AD-A196 419

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Poker 4.1: A Programmer's Reference Guide		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Lawrence Snyder		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0264
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Washington Department of Computer Science, FR-35 Seattle, Washington 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program Arlington, VA 22217		12. REPORT DATE April 1988
		13. NUMBER OF PAGES 94
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC SELECTE JUN 2 7 1988 S D E

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Poker parallel programming environment

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This manual defines all facilities available within the Poker Parallel Programming Environment. In addition, there are three appendices, one describing the structure of the distributed library, the second describing systems related to Poker, and the third describing the use of Poker with the Cosmic Cube.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

88 6 27 031

Poker (4.1) Programmer's Reference Guide

L. Snyder
Department of Computer Science, FR-35
University of Washington
Seattle, Washington 98195

TR 88-03-03
March 1988

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

This manual defines all facilities available within the Poker Parallel Programming Environment. In addition, there are three appendices, one describing the structure of the distributed library, one describing systems related to Poker, and the third describing porting to the Cosmic Cube.

This document has been funded in part by the Office of Naval Research Contract No. N00014-86-K-0264 and the National Science Foundation Grant No. CCR-8416878.

Poker (4.1) Programmer's Reference Guide

This document gives a succinct description of the facilities available with the Poker Programming Environment. The emphasis is on what is available rather than how to achieve particular results. Although the sections are self-contained, so that they may be referred to independently, there are a few things you should know:

- Poker uses interactive graphics. The graphics are described in Section 2; the interaction is described in Section 3.
- The usual programming language notion of a "source program" as a monolithic piece of symbolic text has been replaced in Poker by a database. The way to create, view, and change the database is described in Section 4.
- Object programs (the "compiled database") are executed or emulated by Poker and snapshots of the execution can be continuously displayed. ~~(See Section 14).~~
- Poker supports a variety of CHiP architectures; the current one can be displayed or changed using the CHiP Parameters facility, Section 7.
- The back page of this document gives a summary of the commands.
- Other versions of Poker exist; consult Appendix B for your particular system.

Table of Contents

1.	First time access	3
2.	The display	4
3.	Keys and cursor motions	6
4.	Views	7
5.	Global commands	9
6.	File Management	11
7.	CHiP Parameters View	17
8.	Switch Settings	19
9.	Code Names View	22
10.	Port Names View	25
11.	IO Names View	28
12.	Process Definition Languages	31
13.	Command Request	57
14.	Trace	60
15.	Execute Commands	63
16.	Object Systems	69
17.	Acknowledgements	71
A.	Library	A1
B.	Related Systems	B3
C.	Using Poker on the Cosmic Cube	C17
D.	Poker Command Summary	

1 First time access

To access Poker the user should include the directory "/usr/poker/bin" in his search path. This requires a (one-time) change to the PATH line of your .profile or .login file. The required modification is to append the text ":/usr/poker/bin" to the PATH line. [CAUTION: Your installation's protocol may differ.] Users of the Teletype 5620 display should add the following line to their .cshrc file: source /usr/poker/bin/setPokeralias. Then typing "setPoker" will download a poker environment into the 5620. See Appendix B.3 for device specific information.

First time users who would like a short computer-assisted overview of the Poker system should type "playpoker." This demonstration is intended to be a brief (less than ten minutes) and entertaining introduction to the major components of Poker.

To use Poker, type 'poker' from the bit-mapped display (see Section 2). The system will begin at the point where execution was previously terminated, or to the CHiP Parameters display (Section 7) if there was no previous session.

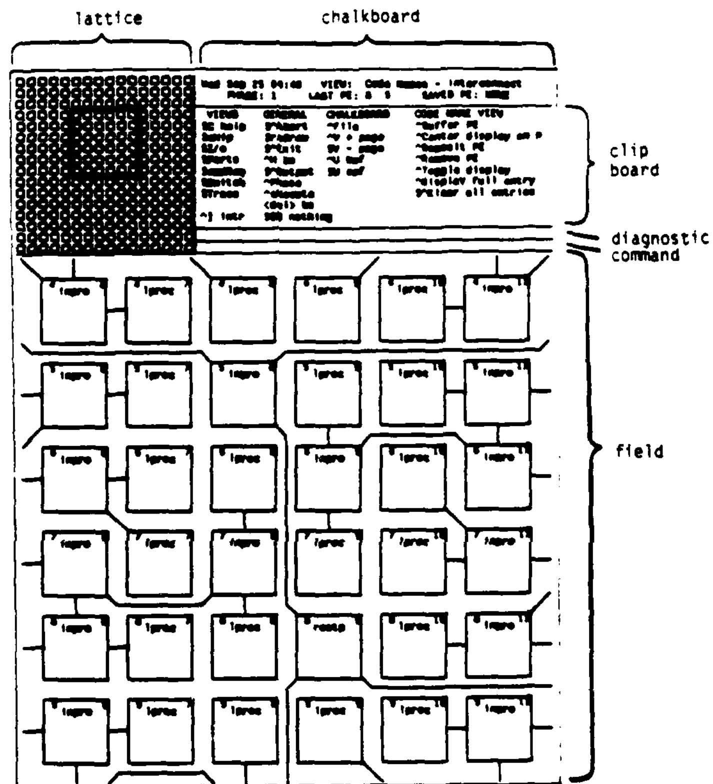


Figure 1. A typical Poker display.

2 The display

The Poker System uses two displays: a bitmapped display and a secondary terminal. [CAUTION: Your system may differ.] It is possible to use just the bitmapped display by using the pause (^z) command described below. The user should be logged into both terminals and should have both referring to a common directory. To avoid name conflicts, it is advised that the directory be empty (initially). As mentioned in Section 1, the command 'poker' from the bitmapped display terminal causes the system to be entered (assuming the modifications of Section 1).

The bitmapped display will have a form of the type shown in Figure 1. The regions of the display are as follows:

field a region showing a schematic picture of the communication structure of the program; this is the region where most programming activity takes place,

lattice a schematic diagram of the processing elements (PEs) with a box enclosing those PEs currently shown in the field; no direct user activity is available in the region,

chalkboard the upper righthand region of the display giving status information,

command line the area where textual commands are given; last line of the chalkboard,

diagnostic line the area where error indications are given; the next-to-last line of the chalkboard,

clipboard a ten line region of the chalkboard used for the display of transient information and available for displaying files,

status area displaying current state information; top two lines of the chalkboard.

The information shown in the status area is updated as follows. The time of day is current to the last write to the screen. The view and phase number are updated with the key stroke that changes them. Last PE is updated when the cursor visits a new PE; visiting switches does not change Last PE. Saved PE is set in Code Names and Port Names when the PE is buffered, and it is cleared when that PE is modified or the view changes. (See Sections 9 and 10.) Num Ticks (Command Request and Trace) is updated each time the emulator stops. (See Sections 13 and 14.)

3 Keys and Cursor Motions

The Poker system is interactive: *virtually all key strokes cause an immediate action.* Most actions are given by composite key strokes formed either by *depressing* the control key while striking a letter key (e.g., we write ^h to denote depressing the control key while striking a letter h (which causes the cursor to backspace)), or by first striking the escape key (written **$**) followed by another (possibly composite) key (e.g., **$**^e is the command to exit and return to UNIX). Should escape be inadvertently struck, it can be cleared by striking it twice more, i.e. **$$$** is a "no operation".

Movement around the display is controlled by the numeric keys of the key pad (located on the right side of the keyboard and illustrated in Figure 3). Two kinds of motions are provided: gross cursor motions and fine cursor motions. The gross cursor motions, which are two-key operations composed of an escape followed by a directional key, usually move to the next PE in the indicated direction. Fine motions, which are given just by a directional key, vary in meaning with the view being displayed. The "home" key is used to move back and forth between the command line and various positions in the field. [CAUTION: This paragraph is device-specific and the explanation may only partially apply your terminal; see Appendix B for an explanation of differences, especially the mouse usage for the Teletype 5620 and the keypad differences for the MicroVax workstation.]

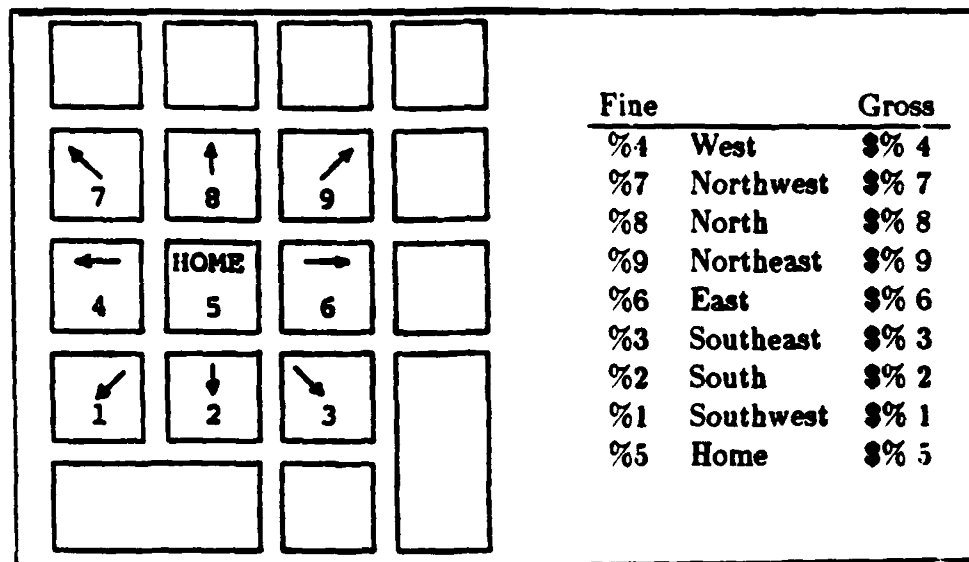


Figure 3. Cursor key bindings.

4 Views

The current state of the Poker system is stored as a database that can be queried, created and changed by using one of seven *views*. (A checklist of the information required to produce and run a Poker program is given in Figure 4.) The system begins with the view active at the end of the previous session, or if there was none, the CHiP Parameters view. The available views are:

$h cHip parameters: Specifies the architectural characteristics (e.g. number of processor elements) of the machine being programmed. (See Section 7.)

$s Switch settings: Specifies the graph of the communication structure to be used by the processor elements. (See Section 8.)

$c Code names: Specifies the name of and actual parameters of the process assigned to each processor element. (See Section 9.)

$p Port names: Specifies the symbolic names of a processor's communication channels. (See Section 10)

$i Input/output: Specifies the input and output data streams. (See Section 11.)

$r command Request: Converts the source information into object form for execution; the database is not actually displayed in this view. (See Section 13.)

$t Trace: Displays the current state of the traced variables of the executing program. (See Section 14.)

In addition, the processes themselves are defined using a standard editor on the secondary display. The processes are written in a sequential programming language such as C or XX. (See Section 12.)

As noted before the database is the Poker "source program." It is composed of a sequence of phases; there is a switch settings, code names, port names and IO names specification for each phase. A phase corresponds roughly to a single algorithm that uses a single processor interconnection structure (switch settings). For example, an algorithm to multiply two matrices would likely be a phase. Phases are numbered, but they can also be given symbolic names. (See Section 7.)

The above composite keys are always recognized. An attempt to change to the current view (e.g. the use of $s from switch settings) results in the display of the legal commands, i.e. *the key for the current view is the help request.*

Programming Checklist

To write and run a Poker program one must:

1. Comprehend the algorithm.
2. Specify the communication structure – use Switch Settings view.
3. Define processes (PE codes) – use Poker C or XX language and standard editor.
4. Assign processes to PEs with parameters – use Code Names view.
5. Name communication ports – use Port Names view.
6. Name data streams – use IO Names view.
7. Create and format data files – use standard editor and, if needed, packIO.
8. Bind stream names to file names – use Bind, an execute command.
9. Convert source "program" to object form – use Command Request view.
10. Run program and watch results – use Trace view.

Notice that the order of steps 2 - 7 is arbitrary.

Figure 4. Programming Checklist

5 Global Commands

In addition to the view commands previously described (Section 4), the following commands are always recognized:

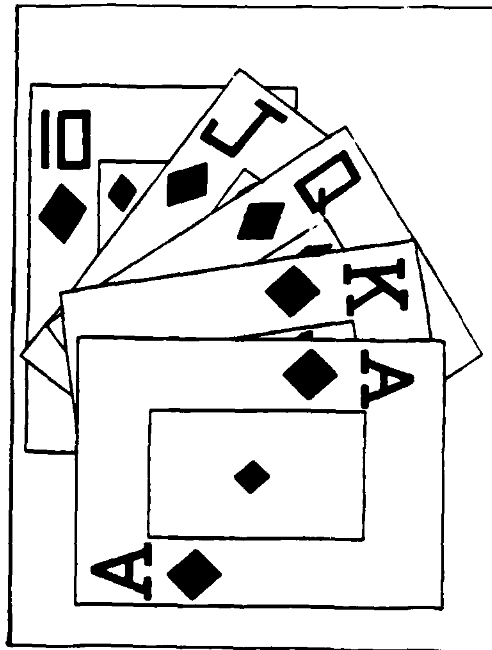
$^a	Abort	Return to UNIX without saving state. (See Interrupt key below.)
$^e	Exit	Return to UNIX and save the current source specifications (i.e. CHiPParams, SwitchSet, CodeNames, PortNames, IONames). (See Interrupt key below.)
^z	pause	Return to UNIX saving state. [CAUTION: This command is not available with all UNIX implementations.]
$^o	Output	The bitmapped display's raster memory is dumped to a file in postscript format named PS*XXXXXX* in the current directory, where *XXXXXX* is a random number. [CAUTION: Your system may vary.] As the bitmap is being uploaded, the screen is complemented; when the upload is finished, the screen is recomplimented and a message indicating completion is given. The file can then be sent to a postscript based laser printer (e.g. Apple Laserwriter).
$d	reDraw	The screen is redisplayed.
^p	Phase	The symbolic phase name or phase number given on the command line becomes the new phase; if none is given, the next phase is the current phase plus one, cyclically.
^x	eXecute	The command, given on the command line, is executed; if the command line is blank, a list of the currently available commands is given in the clipboard area. (See Sections 6 and 15.) The 'return' key has the same effect as ^x.

Commands specialized to the clipboard:

^f	File	The file whose name is given on the command line is made the "currently displayed file" and its first ten lines are displayed. If no file is given, the current contents of the Clipboard file, a file used for diagnostics and auxiliary functions, is given.
^v	+page	Advance the file display a page.

$v	-page	Backup the file display by a page.
^u	eof	Advance the file display to the end of file.
$u	bof	Backup the file display to the beginning of the file.

Additionally, the (UNIX) interrupt key is bound to ^]. If Poker terminates with Abort or Exit and the user wishes to preempt drawing of the final display, the user's "normal" UNIX interrupt key (usually ^c) can be used. (Alternatively, invoking Poker with the -c flag bypasses drawing the final display.)



Final Display

6 File Management

As a programming environment, Poker makes extensive use of many types of files; Figure 5 gives a summary of the types. This section gives a description of the contents of the files and *a description of the copy command.*

Source Database Files: Poker replaces the usual notion of a source program – a monolithic piece of symbolic text – by a database that is displayed using views (Section 4). Although views generally show a composite picture of a portion of the database built from information of several database entities, all the information entered in one view is regarded as one entity and stored externally as one file:

View	Entity Name	FileName
CHiP Parameters	CP	CHiPParams
Switch Settings	SS	SwitchSet
Code Names	CN	CodeNames
Port Names	PN	PortNames
IO Names	IO	IONames

Note that although these files are regular UNIX files they are treated as typed information and are safely manipulated with the **copy** command (see below). Each entity file contains a description of the architecture parameters in force at the time of its definition.

