using indexing to select only those array sections you want to see.

3 Debugging Message-Passing C* Code

pndbx Version 1.2 contains some support for debugging message-passing C* code.

To debug message-passing C* code, supply the **-node** and **-g** options to the compiler, as described in the previous section.

pndbx can print parallel variables of any of the scalar base types (for example, **char**, **short**, **int**, **float**, **double**). The result is treated as if it were an array. You can look at the resulting array in its entirety, or you can use CM Fortran array syntax to look at sections or individual elements. Here is an example of looking at a parallel **int** in its entirety, then looking at a section of it:

The C* source code:

```
...
shape [16]s;
int:s i1 = 2 ;
...
```

The **pndbx** session:

```
(pndbx 0) whatis i1
parallel int i1;
(pndbx 0) print i1
(2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)
(pndbx 0) print i1(0:8)
(2, 2, 2, 2, 2, 2, 2, 2, 2)
```

You can view parallel arrays (that is, per-processor arrays) in their entirety, but at this time sections of these arrays cannot be printed. Also, parallel structs currently cannot be printed. For unsupported objects, **pndbx** will print out an error message of the form:

```
(pndbx 0) print t1
Sorry, at this time pndbx does not support printing of
parallel structs.
```

4 Debugging DPEAC Code

If you specify the **-g** flag to the DPAS assembler when you assemble your DPEAC code, **pndbx** Version 1.2 automatically displays DPEAC instructions when they are encountered. The example below shows **pndbx** displaying a **dpwrt** instruction after a **stepi** and on an examine-instruction command:

```
(pndbx 0) stepi
stopped in cmpe_floatarray_0_ at 0x2a44
cmpe_floatarray_0_+0xc: dpwrt *, %g5, S2
(pndbx 0) print $pc
10820
(pndbx 0) 10820/i
cmpe_floatarray_0_+0xc: dpwrt *, %g5, S2
```

You can disable the display of DPEAC using the **dpeac** toggle. In the example below, DPEAC display is turned off and we see the underlying SPARC instructions:

```
(pndbx 0) dpeac
dpeac mode turned off
(pndbx 0) 10820/4i
cmpe_floatarray_0_+0xc: sethi   %hi(0xd0000000), %g3
cmpe_floatarray_0_+0x10:        st      %g0, [%g3 + 296]
cmpe_floatarray_0_+0x14:        sethi   %hi(0xd0800000), %g3
cmpe_floatarray_0_+0x18:        st      %g5, [%g3 + 8]
```

To get at vector-unit registers, there is a built-in variable called **$dp_state**. This is an array of four structures (one for each vector unit). It can be printed in the usual ways. In the example below, we've used **$dp_state** to see the state of vector unit 0:

```
(pndbx 0) set $hexints = 1
(pndbx 0) print $dp_state[0]
(alu_mode = 0x0, vector_length = 0x0, stride_rs1 = 0x0,
stride_memory = 0x0, instruction_ext = 0x80100000, instruc-
tion_ext_enb = 0x0, vector_mask = 0x0, vector_mask_buffer =
0x0, vector_mask_mode = 0x0, vector_mask_direction = 0x0, sta-
tus_enable = 0x0, status = 0x0, heap_limits = 0x0,
stack_limits = 0x16f016f, memory_access_mode = 0x1, inter-
rupt_enable_green = 0x0, interrupt_enable = 0x73f,
interrupt_cause_green = 0x0, interrupt_cause = 0x0,
bad_address_high = 0x0, bad_address_low = 0x13e000,
bad_instruction_high = 0x0, bad_instruction_low = 0x87f8020,
current_element = 0x1, interrupt_cause_green_stored = 0x0,
interrupt_cause_stored = 0x0, data_regs =
(0)     0x0
```

```
(1)     0x0
(2)     0x0
(3)     0x0
(4)     0x0
(5)     0x0
...
(127)   0x0
```

To look at the vector registers in a format other than integer, **$dp_state** can be used as the "address" in a memory-examine command. For example, the following **pndbx** command looks at the 128 registers of DP 0 in single-precision **float** format:

```
(pndbx 0) $dp_state[0].data_regs/128f
  0:   0x00000000     +0.000000e+00
  1:   0x00010000     +9.183550e-41
  2:   0x00000000     +0.000000e+00
  3:   0x00000030     +6.726233e-44
  4:   0xffffff00     -NaN
  ...
```