Copy Command: The **copy** is an execute command (Section 15) given on the command line and followed by ^x. The activity, as with any copy command, is to move the information stored in the *from structure* to a new location in the *to structure.* The *from* and *to structures* can refer to the current internal state of the Poker system or to external files. As examples we have:

Source Database Files

CHiPParams	Information entered in CHiP Parameters view (Poker)
SwitchSet	Information entered in Switch Settings view (Poker)
CodeNames	Information entered in Code Names view (Poker)
PortNames	Information entered in Port Names view (Poker)
IONames	Information entered in IO Names view (Poker)

Process Files

<proc>.x	XX source text for process <proc> (User)
<proc>.pc	Poker C source text for process <proc> (User)
<proc>.<ext>.o	Object code for process <proc> (Poker)
<proc>.<ext>.err	Diagnostics file for process<proc> (Poker)
.pe<i>,<j>.o	Object Code for processor element <i>, <j> (Poker)
pe<i>,<j>.err	Diagnostics for processor element <i>, <j> (Poker)
<proc>.inter	Process <proc> interphase variable data (Poker)
Code_<#>.c(_o)	"Stacked" source (object) code for cosmic cube emulator
code_<#>	Object doe for cosmic sube processor (Poker)
inter_phase.c(_o)	Compilation of interphase data (Poker)

Communications File

connections	Symbolic list of source, target pairs of communication graph (Poker)
connections.o	Object form of communication graph (Poker)

Object File

SnapShot	Emulator state (Poker)

Data Files

<name>	Streams laid "side-by-side" (User/Poker)

System File

Clipboard	Save area for Clipboard displayed information (Poker)
error	Error/trace messages from Poker (Poker)

Emulator Files

PringleDefs	Interface data for the emulator (Poker)
GenericDefs.c	Interface program for generic emulator (Poker)
CosmicDefs.c	Interface data for the cosmic cube emulator (Poker)
Generic-pc	Generic Emulator
Make-x	Process file dependences for pringle (Poker)
Make-x-pc	Process file dependences for generic-x (Poker)
Make-pc	Process file dependences for generic-pc (Poker)
Make-cce	Process file dependences for cosmic cube emulator (Poker)
Make-cc	Process file dependences for cosmic cube (Poker)
Spawner	Spawner program for cosmic cube emulator
fileio	File I/O program for cosmic cube emulator

Figure 5. Summary of Files, contents and (creators).

copy PreviousSS, SS 3	loads phase 3 switch settings from a file, PreviousSS.
copy *, .	saves all source entities (* = SS, CN, PN, IO) in the current directory
copy Proglib, *	loads all source entities from a stored program into Poker; relevant ".x" or ".pc" files are also transferred.

Notice in the third example, Proglib is a directory.

The syntax is:

copy <from structure>, <to structure> ^x

where both operands have the form

<structure> [<phase>]

where the optional phase is given either symbolically or numerically and the <structure> is selected from

SS|CN|PN|IO|*|.|<name>

where

SS, CN, PN and IO refer to the current internal entities of that name,
* abbreviates the set [SS, CN, PN, IO],
. refers to the current directory,
<name> is a file name or directory name,

and the semantics are to copy the specified <from structure> entities to the specified <to structure> entities subject to the following conditions:

- A <name> given as a <from structure> must exist as an entity file (or, if appropriate to the command, a directory).

- A <name> given as a <to structure> with a nonempty phase specification must exist as a entity file.

- A <name> can be an absolute path name.

- A transfer involving * or . has the side effect of transferring those <proc>.x or <proc>.pc files whose name <proc> is mentioned in the Code Names entity.

- The characteristics of the <from structure> must "conform" to the current CHiP Parameters specifications whenever an entity is read in. If they do not conform the user is given the option to have them transformed as described under the "change" paragraphs of CHiP Parameters (Section 7).

Process Files: The sequential code executed by a single PE is called a "process" and it is specified using either Poker C or a restricted sequential language called XX, see Section 12. The symbolic text for the process named <proc> is stored in a (user defined) file in the current directory file called <proc>.pc or <proc>.x in the current directory depending on the language used. The compiler, when executed in the Command Request view, produces an object file called <proc>.<ext>.o in the current directory where <ext> is pc, x or x.pc depending on the source language used. (The x.pc extent is produced when the object machine accepts the C language.) Any errors in the compilation of process <proc> are stored in the file <proc>.<ext>.err where <ext> is pc, x or x.pc depending on the language used for the source.

The link editor binds a <proc>.<ext>.o file to a particular PE <i>,<j> and stores the result in .pe<i>,<j>.o; errors in the process are reported in pe<i>,<j>.err.

Communications Files: The communication graph given in the SS entity is compiled into an adjacency list form, i.e. a set of pairs giving the source and target of every edge. The symbolic (user readable) form of this compilation is called "connections", the object form is called "connections.o" and the diagnostics from the compilation are listed in the Clipboard file.

Object State: The state of the emulator can be saved whenever it is stopped. The file is called SnapShot when no other name is specified.

Data Files: External input/output in Poker is based on the concept of a stream, a sequence of values. Stream names are defined using the IO Names view (Section 11). Stream names are bound to file names using the bind execute command (Section 15). Streams are related to normal sequential files as follows.

A stream of n *values is a file of* n records, each record having a single field. Two streams of length n and m, respectively, form a file

the file is named <filename>, then its streams – say there are three of them – are referred to as

<filename> 1
<filename> 2
<filename> 3

For example, let the file *nums* be the four records:

136.25569102,	666.6666666,	-111.1010101,
3.1415,	0.365590,	-219.333,
-22.01,	444.0444,	515.6161,
20.02,	-11421,	212.33,

This file can be treated as three streams of four values each.

Since a Poker program will use a specific number of stream names which we would like to associate with different files when, for example, we want to run the program on successive data sets, we give the association with a bind command

bind <streamname> <filename> ^x,

which says that references to streams named <streamname> refer to the streams of the file <filename>. (See Section 15.) For example, if the program's three streams are named *data 1, data 2, and data 3*, then they can be associated with the streams of the *nums* file by the command

bind data nums

which associates stream *data 1* with stream *nums 1*, etc.

Data File Formats Records for data files have a fixed number of fields equal to the number of streams. Fields are of fixed size: twelve characters followed by a comma. Real data uses standard floating point format (%f), characters are preceded by an apostrophe ('), Booleans are given by TRUE or FALSE, and integers use normal signed representation and are taken to be sints in XX or ints in Poker C when they are small and positive.

Files not in this format can be converted to this format by using the utility program, packIO, described in Appendix B.

Streams can either be unterminated or terminated by **EOS**, mnemonic for End Of Stream. Reading past the end of an unterminated stream yields zeros which are coerced to the appropriate data type (See Section 12). Reading passed a terminated stream produces an internal PE error. To terminate a stream, place the symbols **EOS** at the end of a stream.

Emulator Files A set of files is used by Poker to support emulation or *execution on parallel* machines. All of the information is internal and need not concern the user.

Script Files The format of the script files is given in Section 15 with the specification of the script command.

7 CHiP Parameters View

Poker was developed for the CHiP family of architectures. Although Poker has been generalized to be applicable to other machines, vestiges of the CHiP machine nomenclature remain. The parameters set in this section define a logical machine and no understanding of the CHiP architectures is needed.

Purpose: To specify the characteristics of the logical machine being programmed and to define symbolic phase names.

Display: The current values of the computer's parameters are given in the command line; their meaning is described in Table 1. In the field are the symbolic names for the phases.

Activity: The cursor is moved right and left along the command line using (gross or fine) east and west cursor motions. Numbers entered replace the symbol pointed to by the cursor. The new values take effect when the view is changed provided they are in range and satisfy the constraints; no changes take place if *any* parameter is illegal.

Parameter	Range	Constraints	Default
n - size, number of PEs on the side of the lattice.	$2 \leq n \leq 64$	$n = 2^k$	8
w - internal corridor width, the routing capability (switches) between two adjacent PEs.	$1 \leq w \leq 4$		1
u - external corridor width, the routing capability (switches) between the perimeter and the edge PEs.	$1 \leq u \leq 4$	$u \leq w$	1
d - degree, number of datapaths incident to PEs (and switches).	8	fixed	8
c - crossover level, number of distinct data paths meeting at a point (switch).	$1 \leq c \leq 4$		4
p - number of phases, (the size of of the switch memory).	$1 \leq p \leq 16$		1

Table 1. Description of the CHiP Parameters.

Limitations: Certain specifications (e.g. n=64) are not possible on some systems due to inadequate page table space in the UNIX kernel.

Change *n*: If the value of *n* is increased, the old lattice becomes the upper left-hand corner of the new lattice. If *n* decreases, the new lattice retains the values of the upper left-hand corner of the old lattice.

Change *w*: A change in *w* causes routing space (switch columns) to be added or removed from the right (bottom) of vertical (horizontal) routing corridors. (Existing switches retain their settings; new switches are unset).

Change *u*: A change in *u* causes routing space to be added or removed at the perimeter. (Existing switches retain their settings; new switches are unset).

Change *c*: A change in *c* permits the number of distinct data paths that can pass through a point (switch) to be either increased or decreased.

Change *p*: If *p* is increased, phases with consecutive higher numbers are added; if *p* is decreased, phases with high indexes are removed. Added phases are clear.

Phase Names: Symbolic names for the phases can be entered in windows displayed in the field. "Home" from the command line moves to the phase name windows, and from the windows back to the command line. Motion between windows uses (gross or fine) north and south cursor keys. Phase names are (a maximum of) 16 alphanumeric characters beginning with a letter.

Global Commands

$h	help	$^a	Abort
$s	Switch	$^e	Exit
$c	Code	$^o	Output screen
$p	Port	$d	reDraw screen
$i	Input/output	^p	Phase
$r	Request	^h	backspace
$t	Trace	^z	pause
		$$$	noop

8 Switch Settings

Although switches are not (by default) shown, they remain as a medium for defining routings; users not acquainted with switches can think of them as points where data paths are channeled between PEs and where they can cross.

Purpose: To specify or modify a processor interconnection structure for the lattice.

Display: The current processor interconnection structure of (a portion of) the lattice for this phase is shown in the field; boxes represent processors, circles if shown represent switches, and lines represent bidirectional data paths. (Display of the switches is controlled by a set command; see Section 15.)

Cursor motion: Gross cursor motions advance the cursor to the next PE in the indicated direction; fine cursor motions advance the cursor to the next entity (PE or switch) in the indicated direction. "Home", from a switch causes the cursor to return to Last PE, from a PE causes it to go to the command line, and from the command line, to go to Last PE.

Activity: The cursor is moved around the lattice. If the draw mode is set, a wire is "pulled along" from the current position to the cursor's new position. If the remove mode is set, wires traced by the cursor are removed. At a switch all wires common to a level can be highlighted. If the chase mode is set, the cursor follows the wire in the direction indicated until it reaches a PE, or terminates, or reaches a switch that fans out, or cycles.

Generalization: The Pringle and Pringle emulator only support point-to-point communication, but in the Switch Settings view it is possible to define paths that fan out. (See Figure 6.) Communication structures with fanout cannot be emulated.

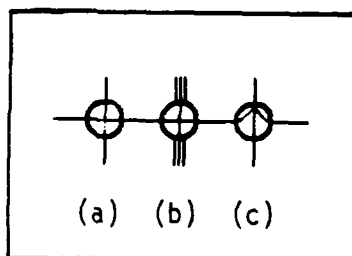


Figure 6. Switch (a) shows two distinct data paths crossing a switch, these paths are on separate levels; the north-south level of the same structure is highlighted in switch (b); switch (c) shows a data path fanning out, i.e. all paths are on the same level.

Recognized Keys:

Key	Name	Description
^c	Center	The cursor is moved to the PE whose index is given on the command line or if this is not visible, the display is changed so it is as close to the center of the field as possible, consistent with the requirement that the field remain fully utilized; if the command line is blank, the Last PE is used for centering.
^d	Draw	The mode is set to "draw" so that subsequent cursor motions cause a line to be drawn.
^r	Remove	The mode is set to "remove", so that subsequent cursor motions that trace a line cause it to be removed.
^g	Go for	Set chase mode, so that (only) the next cursor motion will follow the line in the indicated direction until it terminates, reaches a PE, reaches a switch that fans out or cycles.
^n	Null	End the current mode, i.e. cancel draw, remove or chase.
^l	Level	The level of the switch pointed to by the cursor is changed to the next level. Repeated use of this key cycles through all assigned levels and one unassigned level. The current level is highlighted.
$^k	Klear	Remove all switch settings for the current phase.
<key>		Keys, i.e. alphanumeric text, are placed on the command line.

Commands: The functionality of the above operations is augmented through the use of execute commands. The following are of particular interest for the Switch Settings view:

copy <from structure>, <to structure>
test paths
set switches

Refer to Sections 6 (for *copy*) and 15 for details.

Global Commands

$h	cHip	**$^a**	Abort	^f	File
$s	help	**$^e**	Exit	^v	+ page
$c	Code	**$^o**	Output screen	**$v**	- page
$p	Port	**$d**	reDraw screen	^u	eof
$i	Input/output	^p	Phase	**$u**	bof
$r	Request	^x	eXecute	**$$$**	noop
$t	Trace	^h	backspace	^z	pause

Interpretation: The line segments specified in Switch Settings mode are provided to create connections between PEs or between pads and PEs. However, some line segments may not participate in establishing a well formed connection: for example, two segments could meet at a switch without crossing it, or a sequence of segments might not connect to a PE. In the other views ***only legal connections are displayed***; the computation of these connections causes the pause when leaving Switch Settings for another view.

9 Code Names View

Purpose: To specify or modify the assignment of sequential processes (Poker C or XX) to the PEs or to specify actual parameters to the processes.

Display: The current code names and parameter assignments of (a portion of) the lattice for this phase are given in the field. One display format shows boxes representing the PEs; the other display format shows boxes representing the PEs and lines representing the interconnection structure; a key (^t) toggles between these two. A name of up to 16 characters, clipped to five characters, is shown for the program name, and four symbol strings of up to 16 characters, clipped to ten characters, are shown for the parameters:

```
1        1
  name-
param1----
param2----
param3----
param4----
```

Cursor motions: Gross cursor motions advance the cursor to the home position of the next PE in the indicated direction; fine cursor motions (north and south) move to the first position of the windows for the code name and the parameters. "Fine home," from a window moves the cursor to the home position of the PE; from the home position in a PE, "fine home" moves back to the last line referenced in a PE. "Gross home" from a PE moves the cursor to the command line, and from the command line, "gross home" moves to the home position of Last PE.

Activity: The names used in the sequential process codes (following the word *code*) and (actual) parameter values are entered into the appropriate positions. (See Section 12.) Code names can be any legal identifier of the sequential programming language not containing blanks, and parameters can be any legal constant of the sequential programming language.

Buffering: The code name and parameters of a PE can be saved in a buffer (using ^b) that is then displayed in the chalkboard. The PE to be saved is the

PE containing the cursor, or if the cursor is on the command line, the <i><j> given on the command line, or if it is blank, the Last PE. The saved values are deposited into one or more PEs by specifying recipient PEs followed by a deposit (^d) command. Recipient PEs are specified in one of two ways, either explicitly, by giving an index pair (i j), or implicitly. The implicit specifications uses an expression where each index position is either an index, a relation (<, <=, >, >=) followed by an index, meaning all indices standing in that relationship to the index, or a period (.), meaning all index values. Thus, a command

. 2

followed by ^d causes all PEs in the second column to receive the buffered values.

Recognized keys:

^b	Buffer	The code name and parameters of the PE containing the cursor are saved and displayed in the chalkboard. Modification to any of the entries of the buffered PE cause it to be removed from the buffer. The buffered PE remains buffered even if the chalkboard is overwritten.
^d	Deposit	Insert the buffered names into the recipient PE(s). If the command line is blank, the recipient is the PE containing the cursor; if the command line is nonblank the recipient is given by the command line expression as described in Buffering above.
^r	Remove	Delete the code names and parameters in the indicated PEs. If the command line is blank clear the PE containing the cursor; if the command line is nonblank the cleared PEs are given by the command line expression as described in Buffering above.
^c	Center	The cursor is moved to the PE whose index is given on the command line or if this is not visible, the display is changed so that the PE is as close to the center of the field as possible consistent with the requirement that the field be fully utilized; if the command line is blank use the Last PE for centering.

^t	Toggle	The display is changed to the "other" format as described in Display above.
^y	displaY	The full (unclipped) entry of the window containing the cursor is shown in the chalkboard.
$^k	Klear	Remove all code names and parameters entries for the current phase.
<key>		If the cursor is in the window, the symbol replaces the symbol pointed to by the cursor; if the cursor is at the home position of a PE or on the command line, the symbol appears on the command line.

Commands: The functionality of the above operations is augmented through the use of execute commands. The following are of particular interest for the Code Names view:

copy <from structure>, <to structure>
replace <old> <new>

Refer to Sections 6 (for *copy*) and 15 for details.

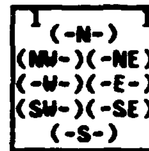
Global Commands

$h	cHip	$^a	Abort	^f	File
$s	Switch	$^e	Exit	^v	+ page
$c	help	$^o	Output screen	$v	- page
$p	Port	$d	reDraw screen	^u	eof
$i	Input/output	^p	Phase	$u	bof
$r	Request	^x	eXecute	$$$	noop
$t	Trace	^h	backspace	^z	pause

10 Port Names View

Purpose: To specify or modify the names assigned to the eight input/output ports of a PE.

Display: The current port names of (a portion of) the lattice for this phase are shown in the field. The display format shows an array of boxes representing the PEs; the other display format shows boxes representing the PEs and lines representing the interconnection structure; a key (^t) toggles between the two. Names of up to 16 characters, clipped to the first five characters, are shown in the PE boxes:



Cursor Motion: Gross cursor motions advance the cursor to the home position of the next PE in the indicated direction; fine cursor motions move the cursor to the first position in the window for the port name for that direction. A "fine home", from a port window moves to the home position of this PE; from the home position in a PE "fine home" moves back to the last window referenced in a PE. "Gross home" from a PE moves to the command line; from the command line "gross home" moves to the home position of Last PE.

Activity: Port names are entered into the appropriate windows to name the ports connecting to the incident data paths. Port names can be any legal identifier of the sequential programming language not containing blanks.

Buffering: The port names of any PE can be saved in a buffer (using ^b) that is then displayed in the chalkboard. The PE to be saved is the PE containing the cursor, or if the cursor is on the command line, the <i><j> given the command line, or if it is blank, the Last PE. The saved port names can be deposited into one or more PEs by specifying recipient PE(s) on the command line followed by a deposit (^d) command. Recipient PE(s) are

specified in one of two ways, either explicitly, by an index pair (i j), or implicitly. The implicit specification uses an expression where each index position is either an index, a relation (<, <=, >,>=) followed by an index, meaning all indices standing in that relation to the index, or a period (.), meaning all index values. Thus a command

. <= 4

followed by ^d causes the first four columns to receive the saved port names.

Recognized keys:

Key	Name	Description
^b	Buffer	The port names of the PE containing the cursor are saved and displayed in the chalkboard. Modification of the port names of a buffered PE cause it to be removed from the buffer. A buffered PE remains buffered even if the chalkboard is overwritten.
^d	Deposit	The buffered names are placed into the recipient PE(s). If the command line is blank, the recipient is the PE containing the cursor; if the command line is nonblank the recipient(s) are given by a command line expression as described in Buffering above.
^r	Remove	Clear all port names in the specified PE(s). If the command line is blank the cleared PE is the PE containing the cursor, if the command line is nonblank, the cleared PE(s) are given by the command line expression as described in Buffering above.
^c	Center	The cursor is moved to the PE whose index is given on the command line or if this is not visable, the display is changed so that the PE is as close to the center of the field as possible, consistent with the requirement that the field remain fully utilized; if the command line is blank, the Last PE is used for centering.
^t	Toggle	The display is changed to be "in the other" format; see Display above.
^y	displaY	The full (unclipped) entry of the window containing the cursor is given in the chalkboard.
$^k	Klear	Remove all port name entries for the current phase.
<key>		If the cursor is in a window, the symbol replaces the symbol pointed to by the cursor; if the cursor is at the home position of a PE or on the command line, the symbol appears on the command line.

Commands: The functionality of the above operations is augmented through the use of execute commands. The following are of particular interest for the Port Names view:

copy <from structure>, <to structure>
replace <old> <new>
test ports

Refer to Sections 6 (for *copy*) and 15 for details.

Global Commands

$h	cHip	$^a	Abort	^f	File
$s	Switch	$^e	Exit	^v	+ page
$c	Code	$^o	Output screen	$v	- page
$p	help	$d	reDraw screen	^u	eof
$i	Input/output	^p	Phase	$u	bof
$r	Request	^x	eXecute	$$$	noop
$t	Trace	^h	backspace	^z	pause

11 IO Names View

Purpose: To specify or modify the stream names assigned to the lattice's input/output pads. The "input/output pads" are the points at which wires, given in the switch settings view, extend "off" the edge of the lattice. *Data streams*, sequences of data values, can be read and written through the input/output pads. A file, named <name>, composed of records with k fields is interpreted as a collection of k streams laid "side-by-side". The individual streams are referred to by their <name> and an index which is the field number. For example, <name> 3 is a stream whose first element is the third field of the first record of file <name>. Note, streams are unidirectional.

Display: The field is divided in half. The upper half lists the input/output pads, clockwise beginning from the northwest corner, with the associated stream name, stream index, direction (input or output), as well as the port direction, port name, code name and index of the connected PE. The lower half gives a schematic diagram of the lattice with an arrow pointing to that pad whose window, in the upper half, contains the cursor.

Cursor motions: North/south cursor motions move between pad windows: Fine motions move a single window; gross motions move six windows. East/west (gross or fine) cursor motions move between the name and the index panes. "Home", from a pad window moves to the command line, and from the command line returns to the last pad window.

Activity: The names, indexes and directions (input or output) are entered into the appropriate windows.

Buffering: The stream name, index and direction of a pad can be saved in a buffer (using ^b) and displayed in the chalkboard. The saved values, appropriately modified, are deposited into one or more pad windows by specifying the recipient pads followed by a deposit (^d) command. Saved values are modified by incrementing the saved index value by one prior to each deposit. The recipient pad is specified explicitly by giving one, two or three values on the command line. A single value specifies the pad number of the recipient, two values specify the start and end of a range of recipient pads; and three values specify the start, end (inclusive) and step size of a subrange of recipient pads. Thus the command

4 1 -1

followed by ˆd assigns the first four pads the buffered stream, index and direction information such that the indexes are decreasing, i.e. the lowest index is assigned to pad 4, the next lowest to pad 3, etc.

Recognized keys:

ˆb	Buffer	The stream name, stream index and direction of the window containing the cursor are saved and displayed in the chalkboard.
ˆc	Center	The pad window whose number is specified on the command line is placed at the center of the upper half of the field, if the command line is blank, the window containing the cursor is centered.
ˆd	Deposit	If the command line is blank, the pad window containing the cursor receives the buffered information after the stream index has been incremented, a nonblank command line is treated as a range specification (as described in Buffering above) and the saved information is deposited into each of the specified pads with an incremented stream index.
ˆi	Input	The pad whose window contains the cursor is designated as an input pad.
ˆo	Output	The pad whose window contains the cursor is designated as an output pad.
ˆr	Remove	If the command line is blank the pad window containing the cursor is cleared, a nonblank command line is treated as a range specification and the specified pad windows are cleared.
$ˆk	Klear	Remove all stream names, indices and directions for the current phase.
<key>		The key replaces the symbol pointed to by the cursor.

Commands: The functionality of the above operations is augmented through the use of execute commands. The following are of particular interest for the IO Names view:

bind <stream> <file>
copy <from structure>, <to structure>
display bind
replace <old> <new>
test pads
unbind <stream>

Refer to Sections 6 (for *copy*) and 15 for details. In addition, facilities for formatting files, such as packIO, are described in Appendix B.

Global Commands

$h	cHip	$^a	Abort	^f	File
$s	Switch	$^e	Exit	^v	+ page
$c	Code	$^o	Output screen	$v	- page
$p	Port	$d	reDraw screen	^u	eof
$i	help	^p	Phase	$u	bof
$r	Request	^x	eXecute	$$$	noop
$t	Trace	^h	backspace	^z	pause

12 Process Definition Languages

A Poker programmer must define sequential processes to run on the PEs using an appropriate sequential language and a standard editor. (This activity is usually performed in the secondary terminal.) In principle most sequential languages could be enhanced to be "appropriate", but Poker presently supports only Poker C and XX. Sequential processes can be defined in either language, but only one language can (at present) be used for a single Poker program. Although both will be supported for the foreseeable future, users not interested in using the Pringle parallel computer may prefer to use Poker C.

12.1 The Poker c2c Compiler.

12.1.1 Introduction.

The Poker C compiler, **c2c**, is a source to source compiler that translates a Poker C program *into an, often much expanded,* C *program.* The resultant code is compiled either using the standard C compiler so that it can link into the Poker Native Code run-time system which simulates Poker programs, or with other C compilers to create code to execute on parallel computers such as CalTech's Cosmic Cube[1].

Briefly, Poker C is regular[2] C with the following extras:

- More types and some restrictions on the use of variables,
- Tracing,
- Port I/O (<-),
- Inter-process and inter-phase structure definitions,
- An inter-phase variable space, and
- More syntax.

Poker C disallows the following features of standard C:

- `Extern` declarations.
- `Static` declarations.
- Variables external to routines.

The rest of this section describes the syntax and semantics of Poker C, features unique to Poker C, tracing in Poker C, and finally a profiler which is part of the Poker Generic simulator and provides timings for Poker programs according to a timing model tailorable to different architectures.

[1] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[2] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Academic Press, New York, 1978

12.1.2 Using The c2c Compiler.

In general you will never have to use c2c directly; Poker will call c2c when needed. You can use the same sequence of commands to make and execute a Poker C Poker program as a XX Poker program.

- *pkcc.* The most common use of c2c will be via the Poker c compiler, pkcc. Pkcc works in the same way as the normal UNIX cc command, controlling the flow of commands needed to compile the source code into the desired target code. In addition pkcc:
  - compiles `.x` files into `.x.pc` files,
  - compiles `.pc` files into `.pc.o` files,
  - and has a number of extra flags, particularly the `-T` profiler option (see below).

- *Flags.* The c2c compiler has a number of flags which may occur in any order and with one or more flags per '-'. Two flags are used with the timing model (see Section 12.1.11):

  `-T` — do not attribute the code with calls to `USERTICKS()`. The default is to attribute (see Section 12.1.7).

  `-T#` — attribute the code with calls to `USERTICKS()`, according to the `#`:
  - `0` — minimal; one tick at the beginning of each loop, routine, etc.,
  - `1` — one tick per statement,
  - `2` — full profiling according to the costs in the `.poker-time-model` file.

  All other flags aid in debugging the compiler. As such, the explanation will not be very useful if you do not know about the internals of the compiler:

  `-b` — Print the trees created by BuildTree during parsing.

  `-e` — If there is a parse error, try to print as much of the output code as possible before dying.

  `-l` — Print the value of each token as returned by the lexical analyzer. Note, the lexer is concerned only with tokenizing the input stream; it does not access any parser data structures, including the symbol table.

  `-s` — Print the symbol table on `stdout` at various points during a compilation.

  `-t` — Print the value of the trace list and trace messages during parsing.

  `-y` — Verbose yacc output.

- *Pre-processing.* The c2c compiler supports the macro pre-processor, cpp, in two ways:

    1. C2c correctly consumes the `#` lines created by cpp.
    2. A line that starts with `%%` is special. The compiler replaces the `%%` with a `#` and passes through the rest of the line verbatim. These `%%` lines may occur only before the "`code` *code-name*", before, after, or between routines, and between statements. This may be useful for passing a "`#define`" through the c2c compiler and into the cpp run by the normal cc compiler.

- *Parameters.* C2c may take up to 2 file names as command line arguments. The first is the input file, the second the output file. Input defaults to `stdin`, output to `stdout`.

- *Poker C files.* Poker C file names must terminate with the "`.pc`" suffix.

12.1.3 Port I/O.

The port I/O statements are

lvalue <- *port*
port <- *expression*

for reading from and writing to ports, where

- *expression* must be of a type in the inter-process/inter-phase type space (see Section 12.1.6). The expression on the lhs of a <- must be an *lvalue*[3].

- *port* may be either a port name from the `ports` list, or a variable of type `port`.

- The port I/O operator, <-, is a binary operator which is a statement; it is not an expression, and cannot be included in other expressions.

Inter-process communication lines are not typed; they may pass any of the legal types in any order. However, each datum is typed, so that the sending and receiving process must be using compatible types.

[3]An expression referring to a manipulatable region of storage.

12.1.4 Data objects

There are no global data objects in Poker C. The inter-phase data space is private to a sequence of process codes living in the same process space, one process code per phase. Within a phase, all of a process' data is internal to some routine. This includes the parameters to the `main` routine; if another routine in that process code wants to access the value of a parameter to `main`, you have to ship it explicitly to that routine.

The compiler implicitly passes port names declared in the **ports** declaration to every routine in the process code.

The c2c compiler does *not* implicitly pass port names to routines declared as `library`.

12.1.5 The inter-phase variable space.

Processes communicate across phases through the use of an inter-phase variable space. The variables in this space live until the end of the entire Poker program; they may exist before and after the process codes that use them. See Section 12.1.6 for the legal types of inter-phase variables.

Access to the inter-phase variables occurs via only one of two expressions:

import(local, inter-phase)
export(local, inter-phase)

which copy the value of the inter-phase variable, respectively, into or out of the local variable. The `local` variable may be any legal lvalue[4] that does not contain a pointer, including entire arrays and structures, as described in Section 12.1.6 and the inter-phase variable may be any identifier other than a keyword. Both routines return TRUE upon success, and FALSE if an error occurred. Errors include (1) importing a variable that has not been exported, (2) importing a variable to a local of a non-compatible type, and (3) exporting to an inter-phase variable already containing a value of a noncompatible type. In all cases, the copy is aborted leaving the local value unchanged.

In essence, the inter-phase variables live in a separate address space from the process codes that access them.[5]

C2c stores the unique identifiers associated with structures in the **.poker_structures** file. While we allow for a huge number of inter-phase structures, if the compiler ever complains that it is running out of unique structure numbers it will print a message with instructions on how to relieve the problem.

[4] An expression referring to a manipulatable region of storage.

[5] However, the implementor is not actually required to use separate address spaces as long as access to the variables is restricted correctly.

```
normal              #include "error_codes.h"
cpp                 #define MAX_SIZE 100
support
                    code head;                         /* name of code
                    trace #;                           trace line #'s
header              ports in;                          port names
                    trace read().i, read().temp;       list of variables
                                                       to trace
                    library reduce();                  routines defined external
                                                       to this file ports not
                                                       implicitly passed.
                    main(size)                         parameter from Code Names
                                                       View
                        int size;
                        {
routines                    float startData[MAX_SIZE];
no                          read(startData, size);     Code Names parameters
external variables                                     are local to main().
                            reduce(startData);
                            export(startData, "Data"); save in inter-phase
                                                       data space
                        }
                -   read(data, size)
                            float data[]; int size;
                        {
                            int i;
                            float temp;                use temp to trace the
                                                       values as they come in
                            for (i=0; i < size; i++)
                                {
                                    temp ⟵ in;         ports defined in header
                                    data[i]=temp;      implicitly passed to all
                                }
routines in this file.
                        }
```

Figure 7
Annotated Example Poker C Program

Figure 7 shows an annotated example of a Poker C program. The header defines the local environment of this code – its name, symbolic names for the ports used in Port Names View, variables and other entities to trace, and which routines are defined external to this file. In general, all routines for a given process are defined in a single file. However, common routines, such as Sin(), may be declared in library routines.

Following the header is a list of routines. main() is the starting place for execution of the code, and can alternatively have the same name as the code (head(), in this case). The Code Name parameters are listed after the computation as variables local to main (). All routines in this file implicitly know about the ports defined in the header.

12.1.6 Types In Poker C.

Poker C was designed to retain the flavor of C while providing enough stability to support a structured view of parallel programs. In this light, we extended the C types to include:

- *Type* `bool`. Variables of type `bool` are boolean variables with value `TRUE` (-1) or `FALSE` (0). Note that these truth values are different from those defined for C; this allows us to have `(!TRUE == FALSE)` and `(!FALSE == TRUE)`.

- *Type* `port`. Variables of type `port` may be used in place of a `port` name. A `port` variable may occur on the lhs of an assignment if and only if the rhs is either a `port` name or a `port` variable. `Port` variables are not automatically initialized to a particular value.

- *Inter-phase/inter-process data types.* (See Section 12.1.5.) In order to pass data safely between processes and phases, we strictly define types across processes and phases. Poker C allows the following types of data to be passed between processes and phases:

  - *Fundamental types:* (*c.f.* Kernighan & Ritchie for the definitions of the flavors of types.) Values of these types may freely convert among themselves as specified for the C language.
  - *Arrays:* The type of an array is a pair: (type of base element, size). Only entire arrays of statically declared size may be passed between processes or phases.
  - *Structures:* Structures sent on the I/O system or between phases may not contain pointers or unions. All other structure automatically becomes part of the inter-phase/inter-process structure space.

    Inter-phase/inter-process structures are typed by the order, number, and types of their sub-types. Variable names are unimportant to this typing scheme. Thus, in:

```
struct foo                  struct tee
   {                           {
   int i, j;                   float fee;
   struct                      };
      {                     struct bar
      float f;                 {
      }                        int lee;
   } fooVar;                   int ree;
                               struct tee see;
                               } barVar;
```

structures `foo` and `bar` have the same type, as do the un-named internal structure in `foo` and the structure `tee`. Note that the expression "`fooVar = barVar`" is allowed by `c2c`, but not by `cc`, and thus is not a legal construction Poker C. In other words, Poker C types structures by their form, for inter-phase/inter-process communication, and by name, for intra-process assignment.

Inter-phase/inter-process structures may contain arrays and *vice versa*, as long as neither contains any pointers.

The implementor of the run-time system is free, in fact encouraged, to send structures and arrays in the most efficient way possible. This could mean packaging the array or structure into a few large packages.

The external file system does *not* (currently) support structures or arrays. The concept of an inter-process/inter-phase type space is for use inside the Poker program.

12.1.7 Special Routines in Poker C.

Poker C uses routine calls and a few assignments to global variables to interface to the run-time system. All of these global variables and most of these routine names are preceded by an underscore (_) and are meant to be private between the c2c compiler and the run-time system.[6]

Other interface routines are part of Poker C; we describe them here.

`int USERTICKS(`*expr*`)` — This routine takes a single argument, an integer valued expression, indicating the number of ticks by which to bump the process' clock. To assign specific costs to sections of your program, insert `USERTICK()` calls. This routine returns an int of undefined value.

[6]The industrious and inquisitive may root through the `.c` files generated by the c2c compiler to see the routine calls, but be warned that these routines are not intended for human-use. See also the documentation for the generic simulator.

A more general method of attributing your program with costs is described under the Poker Profiler, Section 12.1.10.

`bool DataAvail(`*port*`)` — This routine returns **TRUE** if there is data currently waiting to be read from port *port*, and **FALSE** otherwise. If returns FALSE if there is an EOS.

`int CheckType(`*port*`)` — This routine returns an integer which corresponds to the type of the awaiting value:

Integer	Type
-1	*no data available*
0	EOS
1	bool
2	char
3	short
4	int
5	long
6	float
7	double
>7	*structure or array*

`bool IsEOS(`*port*`)` — This routine returns **TRUE** if there is an **EOS** on the input port *port*, and **FALSE** otherwise.

`ClearEOS(`*port*`)` — This routine clears the **EOS** from the port *port*. If there is no **EOS** to clear. you will get a fatal run-time error. `ClearEOS()` is the only way to get rid of a waiting **EOS** since attempting to read from a port containing **EOS** results in a fatal run-time error.

`ExitPE(`*error-code, errorMessage*`)` — This routine terminates the execution, for this phase, of the calling PE. If the error-code is zero, termination is normal. Otherwise, the run-time system prints an error message with the integer *error-code* and terminates the PE.

12.1.8 Tracing variables and state in Poker C.

Poker provides help in debugging a program in that you can "trace" the values of variables and observe the values in Poker's Trace View. This section describes which variables may be traced, how to request a trace, and how to specify the granularity of the trace.

Form	Meaning
`foo()`	trace `foo`'s entry and exit
`foo()!`	exhaustively check for changes to trace variables in routine `foo`
`foo():`	trace `foo`'s entry/exit & labels
`foo()#`	trace `foo`'s entry/exit & line #'s
`foo():#`	trace `foo`'s entry/exit & line #'s, & labels
`foo()#:`	trace `foo`'s entry/exit & line #'s, & labels
`foo().name`	check for any change to variable `name` in routine `foo`
`foo().name[]!`	exhaustively check for change to any traced element of `name`
`name`	same as "`main().name`"

Table 1: Traceables. Omitting "`foo`"causes the first six rules to apply to all routines.

- *Legal Trace Variables.* Trace variables may be:

Type	Comments
Variables of fundamental type	`char`, `unsigned long`, `double`, ...
Variables of type `bool`	boolean: `TRUE` or `FALSE`
Pointers	
Line numbers	
Routine entry/exit/recursion-depth	
Labels	

  More complex variables, such as arrays and structures, may be traced, but only by tracing the individual elements of the array or structure.

- *Requesting a trace.* The trace statement in the header of a Poker C program lists the variables and other items to trace. Table 1 shows the form and variety of the traceable variables in Poker C. Section 12.1.15 shows the precise form of all trace variables.

  The variable name, `name`, in a trace list may either be a single identifier or a dot sequence denoting a member of a structure, *e.g.* `structName.field1.field2`.

  If the trace variable is the location pointed to by a pointer, `*ptr`, changes to either `ptr` or `*ptr` update the trace value of `*ptr`. This cascading of updates can be arbitrarily deep.

- *Exhaustive checking.* In deference to execution efficiency, Poker does not attempt to trace aliases; the compiler recognizes only literal instances of a trace variable. However, Poker C does have a mechanism to insert run-time code that will exhaustively check all assignments for changes to trace variables. There are three granularities of exhaustive checking, each requested by appending an exclamation point to the range of data to check.

- *in an array, e.g.* "foobar[]!". The compiler inserts a run-time check after each assignment to an element of that array, recognized by a literal instance of the array name, to check to see if the values of the indicies correspond to a traced element of that array. This level of tracing will *not* detect that *(pa+1), where **pa** points to **a[0]**, is a reference to **pa[1]**.
- *in a routine name, e.g.* "foo()!". The compiler inserts a run-time check after each assignment in that routine, to check for a change in the value of a trace variable. This includes array assignments in the routine.
- *in all routines,* "()!". The compiler inserts a run-time check after every assignment in the entire program. This is equivalent to requesting an exhaustive check for each routine in the entire Poker C program.

12.1.9 Other Notes.

C2c attempts to mimic the cc compiler's error messages and type checking. Thus, error and warning messages have the file name and line number in the format used by the emacs "next-error" command, and **c2c** will catch most of the syntax and type errors found by cc.

In the cases where **c2c** does not catch an error, the line numbers reported by the cc compiler, as it compiles the output of the **.pc** file, may only approximate the location of the error in the **.pc** file.

Typedef declarations must have **typedef** as the first word in the declaration.

To declare a function as returning a pointer to an array, you must first declare a type, using typedef, corresponding to the array and then declare the function to return a pointer to this type. This fix is needed to work around the differences between arrays in Poker C and normal C.

For the same reasons Poker C formal parameters declared to be arrays are treated as arrays; not as pointers.

Unnamed structures are implicitly named by their unique structure identification number as defined in **.poker_structures**. For example, the unnamed structure with unique id 12 is named **_12**.

C2c makes copious use of identifiers starting with a single underscore (_). Poker C programs should avoid this naming style.

A function that is not defined in your Poker C program or the libraries it includes may cause problems.

The cosmic cube does not like variables declared as '**int's**' due to compatibility problems. The C2c compiler convers '**int's**' to '**long's** automatically and issues a warning.

In a modest departure from C, the parser expects all compound assignment operators to be a single token; they may not have white space in them. In other words, "+=" is legal, but "+ =" is not.

12.1.10 The Poker Profiler.

The Poker C generic simulator allows the user to time the execution of Poker programs. To do this timing, the user first has to set up a time model specifying the costs for the basic C operations as they would run on the user's idealized machine. The Poker compiler uses this file to attribute the Poker C program with calls to the `USERTICKS()` routine. When the code executes, these `USERTICKS()` calls bump the process' clocks, thereby providing a dynamic execution time for the process code.

12.1.11 The Machine Model.

The Poker time model specifies the cost at the level of the basic C operations according to the type(s) of their operand(s). The specification file specifies costs with lines of the following forms:

Form	Use	Example
type operator type cost	binary operators	`int *= float 10`
operator type cost	unary operators	`* int 1`
operator cost	indexing operators (., ->, [])	`. 1`
routine cost	calls to routines	`foo 150`
`port <-` *constant per-byte wait*	port write costs	`port <- 100 1 TRUE`
`<- port` *constant per-byte wait*	port read costs	`<- port 100 1 FALSE`
`<-` *constant per-byte*	port transmit costs	`<- 10 1`
`return` $cost_1$ $cost_2$	cost of routine entry $_1$ & exit $_2$	
`import` *per-byte*	import cost	
`export` *per-byte*	export cost	
`struct` *per-byte*	structure assignment cost	
keyword cost	cost for control flow	`if 2`

where the terms are as follows:

- *type*. The types are one of the fundamental types of C: `char`, `short`, `int`, `long`, `float`, and `double`. We assume that operations cost the same for both unsigned and signed types.
- *operator*. The operators are the standard operators of C, including the primary-expression operators: `()`, `[]`, `.`, and `->`. Here, `()` indicates the cost of implicit coercion from the lhs type to the rhs type and may be read as "coerced to". The other primary-expression operators indicate indexing into their data objects.
- *cost*. All costs are positive integer values.

- *routine.* There are two ways to indicate the cost of library routines, such as `sin()`. The first is to run the library source through the `c2c` compiler and attribute its code with `USERTICKS()`, in the same manner as for user programs, and then use this attributed code for the library routine. This will give as accurate a timing as possible under our timing model.

  A second, simpler but less accurate, method is to assign a constant cost to the routine. This is done using the fourth construct in the table. For instance, including a line of "`foo 1000`" in the file describing the Poker time model, will charge 1000 ticks for each call to the routine `foo`.

- `port <-` *constant per-byte wait.* The cost of receiving a message, assuming instantaneous transmission, where
  - *constant* is the constant cost (overhead) of a read.
  - *per-byte* is the extra cost accounting for the increased cost of handling longer messages. For example, it could be the cost of packaging the data into the correct message buffer.
  - *wait* defines whether the process has to wait for the data transfer to complete before incurring the *constant* read cost (`TRUE`) or whether it may transfer data at the same time as it incurs the *constant* read cost (`FALSE`). In either case the reader cannot continue computation until the data transfer is complete; reads are blocking.

- `<- port` *constant per-byte wait.* The cost of sending a message, assuming instantaneous transmission, where
  - *constant* is the same as for read.
  - *per-byte* is the same as for read.
  - *wait* defines whether the process has to wait for the data transfer to complete before resuming computation (`TRUE`) or whether it may transfer data and compute at the same time (`FALSE`). In either case the writer must incur its *constant* write cost before starting the data transfer.

- `<-` *constant per-byte.* The cost of transferring a message between sender and receiver, assuming instantaneous setup, where
  - *constant* is the same as for read.
  - *per-byte* is the same as for read.

- **return**. The first cost is that of entering or calling a routine. The second cost is for exiting a routine.
- **import**, **export**. These are the per-byte costs for copying the inter-phase value into the local variable.
- **struct**. Poker C supports structure assignment (since the cc compiler supports it; the target compiler determines whether structure assignment is allowed) and we assume that all structures incur the same per-byte copy cost.
- *keyword*. This is the cost of control flow. *Keyword* may be on of the following constructs: **break**, **continue**, **do**, **for**, **goto**, **if**, **switch**, **while**, **?**.

Although c2c does not have a default time model, Poker has two available as "standard." **default-model**, located in /user/poker/lib, is a minimal model assuming 1 tick for all integer operations, 10 ticks for all floating point operations, port read/write using 1000 ticks for the constant part, 1 tick per byte, and no I/O co-processor, and 100 ticks to set up the transfer which consumes 10 ticks per byte. Part of **default-model** is shown in Figure 8.

pringle-model is a time model for the Pringle Parallel Computer[7]. Within the time model there are no explicit defaults; it does have rules to implicitly define costs in a brief manner. When the profiler looks up a cost, it first looks at the operator using the type(s) of the expression. If the cost is specified, fine. If not, the profiler continues looking in the table by first incrementally bumping the type of the lowest type toward the higher type in the order: **char** → **short** → **int** → **long** → **float** → **double**. If their types are the same it next bumps along the diagonal toward **double**. In this way, the minimal model may define just one cost per operation, the operation on operand(s) of type double, and have all other operands cost as much: specifying only "**double + double 100**" for addition implies that every addition will cost 100 ticks.

The port I/O time model assumes asynchronous writes to an infinite length hardware FIFO buffer at the destination process. (Of course, the buffers are actually finite; see **set** command in Section 15.) One or more writers can concurrently write into the buffer of the reader without consuming any cycles from the reader. Writes may be blocking or non-blocking, according to the boolean flag in the cost specification. Reads are blocking.

The writer incurs its constant and per-byte write costs before starting the transfer. If write wait is **FALSE**, the writer may proceed with computation as soon as it is done with its write costs; otherwise it waits for the transfer to complete before continuing with computation.

[7] Alejandro Kapauan, Ko-Tang Wang, Dennis Gannon, Janice Cuny, and Lawrence Snyder. The Pringle: an experimental system for parallel algorithm and software testing. *Proceedings of the International Conference on Parallel Processing, IEEE*, 1–6, 1984

```
[] 0
. 0
-> 0

port <- 1000 1 TRUE
<- port 1000 1 TRUE
<-  100 10
return 10 10

import 1
export 1
struct 1

break 0
continue 0
do 0
for 0
goto 0
if 0
switch 0
while 0
? 0

char +  char 1
short +  short 1
int +  int 1
long +  long 1
float +  float 10
double +  double 10

* char 1
* short 1
* int 1
* long 1
* float 10
* double 10
```

Figure 8.
Beginning of default time model in file `default-model` of the distribution tape.

The reader may not proceed with computation until after the transfer is complete. However, if read wait is **FALSE**, the reader incurs the constant and per-byte read costs during the transfer. In this way, the reader may proceed immediately with computation, assuming that there was enough overlap to hide all of the read cost during the transfer.

Figure 9 shows the effect of varying the boolean parameters for a scenario of two sends and receives. The leftmost time line has read wait and write wait set to **TRUE**, the second time line sets write wait to **FALSE**, and the third sets both to **FALSE**. These boolean parameters can be thought of as specifying the absence of I/O co-processors, though a more realistic model of I/O co-processors would also change the transfer times. The settings are written below each time line in the syntax of the time model files. Question marks indicate arbitrary natural numbers.

12.1.12 Profiler Files.

The compiler uses the time-model in the **.poker-time-model** file in the current directory. Poker automatically copies the time model you specified to **.poker-time-model**.

If you wish to see a tabular representation of the time model, run the program **profile** with up to two arguments; this will generate a file you may peruse.

profile — uses the time model in **.poker-time-model** and leaves the table in the file **.poker-time-model-table**.

profile infile — uses **infile** as the time model and leaves the table in the file **infile-table**.

profile infile outfile — uses **infile** as the time model and leaves the table in the file **outfile**.

12.1.13 Profiler operation.

The c2c compiler inserts profiling costs into Poker C code unless invoked with the -T option. (See Section 12.1.2.)

12.1.14 Interesting Problems

While the timing model provides a good deal of generality, allowing the user to test the algorithm's performance on different architectures, there are a number of limits to its accuracy. First and foremost, the compiler inserts costs according to the frequency of the operations in the source code, not in the assembly language that would execute on the target architecture. Different assembly languages and optimizing compilers will affect the accuracy of this representation. Second, the architectural model and the I/O model are fairly simple and cannot reflect the entire range or specificity of actual architectures. For instance, the I/O model either has no I/O co-processors or one I/O co-processor per process. This scheme

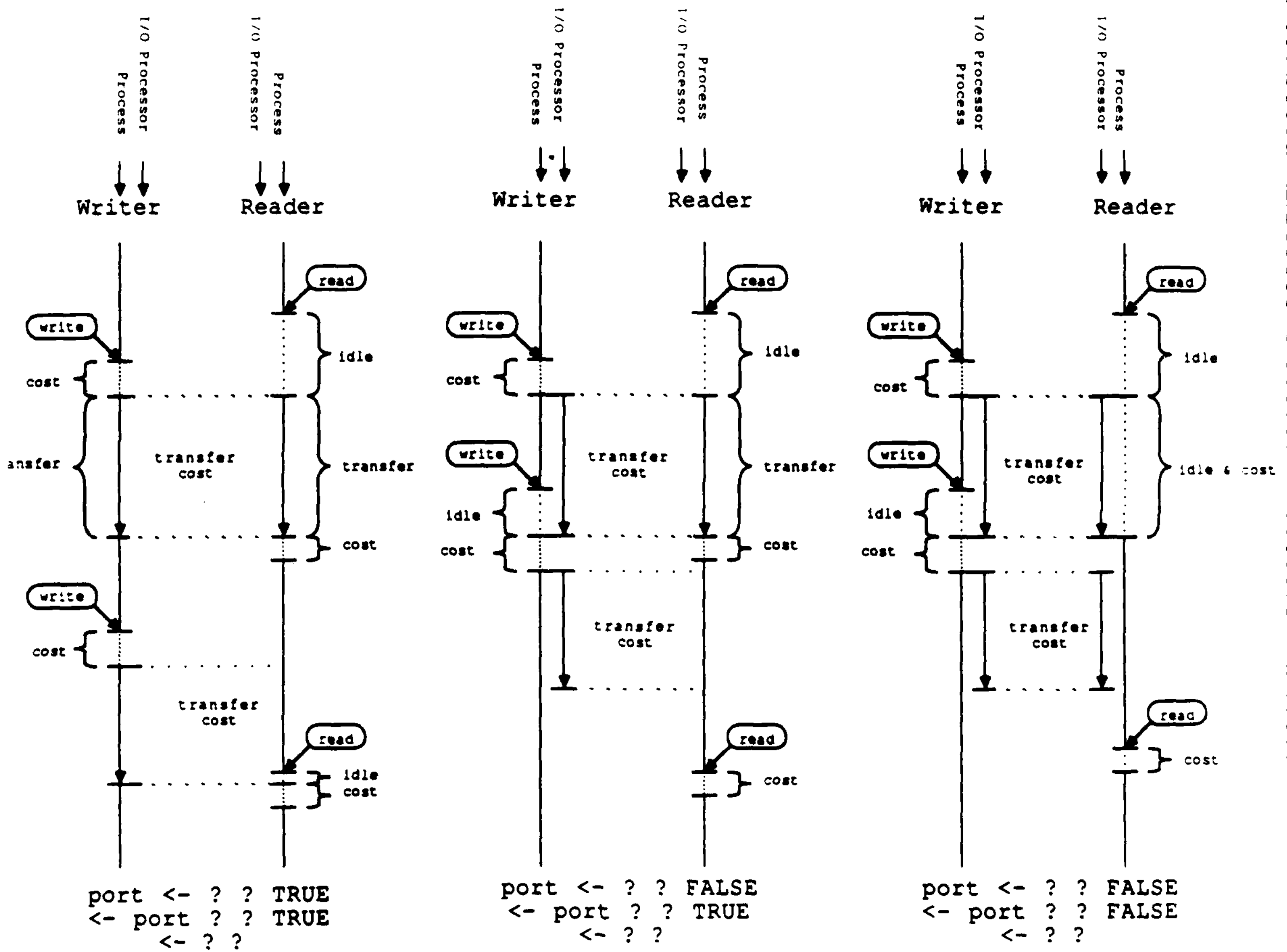


Figure 9
Poker time model showing effect of varying port I/O parameters.

does not deal with several processes on a single processor and the sharing of any available co-processor.

However, the model is easy to change and the generic simulator produces dynamic timings making possible some architectural experiments.

12.1.15 Poker C Syntax.

The following table describes the syntax for a Poker C file. Words in *italics* are non-terminals, while words in `typewriter` font are terminals. λ is a special symbol indication an empty string. The non-terminals in [boxes] correspond to the non-terminals in normal C.

program =
  optional-%%-lines
  `code` [*id*] ;
  poker-header
  external-definition-list
  | *optional-%%-lines*
  `library` [*id*] ;
  library-list
  external-definition-list
poker-header =
  poker-header trace-list
  | *poker-header ports-list*
  | *poker-header library-def*
  | λ
trace-list =
  `trace` *trace-name-list* ;
trace-name-list =
  trace-name-list , *trace-name-declaration*
  | *trace-name-declaration*
trace-name-declaration =
  | [*id*] () *trace-modifier* – *per routine*
  | [*id*] (). *trace-variable* – *per routine*
  | [*id*] (). *trace-variable* [] ! – *exhaustive array per routine*
  () *trace-modifier* – *global*
  | *trace-variable* – *in main*
  | *trace-variable* [] ! – *exhaustive array in main*
trace-modifier = – *up to one of each of ...*
  !
  | #

| :
| :#
| #:
| λ

trace-variable =
* *trace-variable*
| *trace-variable* . [*id*]
| *trace-variable* -> [*id*]
| *trace-variable* [[*constant-expression*]]
| (*trace-variable*)
| [*id*]

ports-list =
ports *name-list* ;

name-list =
name-list , [*id*]
| [*id*]

library-list =
library [*type*] *routine-name-list*;
| **library** *routine-name-list*;

routine-name-list =
routine-name-list , *routine-name*
| *routine name*

routine-name =
* *routine-name*
| [*id*] ()

external-definition-list =
external-definition-list external-definition
| λ

external-definition =
[*typedef*] *type declarator* ;
| *optional-%%-lines*
| [*structure-declaration*] ;
| [*enumeration-declaration*] ;
| [*function-declaration*] [*function-body*]

statement =
the normal C statements
| *optional-%%-lines*

expression =

the normal C expressions
| *port-expression* `<-` `EOS`
| [*lvalue*] `<-` *port-expression*
| *port-expression* `<-` *expression*
| `import` (*expression*, [*id*])
| `export` (*expression*, [*id*])

A *port-expression* is either a port name, declared in the **ports** declaration of the header, or a variable of type **port**.

Execution of a process starts in the routine of the same name as in the opening "code *id*;" declaration at the top of the file. Alternatively, this routine may be called "**main**". This routine may have arguments corresponding to the value slots in the process boxes of the Code Names View. You have to declare the types of these arguments in the usual manner for a routine.

12.1.16 Additional Keywords, Operators, and Poker C Routines

Keywords	Constants	Operators	Routines
bool	PEi	<-	USERTICKS()
code	PEj		DataAvail()
library	PEn		CheckPort()
port	PEid		IsEOS()
ports	FALSE		ClearEOS()
trace	TRUE		ExitPE()

12.2 The XX Programming Language

Purpose: The XX (dos equis) programming language is a simplified sequential programming language for defining the codes to be executed by the processing elements of the CHiP computer family emulator, the Pringle Parallel Computer.

Activity: Files are created or modified using a conventional UNIX editor. The files are named <proc>.x where <proc> is the name of the process referred to in the Code Names entries. For convenience in referring to Poker state information on the bitmapped display, it is recommended that XX program files be developed on the secondary Poker display.

Programs: XX programs begin with a preamble that gives the program name, the formal parameters, trace variables and the port names. The preamble is followed by the program body block:

<program> ::= **code** <id> <parmlist>; <tracelist> <port list> <body>
<parmlist> ::= (<idlist>) | λ
<tracelist> ::= **trace** <idlist>; | λ
<portlist> ::= **ports** <idlist>; | λ
<idlist> ::= <id>, <idlist> | <id>
<body> ::= **begin** <declarations> <statlist> **end.**

where the parameters identifier list and **trace** identifiers list are limited to at most four identifiers each separated by commas and the **ports** identifier list is limited to 8 identifiers separated by commas. The identifier following **code** names the program and should match the <name> of the file and the <name> used in the Code Names entries. The parameters are formal parameters that correspond one-to-one to the actual parameters stored in the Code Names/Parameters entries of the PEs; each formal parameter must be declared in the <declarations> section of the <body>. The trace list identifiers have their values displayed during tracing and they must be declared in the <declarations> section of the <body>. They are traced in the order given in the *trace* declaration. The port list identifiers are the symbolic port names that are assigned physical positions in the Port Names entries. Values sent and received through ports are typed and coerced at the destination as described in Table 2.

Declarations: There are five data types: Booleans (1 bit), characters (8 bits), short natural numbers (8 bits), signed integers (32 bits), and signed floating point numbers (32 bits). All identifiers, except statement labels and identifiers

declared in **port** or **ports** declarations, must be declared to have a data type; notice this includes the formal parameters. Simple identifiers are scalar values of the indicated type. Identifiers followed by square brackets specify arrays of one or two dimensions whose indices in a dimension run from 0 to <unsignint> in values:

<declarations> ::= <decl>; <declarations> | λ
<decl> ::= <type> <varlist>
<type> ::= **real** | **int** | **bool** | **char** | **sint** | **port**
<varlist> ::= <varid>, <varlist> | <varid>
<varid> ::= <id> | <id> [<unsignint>]
| <id> [<unsignint>, <unsignint>]
where no <id> appears more than once.

The **port** declaration specifies that the variable can be assigned port name constants (declared in the **ports** declaration of the preamble) and can be used in functions such as dataavail (see Built-in functions, below.) Variables of type **port** can be used with assignment statements, I/O assignments, and the relational operators = and < >; otherwise, **port** variables cannot be used in expressions; **port** variables cannot be traced.

Statements: The statements are:

<statlist> ::= <lstatement>; <statlist> | <lstatement>
<lstatement> ::= <id>: <statement> | <statement>
<statement> ::= <assignment> | <conditional> |
<while> | <break> | <for> | <compound> | <io> |
<gosub> | <return> | <exit> | λ

where <id> is used for the target of a **gosub**.

Assignment: The Assignment statement is:

<assignment> ::= <varid> := <expression>

where the coercion to the left-hand side identifier type is provided as described in Table 2.

Conditional: In the Conditional statement

<conditional> ::= **if** <expression> **then** <lstatement>
else <lstatement> | **if** <expression> **then** <lstatement>

the <expression> must evaluate to a Boolean value and an **else** is associated with the immediately preceding **then**.

While: In the While statement

<while> ::= **while** <expression> **do** <lstatement>

the expression must evaluate to a Boolean value. To assist in synchronization the compiler recognizes the construction **while true do** <lstatement> as a special case and does not generate the conditional branch code.

Break: The Break statement

<break> ::= **break**

has meaning only within the <lstatement> of a While statement, and causes control to skip to the statement following the immediately surrounding While statement.

For: In the For statement

<for> ::= **for** <id> **:=** <expression> **to** <expression> **do** <lstatement>

the two expressions, the lower and upper limits of the iteration, respectively, are evaluted once prior to beginning the loop. If the lower and upper limits are not integers, they are coerced to integers as described in Table 2.

Compound: Notice that the Compound statement

<compound> ::= **begin** <statlist> **end**

is not a block and may not contain declarations.

I/O: The I/O statements

<io> ::= <varid> <- <expression> |
<varid> <- <expression>, <varid> <- <expression>

require that either the lefthand side or the righthand side of the operator be an identifier declared in the **ports** or **port** declarations. If the port name is on the left then the operation is write or send to the indicated port; if the port name is on the right the operation is a read or receive to the indicated port. If two I/O statements are specified, one must be a read and one a write. (The semantics are "simultaneous I/O" but since the current hardware cannot support that, the statement is essentially serially executed.)

When a PE performs a read and the data has not been received, the PE waits until the data arrives. If the type of data sent differs from the type of the variable receiving it, type coercion is performed at the destination as described in Table 2. (For the purposes of synchronization, a keyword

port named **null** is recognized. It does not actually do I/O, but it uses the same amount of time.)

Subroutines: The statement

<gosub> ::= **gosub** <id>

provides a parameterless subroutine branch to the statement with label <id>. The <id> cannot be located on a statement in the body of a **for** or **while** loop. All variables are global. Recursive calls are permitted. Execution of a **return** from the subroutine will cause execution to resume with the statement following the **gosub**.

bool → char: The Boolean bit becomes the least significant bit; others are 0.
char → bool: The least significant bit forms the Boolean.
char = sint: The bit sequences are the same.
sint → int: The 8 character bits become least significant bits; others are 0.
int → sint: The eight least significant bits form the short integer.
int → real: Converted to floating point notation.
real → int: The floating point value is truncated and converted to integer form.

Table 2. Semantics of representation conversion; conversions not listed are performed transitively: type1 → type2 → type3, etc.

Returns: The Return statement

<return> ::= **return**

causes execution of a subroutine to terminate and for control to resume at the instruction following the **gosub** call. Execution of a **return** when no subroutine is pending produces unpredictable results.

Exit: The statement

<exit> ::= **exit**

causes all execution of the PE code to terminate and for the PE to enter the quiescent state.

Assembler: Allows the direct inclusion of assembler statements in XX code.

Expressions: The expressions

<expression> ::= <expression> <binary> <expression> |
<unary> <expression> |
<expression> <relational> <expression> | <builtin> () |
<builtin> (<expression>) |
<builtin> (<expression>, <expression>)
(<expression>) | <varid>
<unsignint> | <unsignreal> | <character> |
<boolean> | PEi | PEj | PEID | PEn | EOS

have precedence and association as in the C programming language. Expressions of mixed type are coerced to the higher type, where types are ranked **bool < char = sint < int < real,** as described in Table 2. The operators are given in Table 3.

Constants: The constants are unsigned integers for <unsignint>, reals in standard formats, for <unsignreal>, quoted (') characters for <character>, and TRUE and FALSE for <boolean>. A special constant, EOS, terminates streams and can be used as the right-hand-side of an I/O assignment only.

Identifiers: All identifiers begin with a letter and are followed by any combination of letters and numerals. The maximum length of an identifier is 10 symbols. Two keyword identifiers, **PEi** and **PEj** of type sint, are available giving the row and column index, respectively, of the PE on which the code is executing. **PEn** is a type sint and gives the number of PEs in a row or column for the current machine. Additionally, a keyword identifier, **PEID,** is recognized by the compiler but not otherwise supported.

<unary>		<binary>	
+ <real>	no op	<real> + <real>	addition
- <real>	negation	<real> - <real>	subtraction
~ <char>	not	<real> * <real>	multiplication
		<real> / <real>	division
		<real> **mod** <real>	modulus
		<real> >= <real>	greater than or equal
		<real> > <real>	greater than
		<real> <> <real>	not equal
		<real> < <real>	less than
		<real> <= <real>	less than or equal
		<real> = <real>	equal
		<char> & <char>	and
		<char> | <char>	or
		<char> || <char>	exclusive or

Table 3. XX operators. The type indicates the highest type for which the operation is defined; the operation is defined for all lower types.

Arrays: Arrays can only be subscripted by character, sint or integer types.

Built-in functions: The built-in functions taking real arguments and giving real results are: pi ( ), asin(<real>), acos(<real>), atan(<real>), cos(<real>, exp(<real>), log(<real>), ln(<real>), sin(<real>), sqr(<real>), sqrt (<real>), tan(<real>), pwr(<real>,<real>), where the arguments to the last function are base and exponent, respectively.

A built-in function, dataavail (<port>), is a predicate that returns TRUE if unread data is waiting on the <port> and returns FALSE otherwise. The argument <port> can be either a constant (from the **ports** declaration) or a variable of type **port**. (Two other functions, bitmap and checkports, are recognized by the compiler but not otherwise supported.)

A built-in function, IsEOS (<port>), tests a port (either constant or **port** variable) to be equal to the end-of-stream delimiter, **EOS**, returns TRUE if so and FALSE otherwise.

The relationship between dataavail and IsEOS is as follows: If IsEOS(port) is true then dataavail(port) is false. If IsEOS(port) is false then dataavail

(port) may be true - meaning that the stream has not ended and the next data item is available - or it may be false meaning that neither data nor an **EOS** token has arrived. An acceptable program sequence for reading a terminated stream is:

```
while ~ IsEOS(port) do
  if dataavail(port) then begin ... end;
```

This will busy wait on data, process it in the compound statement when it arrives, and exit the loop when the stream ends.

Multiple terminated streams, may be passed between PEs. To do so, the EOS of one stream must be purged by executing the statement clearEOS (<port>) in order to permit the next stream to enter.

Comments: Comments begin with the characters /* and end with the characters */.

13 Command Request

Purpose: To convert the source form of the program, as specified by the switch settings, code names, port names, IO names and the XX programs, into object form for execution.

Display: The field is cleared and status information is given.

Activity: Commands are invoked which cause the database to be compiled and loaded into the Pringle emulator, the generic emulator, the Cosmic Cube emulator, the Pringle, the Cosmic Cube or other physical hardware. The object machine is specified using the "set machine" command (See Section 15.) This specification carries with it an implied source language for the code segments which in turn is reflected in the <ext> field of the files produced in this view. We recognize the pairs given in the table. In addition the communication structure is compiled and loaded into the emulator. Diagnostics are given in the appropriate ".err" file and in the file Clipboard, which may be displayed using ^f with a blank command line.

Set Variable Value	Source Language	Object Machine	<ext>
pringle-em	XX	pringle emulator	x
generic-c	Poker C	idealized machine	pc
generic-xx	XX	idealized machine	x.pc
ccube-em	Poker C	cosmic cube emulator	.pc.cce
ccube	Poker C	cosmic cube	.pc.cc

Table 4. Definition of <ext> field for Poker produced files.
Set variables are defined in Section 15, Source languages in Section 12 and Object Machines in Section 16.

Recognized keys:

Key	Name	Description
$m	Make	For each process name <proc> given in code names, a file named <proc>.<ext> is sought, compiled into <proc>.<ext>.o and link edited into .pe<i>,<j>.o, where <ext> is determined by the value of the set variable "machine" (See Section 15) as explained in the table. The switch settings are compiled into connections.o, and the whole object specification is loaded into the object machine.
^c	Compile	If the command line is blank, then for every <proc> given in code names, the file <proc>.<ext> is sought and compiled into a file called <proc>.<ext>.o where <ext> is defined in the table. If the command line contains the name of a process <proc>, only it is compiled. Diagnostics are given in <proc>.<ext>.err.
^k	linK	The object code for the process for each PE<i>,<j> is link edited and stored in the file .pe<i>,<j>.o. [Note .pe<i>,<j>.o is not listed by the UNIX ls command without the -a option.] Diagnostics are given in pe<i>,<j>.err.
^n	coNnections	The communications graph given in switch settings is "compiled" into an adjacency list form and stored in connections.o. A symbolic version is given in "connections" and diagnostics are given in the Clipboard file.
^l	Load	The object machine is loaded with the link edited object codes, the connection files and the file-to-stream bindings in preparation for emulation.
^g	Go	If the command line is blank, the current phase of the (loaded) program is emulated for 1000 time units or *ticks* and the progress of the emulation is reported each 100 ticks. A nonblank command line is interpreted as the number of ticks to be emulated; progress is reported every 100 ticks.
^d	Dump	The current state of the emulator is saved in the file named on the command line; if the command line is blank the file is called SnapShot. These files serve as checkpoints with which Poker can be restarted.
^r	Reload	The saved emulator state stored in the file named on the command line is reloaded into the emulator; if the command line is blank, the file name SnapShot is used.

^b	liBraries	Compile the libraries.
^i	Interface	Write the definitions file that establishes an interface to the object machine.
^w	Write	Write the file of "make" dependencies.
^\	interrupt	The object machine is interrupted.
<key>		Key input is directed to the command line.

Commands: The functionality of the above operations is augmented through the use of execute commands. The following are of particular interest for Command Request

bind <stream> <file>
continue [<condition>] [trace | notrace]
display
flushbuffers
run [<phase>] [<condition>] [trace | notrace]
script <file>
set <parameter> <value>

Refer to Section 15 for details.

Global Commands

$h	cHip	$^a	Abort	^f	File
$s	Switch	$^e	Exit	^v	+ page
$c	Code	$^o	Output screen	$v	- page
$p	Port	$d	reDraw screen	^u	eof
$i	Input/Output	^p	Phase	$u	bof
$r	help	^x	eXecute	$$$	noop
$t	Trace	^h	backspace	^z	pause

14 Trace

Purpose: To execute a Poker program on an object system and to display the current values of the traced variables. See Section 16 for a listing of object systems.

Display: The code name and the current values assigned to the trace variables of PEs in (a portion of) the lattice for this phase are given in the field. One display format shows boxes representing PEs; the other display format shows boxes representing PEs and lines representing the interconnection structure; a key (^t) toggles between the two. The code name is clipped to five characters (and cannot be changed) and values are shown clipped to the first 10 symbols:

```
I  name-  I
value1----
value2----
value3----
value4----
```

If the value of a traced variable changes between two consecutive displays, the value is highlighted.

An equal sign (=) appearing in the home position of a PE indicates that when emulation was suspended, the PE was executing (as opposed to being quiescent). An asterisk (*) in the home position of a PE indicates that the PE has "crashed" due to a fatal internal error.

On the command line the elapsed execution time is given in a unit called a "tick". The amount of time represented by this unit depends in the object system as described in Section 16.

Cursor motions: Gross cursor motions advance the cursor to the home position of the next PE in the indicated direction; fine cursor motions (north, south) move to the first position of the windows for the code name and trace variables. "Fine home", from a window moves the cursor to the "home" position of the PE; from the "home" position in a PE, "fine home" moves back to the last referenced line. "Gross home" moves the cursor between the command line and the "home" position of Last PE.

Activity: The execution of a loaded program is controlled and the values of the traced variables are displayed. Execution can be effected in single step units, multiple steps or until any displayed trace variable changes value.

Limitations: This view cannot be entered unless a program is loaded. Incidentally, "poke" – the feature that gives Poker its name – is not implemented.

Recognized keys:

Key	Name	Description
^c	Center	The cursor is moved to the PE whose index is given on the command line or if the PE is not visible the display is changed so the PE is as close to the center of the field as possible consistent with the requirement that the field be fully utilized; if the command line is blank, the Last PE is used for centering.
^l	Load	The object machine is loaded with the link edited object codes, the connection files and the file-to-stream bindings in preparation for emulation.
^g	Go	The command line is interpreted as the (integer) number of ticks the object machine is to execute of the current phase; if the command line is blank 1000 ticks are executed. The new values of the trace variables are displayed at completion of the execution and a report of progress is given every 100 ticks.
^t	Toggle	The display is changed to the "other" format; see Display above.
^y	displaY	The full (unclipped) entry of the traced value window containing the cursor is given in the chalkboard. Notice that the command is needed because certain number representations require more than the available ten character field size.
^e	Event	The command line is interpreted as a pair of positive integers <e> <t> specifying that the object system is to execute the current phase for either <e> events or <t> ticks, whichever comes first. An *event* is defined to be a change in one or more of the currently displayed trace variables. The traced value causing the event is highlighted on the display. A summary of the number of ticks remaining before the emulator stops is given on the diagnostic line. A command line containing only one integer <e> is equivalent to a request of <e> 1000, and a blank command line is equivalent to a request of 1 1000.

^d	Dump	The current state of the object system is saved in the file named on the command line; if the command line is blank the file is called SnapShot. These files serve as checkpoints with which Poker can be restarted.
^r	Reload	The saved emulator state stored in the file named on the command line is reloaded into the object system; if the command line is blank, the file name SnapShot is used.
^\	interrupt	The execution of the object system is interrupted and the current values of the traced variables are displayed.
<key>		The text is directed to the command line.

Commands: The functionality of the above operations is augmented through the use of execute commands. The following are of particular interest for Trace view:

continue [<condition>] [trace | notrace]
flushbuffers
log [<file>]
run [<phase>] [<condition>] [trace | notrace]
script <file>
test PEerr

Refer to Section 15 for details.

Global Commands

$h	cHip	**$**^a	Abort	^f	File
$s	Switch	**$**^e	Exit	^v	+ page
$c	Code	**$**^o	Output screen	**$**v	- page
$p	Port	**$**d	reDraw screen	^u	eof
$i	Input/Output	^p	Phase	**$**u	bof
$r	Request	^x	eXecute	**$$$**	noop
$t	help	^h	backspace	^z	pause

15 Execute Commands

Purpose: To perform transformations on the source or object states of the system.

Activity: Textual commands and their associated parameters are given on the command line in any view (except CHiP Parameters), and followed by ^x. A blank command line gives a list of the available commands.

Commands: The recognized commands are given below. In general, their syntax is prefix notation with operands separated by one or more spaces; the exception is *copy*, where the separator is a comma. Note that commands require only as many letters as are necessary to disambiguate them from other commands; usually a one or two letter prefix suffices.

Recognized Commands

alias <index specification> <*name* 1> <*name2*>
Interphase variable <name2> becomes an alias for <name1>; that is both names refer to the same value for those PE(s) specified in <index specification> where <index specification> is the same as for deposit (^d) in Code Names or Port Names.

bind <stream> <file>
The streams named <stream> of the current phase are associated with the file named <file> as described in "External Data Files" of the file management section (Section 6).

continue [<condition>] [trace | notrace]
The command resumes emulation in Command Request and Trace views only. If no parameters are given then the parameters of the last run or continue command remain in force. The <condition> is a termination condition:

all	Run the phase until all PEs halt.
any	Run the phase until some PE halts.
<i> <j>	Run the phase until PE<i><j> halts.

The trace modifier causes all traced value changes to be displayed; with notrace no changes will be given.

copy <from structure>, <to structure>
Database entities are transferred as described in Section 6.

display <infotype>
Information about the current settings of system or program variables is presented to the user in the Clipboard. The options are:

<infotype>	*meaning*
bind	Lists the stream names and file names bound to them for the current phase (see bind command).
set	Lists the current values of the system parameters that can be set (see set command).

Notice that the <infotype> specification must be given in full.

flushbuffers The contents of the output stream buffers are written to the appropriate files.

help Displays the recognized commands for the current view in the chalkboard; it is equivalent to attempting to change to the current view.

if <i> <j> <var> relop <val> **skip** <l>
This construct is meaningful only for script files where it provides conditional execution of Poker commands. The value of the bool <var> of PE <i> <j> is tested and if it is true the next <l> lines of the script file are skipped. The possible values of <relop> are: <, >, <=, >=, <> (not equal), and =. The possible values of <val> are: TRUE, FALSE (all upper case letters, a character, or a real number; integers are not accepted.

peek <index specification> <name>[file]| <index specification>*[<file>]
The value of the interpahse variable <name> of the PE(s) specified by <index specification> is appended to <file>; the <index specification> is the same as is used for the deposit (^d) command of the Code Names or Port Names View; if * is given instead of the <name>, all interphase variables are logged into the <file>; if <file> is not given, the default "Clipboard" is used.

poke <index specification> <name> <value>
The interphase variable <name> of the PE given by the <index specification > is assigned the <value>. The <index specification> is the same as used for the deposit (^d) command of the Code Names or Port Names View.

log [<file>]
Append to the <file> the current values of all of the traced variables. If the <file> is not specified, the default "Log" is used.

replace <old string> <new string>
In the Code Names, Port Names and IO Names editing views the <old string> is uniformly replaced by the <new string> whenever it occurs in the current phase of the current view. Notice that this is literal textual substitution and substrings matching <old string> are unaffected.

reset [<phase>]
This command resets the emulator to the phase given, or current phase if none is given. This differs from the global command phase (∧ P) which does not communicate with the emulator. This command is useful to reset the phase for use with Event or Go in Trace Mode.

run [<phase> [<condition>[trace | notrace]]]
In Command Request and Trace views only the <phase>, given by its number or symbolic name, is emulated until the given terminating <condition> occurs:

all	Run the phase until every PE halts.
any	Run the phase until some PE halts.
<i> <j>	Run the phase until the PE whose index is i j halts.

The trace modifier causes all traced value changes to be displayed; notrace means no changes will be given. The default are: the current phase for <phase>, all for <condition> and notrace.

script <file>
The keyboard is replaced by the <file> as the primary input source. All

commands (except script) are legal. The following conventions are used:

Keyboard	Script file
control <letter>	ˆ<letter>
escape	$
<fine cursor>	% <digit>
<gross cursor>	$% <digit>

where ˆ is the caret symbol and <digit> is the numeral corresponding to the direction, e.g. % 2 is a fine south cursor move; see Figure 3. Additionally, ! is recognized as "wait for keyboard input, ignore it, and proceed", IF POKER IS WAITING FOR KEYBOARD INPUT, A PERIOD (.) IS DISPLAYED TO THE LEFT OF THE PHASE INDICATOR IN THE STATUS AREA. The two character pairs, /*, */, are recognized as begin comment and end comment, respectively. All spaces are compressed to a single space.

set <parameter> <value>
The global system parameter is set to the specified value. The parameter name must be given in full. The options are:

<parameter>	*meaning*
buffersize	Define the number of bytes of buffer space allocated to each port; the number must be in the range [8, 256] and will be rounded up to a power of two; the default value is 32.
cflags	The value is a string defining a flag sequence that is to be passed to the appropriate sequential process definition compiler, either pkcc or XX. (See Section 12.) The default is -g.
checkpoint	Define the frequency of automatic checkpointing of the emulator state; checkpoints will occur whenever <value> $\neq 0$ and (numticks) *mod* <value> $\equiv 0$; the checkpoint is stored in the file SnapShot and replaces any previous checkpoint. The default value is 0.
includes	**true/false** set to true if any of the code files use the cpp include feature; the default, **false**, is used otherwise. After setting, write (ˆw) the make file in Command Request View.

lflags — The value is a string defining a flag sequence that is to be passed to the appropriate loader. The default is blank.

library — The value is a string of library names, separated by spaces, defining the files to be *included* with the sequential process definition text. The default is blank.

logpoint — Define the frequency of automatic logging of traced values; logs will occur whenever <value> $\neq 0$ and (numticks) *mod* <value> $\equiv 0$; the log of the traced values is appended to the file Log; the default value is 0.

machine — *Defines which system will execute* the Poker program (See Section 16.) Notice that machine must be set prior to the compilation activities of Command Request view. The default is generic-c. The legal values are:

- generic-c — Specifies the idealized machine whose timing model is given by the poker-time-model file. The sequential process code must be written in C; see Section 12.1.
- *ccube-em* — *Specifies the cosmic cube emulator* The sequential process code must be *written in* C; *see Section* 12.1.
- ccube — Specifies the cosmic cube hardware The sequential process code must be written in C; see Section 12.1.
- generic-xx — Specifies the idealized machine whose timing model is given by the poker-time-model file; the sequential process definitions must be written in XX. The Pringle hardware is available at Washington.
- pringle-em — Specifies Pringle emulator; the sequential process code must be written in XX.

switches — Defines whether the switches (circles) in the switch settings view are to be displayed or not; the legal values are *visible* and *invisible*. The default is *invisible*.

model — The value is a string defining the path name of the time model file to use when profiling a generic simulator.

The current values of the set variables are saved when the CHiP Params file is saved - usually with a copy or an exit - and should thus retain their effect between sessions.

shell <command>
The argument is treated as a text string and is sent to the UNIX shell to be interpreted as a UNIX command. A response from UNIX, if any, is sent to the Clipboard and is displayed with ^f.

test <predicate>
The auxiliary function <predicate> is applied to the source database and the resulting diagnostics are displayed in the Clipboard. Auxiliary functions perform diagnostic analysis on Poker programs to insure correctness. The choices for <predicate> are:

paths	Test that all edges given in switch settings are connected paths between two PEs or between a PE and a pad.
pads	Test that the stream specifications are consistent, e.g. that streams with a common name are all either input or output, etc.
ports	Test that for each named port of a PE there is a data path defined, and for each data path connected to a PE there is a port name.
PEerrs	Test the PEs for internal errors such as reading from a terminated stream, divide by zero, etc.

If <predicate> is null a list of the available auxiliary functions is given on the Diagnostic line.

unalias <index specification> <name>
Remove all names aliased to <name> for the PE(s) given in the <index specification>.

unbind <stream>
The indicated <stream> and associated file of the current phase, if any, is purged from the bind table.

16 Object Systems

This section describes the systems capable of executing Poker programs; those systems which are generally available are at present only software emulators. (The Pringle hardware is available only at the University of Washington and Indiana University, and the "cube" version of Poker runs only on the CalTech Cosmic Cube. Researchers interested in porting Poker to other parallel hardware should consult "Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment", *Proceedings of the International Conference on Parallel Processing*, IEEE, pp. 628-635, 1986 and contact the Blue CHiP Project for assistance.)

16.1 Pringle Emulator

When Poker is run with the machine "set" to pringle-em, the execution provided in Command Request or Trace views is a software emulation of the Pringle Parallel Computer. (Notice, only those programs with process definitions written in XX can be run with this emulator.) When n=8 the emulator essentially duplicates the function and timing (tick=microsecond) of the hardware, and thus it provides a means of demonstrating achievable performance. Though it provides accurate results, the use of the Pringle emulator has two liabilities: pessimistic performance estimates and speed.

The Pringle was built with conservative technology in 1982. As a result it was running by February 1983, and so it predates most of the presently available parallel machines. Moreover, the Pringle is not a CHiP computer, but rather a hardware emulator, so it understates the achievable performance. (The Pringle implements the interprocessor communication using a polled bus rather than the point-to-point communication of CHiP architectures; the initial position of the bus polling mechanism explains the differences in execution times that one sometimes observes between two conservative runs of the emulator; it is advisable, therefore to execute the program several times to establish the performance range.) For these two reasons - antiquated technology and only approximate modelling - the Pringle and hence, its emulator, grossly understate the CHiPs performance on Poker programs. Because processing and communication can be sped up by different amounts, it is difficult to predict how pessimistic the performance is.

Since the emulator duplicates every detail of the Pringle's function at a "1 microsecond granularity", it is extremely slow. While writing and debugging small programs this fact is of little consequence. But when the programs become large, the slow performance can be an impediment. It is advised that the emulator be run off-line on a higher performance cycle server by invoking,

poker <script>

or that the system be run through the faster x2c. (See Section 16.3.)

16.2 Generic-pc

When Poker is run with the machine "set" to generic-pc, the execution is provided by the host computer. The source language is Poker C (See Section 12).

Poker C is translated into C and compiled to run on the host computer. A runtime library is provided to implement the Poker abstraction. All processor codes and the runtime library are placed on a single UNIX process. This process communicates with Poker using an Interface in the run time Library.

The runtime Library implements "Light-weight processes". At the beginning of a phase, a "light weight process" is started for each processor. These processes are managed by the runtime system to advance "usertime" for all processes at approximately the same rate. Special calls to the runtime system are inserted at critical points by the Poker C-compiler to insure that one process does not "starve" the others. Using generic-pc allows the user code to run at host machine speed, yielding much faster simulations when compared with the Pringle emulator. This is the main advantage of the generic system. Some time is used in processing the Poker abstraction. Since there is no good concept of time, an artificial concept has been added. This requires a counter to be incremented at strategic places in the user code. To assist in making time estimates, the time model and automatic embellishing of the user code was implemented. (See Section 12.)

16.3 Generic-x (Extasy)

Poker programs with process definitions written in XX can be run through the generic emulator by preprocessing them through x2x. This is performed automatically provided the text files have the ".x" extent and the machine is "set" to generic-x. A time model can be given as described for the generic-pc system.

17 Acknowledgments

Poker is the product of the ideas and efforts of many people. Janice E. Cuny and Dennis B. Gannon, in addition to contributing to the definition of the XX programming language, were a continual source of ideas, judgement and constructive criticism. Version 1.0 of Poker was written during the summer of 1982 by a delightful and committed group of gentlemen, the "Poker players": Steven S. Albert, Carl W. Amport, Brian G. Beuning, Alan J. Chester, John P. Guaragno, Christopher A. Kent, John Thomas Love, Eugene J. Shekita and Carleton A. Smith. Primary contributions to Version 1.1 were made by Steven J. Holmes and Ko-Yang Wang; their work steadily enhanced the system. The 1984 "Poker players", another congenial group, completed Versions 2.0 and 3.0: Kathleen E. Crowley, S. Morris Rose, James L. Schaad, and Akhilesh Tyagi. Further contributions to the completion of Version 3.0 were made by Philip A. Nelson and David G. Socha. Crowley, Nelson, Schaad and Socha, who completed Version 3.1 during the summer of 1985, made valuable comments that improved the document. Version 4.0 was produced by Nelson, Schaad and Socha during 1986 with assistance from Kathleen Crowley, Robert Mitchell and Raymond Greenlaw. Richard Korry has been largely responsible for extensions and improvements of versions 4.1 and beyond. Finally, Julie K. Hanover and Eriko De La Mare managed to retain their patience and good cheer throughout the preparation of this complex document and its many predecessors. It is a pleasure to work with such fine people and to acknowledge these valuable contributions.

Since the first distribution, the Poker users, especially the University of Washington Applied Math group, and the groups at the University of Massachusetts and University of California at San Diego have been helpful in reporting bugs and suggesting improvements.

The design, implementation, testing, documentation and preparation for distribution of the Poker parallel programming environment have been activities of the Blue CHiP Project which has received funding from the following sources: The Office of Naval Research contracts: N00014-81-K-0360, SRO-100, N00014-84-K-0143, N00014-85-K-0328 and N00014-86-K-0264, National Science Foundation Grant DCR-8416878, and a Department of Defense Instrumentation Award.

A Poker Program Library

A library of Poker programs and Poker program schemas is located in /usr/poker/lib. [CAUTION: Your installation may differ.] A Poker program schema, which encapsulates the parallel aspects of the program, is a Poker program with one phase, "standard" names and trivial process code files; use of a schema saves programming time for routine interconnection structures, since they are easily modified to become a computational phase.

A Poker program is stored as a UNIX directory containing the five Poker database entities – CHiPParams, SwitchSet, CodeNames, PortNames and IONames – and the associated process text ("*.pc*") files. A schema is like a program except that the text files are trivial.

A program or schema can be read into Poker using the copy command (see Section 6). Notice that the stored program is "typed" information, that is it contains a description of the particular architecture for which it was written, e.g. number of PEs, corridor width, etc. When reading in the stored program or schema the CHiP Parameters of the current Poker system should match those of the program. Program names are capitalized and schema names are not. Furthermore, as a heuristic schemas use a naming convention of the form

<short graph name><n>.<w>

where the <short graph name> briefly describes the interconnection structure, <n> is the number of PEs on the side of the lattice, and <w> is the corridor width (see Section 7.)

The structure and contents of the Poker library are shown in Figure A1, where a trailing slash (/) indicates a directory. Within each directory are the files:

CHiPParams
SwitchSet
CodeNames
PortNames
IONames

and one or more text files (e.g. proc.pc), where the names depend on the logical process types used in the CodeNames entity.

Notice that for most of the above schemas, smaller instances can be created by first loading the schema into a one phase Poker system, changing the CHiP Parameters to the smaller size, (possibly) fixing a few dangling edges, and saving the result. Larger instances can be constructed using the ssc program; see Appendix B.

PlayingPoker/	sample program from Playing Poker document
BatcherSort/	sort program based on Batcher's algorithm
hex16.1/	16 × 16, corridor width 1, hexagonal mesh schema
hex16.4/	16 × 16, corridor width 4, hexagonal mesh schema
mesh16.1/	16 × 16, corridor width 1, 4 mesh schema
mesh16.4/	16 × 16, corridor width 4, 4 mesh schema
shuff8.1/	64 node shuffle exchange graph schema, no pads
torus16.1/	16 × 16, corridor width 1 interleaved torus schema
tree16.1/	255 node tree schema, pad to root on East side
BGboot	Program sent to Bitgraph for doing screen dump
BGupload	Program sent to Bitgraph for doing screen dump
PokerDemo	Directory for use in Playing Poker
bitcap	Termcaps for Bitgraph and ATT 5620
emstartup	datafile for Generic emulator

Figure A1. Structure of library as distributed.

B Related Systems

Poker as described in the foregoing sections represents the production form as it exists at the University of Washington. Other users may prefer other systems because of different hardware or different scientific objectives. The known options are described below; users who port or substantially modify Poker are encouraged to contact the Blue CHiP Project in order that their contribution may be entered into the list.

B.1	Poker Cleanup	B.7	Batch Poker
B.2	XWindow Implementation	B.8	Printing Screen Images
B.3	AT&T 5620 Support	B.9	Switch Set Copy Utility
B.4	Helpful Hints	B.10	Pen Plotting - bbplot
B.5	Backend Interface Language	B. 11	Poker Coordination
B.6	PackIO	B.12	Prep-P

Figure B.1 Appendix table of contents

B.1 Poker Cleanup

The program *cleanup* may be used to remove the Poker generated files from the current director. This is useful in reducing the amount of disk space used by Poker programs. It is possible to completely rebuild the state from the remaining files.

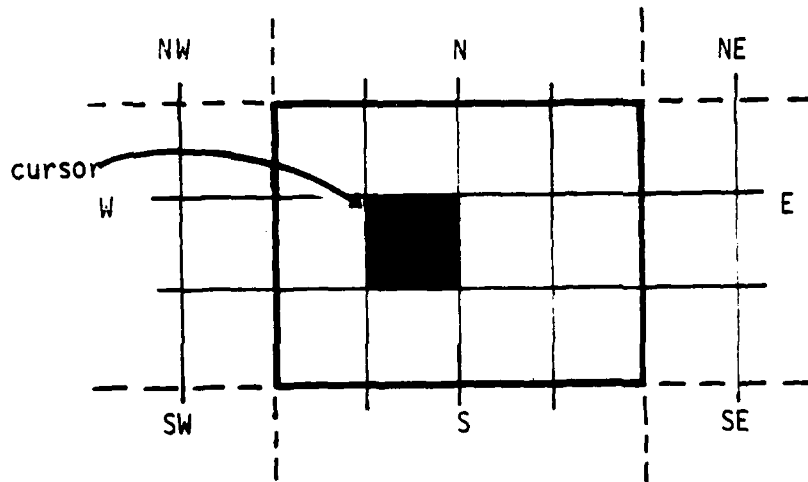
B.2 X Window Implementation

Since many users now have workstations, the Poker system has been ported to the popular X Windows system. Currently it is offered only for version X10.4. (Poker's original display, the BBN Bitgraph is a display that uses "portrait" mode while most workstations (e.g. Suns, IBM RT's, DEC Microvaxes) use "landscape" mode.) As a result, Poker under X has two windows. One contains the field while the other displays the lattice, chalkboard, clipboard, diagnostic and command areas. The cursor can be in either window for input. The mouse is not used for cursor motion; the numeric keypad is still used for that. Due to X Window limitations, the program CreatePixMaps **should** be run once as part of poker installation.

B.3 AT&T 5620 Support

The AT&T 5620 Dotmapped Display (or "BLT") is supported as an output device for Poker. The screen format is the same as that of the BitGraph display. To run Poker using a BLT. it is necessary to download software to the terminal using an alias, called "setPoker", stored in /usr/poker/bin/setPokeralias. [CAUTION: Your implementation may vary.] There are two implementations of the software – one that runs "stand alone" and one that runs under layers. The software can remain resident throughout the session, i.e. it need not be reloaded each time Poker is run.

Cursor motions for the BLT: The keypad is not used for cursor motions, because it is not accessible via the available software. Rather, the mouse is used to indicate direction. A direction is specified by the relationship of the mouse pointer to the current position of the cursor when button 1 (leftmost button on the mouse) is pressed. For example, if the mouse pointer is below the cursor when button 1 is pressed the direction specified is south, and therefore, equivalent to pressing 2 on the keypad of the BitGraph. Specifically, to determine direction, visualize the cursor as if it is leftcenter in a 3×4 character box and assign directions with respect to the box:



When the pointer is within the box the "home" specification is implied. To perform gross cursor motions, press an escape ($) followed by a mouse click to specify direction. Notice that rapidly entering cursor motions can sometimes result in anomalous behavior because the cursor position changes during screen updating, it is advisable to press button 1 only when the mouse pointer is visible.

Button 2 (center) presents a "pop-up" menu from the terminal support software. The options are as follows:

continue	no operation
^Q/^S	enable/disable terminal flow control
nullterm	ignore mouse, except this button
poker	emit symbols for Poker
spread	enable/disable character expansion
clear screen	clear screen
exit	leave terminal emulator

Notice that the options are "commands to change state", so their sense is opposite of the current state. It is advisable to disable *spread* because the Poker display format is based on standard size characters.

Button 3 (right most) does nothing in stand alone mode but provides the layers menu when running under layers.

B.4 Helpful Hints

This appendix is a quick list of hints to help you program and figure out what is going wrong, in other words, what does this error message *really* mean.

1. "Fatal error in c2c". C2c is failing and pkcc is not passing on the message. To get a better idea of the c2c error, you need to invoke c2c expliticly. This is done by running the C preprocessor /lib/cpp then c2c. For example, % /lib/cpp file.pc | c2c > temp.c This also works with cc2c.

2. "e cannot open file dataout". There is usually a problem with directory permissions.

3. "e cannot open file datain". There is a problem with bindings.

4. "e emulator stops due to deadlock". Try using a trace variable that is set to a unique value before and after each port read in your program and run the program in trace mode in Poker. When it stops you can figure out which PE's are reading from which ports. This technique is a lot easier than doing the same thing interactively with the emulator.

5. The emulator dies inside Poker. Your best bet is to run the emulator outside Poker, i.e. from the Unix shell, and see if any error messages are visible. The next step is to use dbx on Generic-pc if it produced a core file. That will usually give the line number in the Poker C program of the statement that's causing the problem (if it's caused by accessing an array out of bounds or something like that). Also make sure any non-poker-C functions are declared type library at the start of the .pc files.

6. Lots of bugs can be caught by running lint on .pc files. You can include a shell script that uses grep to cut down on the amount of meaningless and confusing output. The sequence of commands to use is: % /lib/cpp file.pc | c2c > temp.c; lint temp.c

7. Poker dies at startup when it tries to read IONames and you've been using the ssc program. Try removing IONames and run Poker again.

Programming Tips:

1. If unusual things are happening during compilation, use the "cleanup" command from the Unix shell to remove all unnecessary files from your working directory and then delete "Make-pc" or the equivalent makefile. This forces poker to recreate the makefile and ensures all programs will be recompiled.

2. When you leave Poker, via $^e, be in Command Request mode. In this mode, there are fewer graphics, and a slow terminal bandwidth will not slow you the next time you start up Poker.

3. Make liberal use of job control (^z) in Poker. For example, if in Poker you get a compiler, the linker, and so on error, type ^z to pause Poker, look for the problem, fix it, and then type "fg" to get back into Poker. This is faster than exiting and restarting Poker each time since Poker doesn't have to reload its data files. However, don't forget about your background Poker jobs, or they will clog the system and possibly lead to confusion.

4. When loading a number of data pieces such as an array from a file into PEs, use vertical interconnections to load the data upwards from IO pads connected to the bottom PEs. Then have each PE pass data up to the PE above it bucket-brigade fashion. This has the advantage of preserving the orientation of the data. To dump it, connect the IO pads to the top PEs and again have each PE pass data upwards.

5. If possible and sensible, it is nice to have port names short enough so that the entire name can be displayed in the port names view.

6. ALWAYS save your program (type: "copy *,." from Poker's command line) BEFORE running it.

7. Do not run more than one Poker job in the same directory. Concurrent Poker jobs use the same data files and may interact to produce bizarre errors.

B.5 Backend Interface Language

This appendix outlines the interface between Poker and the various backends (e.g. Generic-pc, Cosmic Cube emulator, etc.). This purpose of this is not to make you an expert on the interface "language" but to provide some basic documentation that can be very useful when trying to figure out "what's wrong?"

This interface was developed when the backend (emulator) part of Poker was split away from the user interface part of Poker and is very simple.

A few definitions:

<index specification>: <startPEi> <startPEj> <endPEi> <endPEj> - this means all PEs starting from <startPEi><startPEj> to <endPEi><endPEj>. For example, 1 2 3 4 in a 4×4 array would mean (1,2), (1,3), (1,4), (2,1), ... (3,3), (3,4). To specify a single PE <startPEi><startPEj> should be the same as <endPEi><endPEj>. For example, 1 1 1 1 would indicate (1,1).

<ipname>: interphase variable name
<stream>: see Section 15 Execute Commands.
<file> : see Section 15 Execute Commands.
<stopCondition> : 0 means run until all PEs are done. 1 means run until any PE finishes.
<PEi><PEj>: this specifies a PE while 0 0 means no PE.
<timeLimit> : a number of User Ticks.

B.5.1 From Poker to the BackEnd

Many of these are similiar to those described in Section 15 Execute Commands. Some additional descriptions are given. Commands that are most often needed when running a back end are marked with *.

alias	- a <index specification> <ipname> <aliasName>
bind	- b <stream> <file>
continue	- c <stopCondition> <PEi> <PEj> : This means continue until <stopCondition> or <PEi> <PEj> halts.
dump	- d : Dump status information. Generic supports: 0 - DeltaUserTime. 1 - PE times. 2 - Errors and error messages.
event	- e <timeLimit> : Run until timeLimit ticks.
flushbuffers	- f : Flush IO buffers.
get interphase	- g <index specification> <ipname> : Get interphase values.
interrupt	- i : When sent while the emulator is running in Poker mode it causes the eumlator to stop at the first possible moment.
kill interphase	- k <index specification> <ipname> : Remove name from interphase space.
reload interphase	- l <file> : Reload interphase variables from file.
dump interphase	- m token : Dump interphase into file.
*set phase	- p <number>
*quit	- q
*run	- r <stopCondition> <PEi> <PEj>
set interphase	- s <index specification> <ipname> <value> : Set interphase values.
set parameter	- t name value : Set internal parameters. Currently Generic supports DeltaUserTime and StackSize. Cosmic Cube does not support any currently.
unbind	- u stream
execute	- x timeLimit

For example, to run phases 1 and 2 (boldface is user input)

% **Generic-pc**
...output...
> **p1**
p1
> **r 0 0 0**
...output...
> **p2**
p2
> **r 0 0 0**
...output...
> q
%

B.5.2 From the BackEnd to Poker

code table info	- c <codenumber> numberOfVars codename : Codenumber is a number to reference the code. NumberOfVars is the number of trace variables. Codename is the name of the code.
error msg	- e msg : Msg is a multi word string.
fatal error	- f msg : Msg is a multi word string. The Backend will exit.
initialize msg	- i PEn numphases numcodes switches : PEn is the PE dimensions. Numphases is the number of phases. NumCodes is the number of codes. Switches is not used (set it to 0).
assign codes	- m <PEi> <PEj> phase codenumber : Assigns codes to PEs.
status	- s userTicks <PEi><PEj> status : UserTicks is the number of UserTicks for that PE. Status is 'h' for halted and 'e' for error.
trace	- t userTicks <PEi><PEj> index valueString : Trace message. UserTicks is the number ticks for that PE. Index specifies the trace variable. It ranges from 0 to 3. ValueString is the value of the trace variable.
update	- u userTicks: An update message indicates all further messages will have a larger tick value.

B.5.3 BackEnd Command Line Parameters

The BackEnd takes command line parameters.

- -P : talking with Poker so that it writes out messages one at a time. It waits for a newline after each message except for the prompt.
- -D : send output to file called Debug.
- -T : send output to file call Traces.

B.6 PackIO

PackIO puts an input (stream) file into the format expected by Poker. Each field is packed into 12 characters, stripping off any characters passed the 12th character or padding the rest of the field with spaces, if needed, and then terminated with a comma in the 13th position. PackIO is a "fixed point operator": repeated invocations on a packed file will not change the file.

The input file must have each entry TERMINATED by a comma. Entries may not have spaces within them, but spaces and tabs may separate entries, commas, and newlines from each other. Make sure to mark "empty" entries with a terminal comma, otherwise packIO will pack your file in a way you might not expect. To quote a comma, i.e. to use a comma as character input, precede it with a single quote ('). The rest of the characters in a field after a quoted character are ignored.

The usage of packIO is:

packIO [<input-file> [<output-file>]]

If no file names are given, packIO acts as a filter, using stdin for the input and sending output to stdout. If only one file name is given, that name is used for the input file, and the output is directed to stdout. If two files are named, the first is the input file, and the second is the output file. The input and output file names must be distinct; if they are not distinct, packIO prints the error message:

Error: input and output file names must be distinct

and then aborts with an error code of 1. If more than three arguments appear after packIO, packIO prints a usage message and aborts with an error code of 1.

If the entry is longer than 12 characters, the extra characters are dropped, and packIO prints a warning message such as:

Warning, line 5, field 2: entry too long; dropping extra characters.

to the stderr device. A warning also is printed for empty lines:

Warning, line 8: empty line in file.

Notice that the files *produced* by Poker are in the proper format as produced and need not be processed by packIO.

B.7 Batch Poker

Poker can be run in a "batch" mode by invoking the system with a file name as a parameter.

poker <script>

where the <script> file uses the conventions described for the script execute command; see Section 15. The result is to run Poker so that the input that would normally come from the keyboard comes from the file <script>. When the <script> file has been processed, the keyboard again becomes the input source. The output is still directed to the screen. It is possible to run Poker in the foreground without using the screen by redirecting the output to a file. The command

poker <script> > <outfile>

stores into <outfile> all of the information that would normally be displayed on the screen; this output can be later "played back" by using the UNIX "cat" command. The output files so produced become enormous (many megabytes) for any reasonable Poker program, so it is suggested that the output be redirected to the bitbucket device, /dev/null. In this case, Poker will only produce results, i.e. output streams stored in files, and there is no graphic record of the run. It is, therefore, recommended that when the screen output is thrown away that log commands (see Section 15) be included in the script. If with redirected output an error occurs while executing the script, Poker will abort (**$**^A); if the script comes to an end, Poker will exit (**$**^E). Finally, to run Poker in the background, one must use ^z, or append an & to the command.

Notice that because batch Poker enables the keyboard when the execution of the script is finished, it is possible to create a script that performs the routine activities normally performed at the start of a Poker session, such as listing .pc files in the Clipboard, etc. in order to save having to perform them manually.

B.8 Printing Screen Images

Screen images (created by **$**^o) are prepared for printing on the laser printer now by poker. See Section 5 Global Commands.

B.9 Switch Set Copy Utility - SSC

Overview

The Switch Set Copy Utility program is a separate utility for manipulating the switch setting entity of the Poker database. This was written as a quick tool for saving time for one Poker user. With a little more work, it was made a little more usable by "any" Poker user.

Switch Set Copy (ssc) manipulates arbitrary rectangles in the switch/pe lattice. The basic operation is to copy one rectangle of switch settings to another rectangle. This includes reflections of the rectangles. The standard copy is a complete copy, but the user can select an "or" mode where previous settings of a switch are not touched. Ssc can handle multiple "SwitchSet" files and multiple phases in a single file. Other commands include turning a square block of switches in increments of 90 degrees and plotting an arbitrary rectangle of the lattice.

Lattice specifications and command formats

In ssc, the user must be able to specify a specific switch, not just the PEs. To avoid counting the number of lattice rows and columns, ssc uses a PE relative addressing scheme for lattice rows and columns. The row with PE 1,1 is addressed as 1. The row of switches above PE 1,1 is addressed as 1-1. The row of switches below PE 1,1 is addresed as 1+1. Also, the row of switches below PE 1,1 can be addressed relative to PE 2,1. For example, with a corridor width of 3, the row of switches below PE 1,1 can be addressed as 2-3.

To specify a particular lattice element, both a row and a column must be given. This is done with the format '<row>;<column>'. For example, the address '1;1' specifies PE 1,1. The address '3+2;4-1' specifies the lattice element that is two lattice rows below and one lattice column to the left of PE 3;4.

Commands are given to ssc in response to the prompt 'ssc>'. Because the first letters of the commands are unique, all that is needed is the first letter. Full command names can be used but all characters between the first character and the first space are ignored. The space is required to separate the command from the parameters. Parameters are separated by commas, and are specified by position; thus, optional parameters must be preceded by the proper number of commas, and commas following the last parameter may be elided.

The commands

1. read [file_name],[logical_name]
write [logical_name],[file_name]

These commands read and write complete SwitchSet files. The files are always referred to using their logical name. This allows for two copies of the same physical file under two different logical names. The logical names must be unique. File names may be complete paths. (Warning: Since this program is written in PASCAL, no attempt is made to verify that the file exists.)

Defaults:

read - file_name = 'SwitchSet'
logical_name = 'ss'
write - logical_name = 'ss'
file_name = original name for file

Examples:

read	reads 'SwitchSet' with logical name 'ss'.
read ,ss1	reads 'SwitchSet' with logical name 'ss1'.
read /u1/phil/prog/SwitchSet,philss	reads the indicated file and calls it 'philss'
write	writes 'ss' to a file called 'SwitchSet'.
write ss1	writes 'ss1' back into the original file.
write ,newss	writes 'ss' into a file called 'newss'.

2. files

This command lists all the files that are in memory. The listing includes the logical name, the file name, the size of the processor array, the internal and external corridor widths and the number of phases. The "current" file and phase are identified in this list. (See visit command.)

3. visit logical_name,[phase_number]

This command specifies the "current" file and phase. Unless specifically noted otherwise, commands operate on the current file and phase. The read command sets the current file and phase to phase 1 of the last file read in. The logical_name is required.

Default: phase_number = 1

Examples:

visit ss	sets 'ss' to current file, phase 1
visit ss,3	sets 'ss' to current file, phase 3

4. copy ul,lr,dest,[from_phase],[to_phase],[from_file],[to_file],[OR]

This command is the reason for this utility. It copies a rectangle of the switch/pe lattice to another identical sized rectangle. The parameters "ul", "lr", and "dest" are what specify the from rectangle and the place to copy it to. "ul" is the lattice element that is the upper left of the rectangle to be copied. "lr" is the lower right lattice element to be copied. "dest" is the

lattice element that the "ul" element will be copied to. The order of copying is determined by the row-major-order sequence of the destination rectangle, and proceeds one switch at a time. Thus, for example, the top row of an array can be duplicated throughout the array by copying rows 1 to n-1 into rows 2 through n. If "lr" is above "ul", the copy goes back row by row from "ul", but the destination is always forward row by row. The switches are correctly reflected about the east-west axis. A similar thing happens for "lr" above and left of "ul", and for "lr" below and left of "ul". The switches are correctly reflected for all copies. The last parameter turns on the "or" mode. Normally, a switch is copied into the new position, over writing the previous setting. A PE copied to a switch leaves an empty switch. A switch copied to a PE does nothing. In "or" mode, current settings of the switch are not disturbed. The copy from a switch sets only unset positions of the destination switch.

Defaults:

from_phase - current phase
to_phase - current phase
from_file - current file
to_file - current file

Examples:

copy 1;1,4;4,1;5	copies PE 1,1 thru 4,4 to 1,5 thru 4,8.
copy 1-2;1-2,2+1;4+2,3-1;1-2	reflects top half of a 4 × 4 to bottom half.
copy 1;1,4;4,1;1,1,2,ss,ss1	copies PE 1,1 thru 4,4 of file ss, phase 1 to PE 1,1 thru 4,4 of file ss1, phase 2.
copy 1;1,2;2,3;3,,,,,OR	copies PE 1,1 thru 2,2 to 3,3 thru 4,4 with the "or" mode.

5. turn upper_left,lower_right,number_of_90s

This command takes a square piece of the lattice and turns it by the specified number of clockwise 90s. The PEs must be placed so that by turning, the PEs end up in the same location relative to the square as before the turn. For example

turn 1;1,2;2,1

is legal, but

turn 1;1,2-1;2-1,1

is not legal. The number of 90s should be an integer in the range 1 - 3. A turn of 4 is a "no operation".

6. plot upper_left,lower_right,[file_name]

This command takes the specified rectangle of the lattice and produces a file for the troff preprocessor "pic". This file when run through pic produces a picture of the rectangle. The user may need to add scaling information to the pic file. The units are in inches. Each lattice element is placed at an integral unit. Seven elements then require seven inches. Scaling and

number size are independent. To size down the numbers use the ".ps" troff command for setting the point size. (See the pic users manual for details.) If no numbers are wanted in the PEs, the macro can be changed by putting an 'X' at the end of the first macro line and deleting the second two lines.

Default: file_name = "picfile"

7. quit

This command does the obvious. No write is performed. Also, no warning is given if a file is modified but not written out.

8. help

This command prints a list of the available commands with their syntax.

Cautions

This program changes part of the Poker database without regard to the other parts of the database. Most notably, information in the IONames view may be lost. It is recommended that this utility not be used to write files after information in IONames view has been entered.

B.10 Pen Plotting - bbplot

A program, called "bbplot," is available for plotting lattices on the HP7585 (Big Bertha) and compatible plotters. This program takes as input a "pic" file as generated by ssc. The program assumes the plotter is inline with a terminal. The plotter commands are written to standard output. The command line is as follows:

bbplot [-pen_number] ssc_pic_file

The pen number is optional and must be a number in the range 1-8. The plot is done relative to p1 and p2, (see HP7585 user manual). Make sure that p1 and p2 are set for the correct size rectangle before issuing the command.

B.11 Poker Coordination

A system is under development by Janice E. Cuny and her students at the University of Massachusetts which "coordinates" Poker programs. A coordinated Poker program is one which has been automatically converted to be run synchronously without the expensive handshaking protocols used for asynchronous I/O [1, 2]. Substantial performance improvements can be achieved with coordination. Inquiries should be directed to:

Professor Janice E. Cuny
Computer and Information Sciences Department
University of Massachusetts
Amherst, Massachusetts 01002

[1] Janice E. Cuny and Lawrence Snyder
Compilation of Data-driven Programs for Synchronous Execution
Proceedings 10th ACM Symposium on Principles of Programming Languages pp. 197-202, 1983.

[2] Duane A. Bailey, Janice E. Cuny and Bruce B. MacLeod
Coordination in the Poker Parallel Programming Environment: Parallel Code Optimization
Technical Report, 85-21, COINS Department, University of Massachusetts, 1985.

B.12 Prep-P

A system, called Prep-P, is under development by Francine D. Berman and her students at the University of California at San Diego which maps large Poker programs down to run on a smaller number of processors [1]. This experimental system accepts a Poker program which has as interconnection structure any undirected graph, and "contracts" the graph to an intermediate graph with a fewer number of nodes. The intermediate graph is then laid out on a CHiP lattice whose size may be specified by the user (at most 8×8). The result is that several processes will be mapped to one PE, so the Prep-P system multiplexes the execution of these several processes and provides for the proper communication. Inquiries should be directed to:

Professor Francine D. Berman
EECS C-014
Applied Physics and Math Building
University of California at San Diego
La Jolla, CA 92093

[1] Francine Berman, Michael Goodrich, Charles Koelbel, W. J. Robison III, Karen Showell
Prep-P: A Mapping Preprocessor for CHiP Computers
Proceedings of the International Conference on Parallel Processing (Douglas DeGroot, editor) IEEE, 1985, 731-733.

C Using Poker on the Cosmic Cube

Introduction

Poker programs written using Poker C can be executed on the Cosmic Cube and Intel iPSC Hyper Cube. To achieve this, all that the programmer needs to do is to specify the use of a different back end from Poker's front end. After this, the steps for running a Poker program on the Cosmic/Hyper Cube are nearly identical to the steps used to run Poker programs on the Generic simulator. Since the procedure for running on the Hyper Cube is almost identical to that for running on the Cosmic Cube, further references to the Hyper Cube will only be made when there is a difference in procedures.

As described in previous sections, the Poker programming environment has two parts: a front end consisting of a programming environment; and a number of different back ends which emulate, simulate, or directly execute the Poker programs. Section B.5 describes the use of the Generic back end. The following describes the use of the Cosmic Cube back end and the Cosmic Cube emulator.

Writing the Poker program

Poker programs do not have to be modified to run on the Cosmic Cube. However, there are a couple of restrictions. First, process codes must be written in Poker C since XX is not supported on the Cosmic Cube. Second, make sure that you specify each integer as being either a short integer (short int) or a long integer (long int). This distinction is needed because the Sun workstations which host the Cosmic Cube assume that unlabeled integers are long integers, while the Cosmic Cube nodes assume they are short integers. This difference can lead to subtle programming errors if integers of unspecified length (int) are sent in messages between the host and cube. The Poker C compiler will warn you when it detects such potential problems.

Debugging program

The next step after writing your program is to debug it. There are three back ends which may be used for debugging Poker programs intended for the Cosmic Cube. The first back end to use is the Generic simulator provided with Poker. This is probably the best place to start since it uses light-weight processes to simulate the Poker PE's and since it provides the best control over the execution of the Poker program. For instance, this simulator allows you to pause the program after each change to a traced variable. It also allows you to interrupt the execution at any point and later resume it. Finally, using Generic does not unduly tie up the resources of the Cosmic Cube as would testing the program directly on the Cosmic Cube.

Section B.5 describes how to use the Generic back end. Briefly, this involves first setting Poker's machine parameter to generic-pc, then making the Poker program from the Command Request View, and finally tracing its execution in the Trace View. The second back end is Caltech's Cosmic Cube emulator. After the program is working under Generic, the program may be executed on Caltech's Cosmic Cube emulator. However, this emulator is not included in the Poker distribution. If you are interested in running it, you first must obtain it from Charles Seitz at Caltech (seitz@vlsi.caltech.edu).

This emulator also cannot handle any Poker program with more than 16 PEs, since the emulator uses heavy weight processes to simulate the Cosmic Cube nodes. UNIX limits the number of heavy weight processes each user may have. The Generic simulator does not suffer from this restriction since it uses its own light-weight processes system for simulating the Poker PEs.

Using the Cosmic Cube Emulator. First, make sure that your UNIX shell environment, "cube," is set to a string of pairs which define the name of your group and of the machine hosting the cube daemon. This can be done by adding a line such as

setenv cube "group test cubedhost machine"

to your .cshrc file. See the Cosmic Cube manual for more information. Then, set Poker's machine parameter to ccube-em. Then proceed to make the Poker program from the Command Request View and trace its execution in the Trace View, in the same way as done with the Generic back end.

The Cosmic Cube emulator is not as versatile as Poker's Generic back end. Specifically, it does not support:

- Interrupting during the execution of a phase.
- Using the "go" and "execute" commands.
- Dumping or reloading interphase variables.

The timing values reported for the execution are elapsed times measured using the UNIX gettime() call. Thus, the values are highly variable and subject to system load and the whims of the scheduler.

The last back end is the Cosmic Cube itself. Using the Cosmic Cube as a back end can be as simple as using Generic, if the Poker front end is on a computer with access to the Cosmic Cube, or slightly more complicated, if you have to remotely access the Cosmic Cube. If you are using Poker on one of Caltech's Sun's or Vaxes which are connected to the Cosmic Cube you can run the Cosmic Cube from within the Poker front end in the same way as you ran the other back ends. Just set the machine parameter to ccube, then make the program from the Command Request View, and then finally trace it in the Trace View. Running the

program from within Poker may be useful for small test programs, but is too cumbersome and slow for most Cosmic Cube Poker programs.

A faster way to execute the Cosmic Cube program is to run the back end directly without the use of the Poker front end. The steps are as follows:

1. Create the Poker program using the Poker front end as usual. Make sure your ints are labeled as either short or long.

2. Create the Poker intermediate files. If you executed your program on the Cosmic Cube emulator, these files already have been created. Otherwise, enter Poker and type

   - CTRL-W, to make the Makefile, mapping, and code files,
   - CTRL-I, to make the inter-phase.c file,
   - CTRL-N, to make the connections.o file, and

3. Create a command file for executing the Poker program. This command file uses the same syntax as Poker when Poker talks to the back ends. Appendix B.5.1 describes this syntax. Three of the most commonly used commands are "p" to set the phase, "r 0 0 0" to run the current phase until all PEs are done, and "q" to quit. An example file to run the first three phases of a program and then quit is:

```
p 1
r 0 0 0
p 2
r 0 0 0
p 3
r 0 0 0
q
```

   Note that the phases are one origin, as the Poker user would expect.

4. Transfer the intermediary files to a computer with access to the Cosmic Cube. This can be done using rcp or ftp. The following files should be transferred: *.pc, Make-cc (Make-ipsc for the Hyper Cube), mapping, inter_phase.c, connections.o, ccode*.c, .CosmicDefs.c.

5. Log onto a computer with access to a Cube computer.

6. Compile the Cosmic Cube executables from the intermediary files. This is done by typing

% make -f Make-cc (Make-ipsc for Hyper Cube)

This make command creates three separate executables: the spawner that interfaces to Poker and controls the execution of the program; the file server (fileio) that is the interface between the Cosmic Cube program and the UNIX file system; and a set of codes to be mapped onto the Cosmic Cube nodes.[8]

7. If necessary, modify the mapping file produced by Poker. The purpose of this mapping file (named mapping) is to specify the type and size of cube to allocate and then map the PEs onto the nodes of the Cosmic Cube. For instance, the first line of the file

```
cosmic_cube 2
1 1 0 0
1 2 1 0
2 1 2 0
2 2 3 0
```

allocates a two-dimensional sub-cube of the Cosmic Cube. The next four lines each map one PE of a 2 by 2 array to its own node. Each of these lines is of the form

```
PEi PEj node processID
```

where the *PEi* and *PEj* are one-based and the *node* and *processID* are zero-based. If there are less than 64 PEs, Poker allocates a large enough cube to map each PE to its own node. The PEs are assigned to nodes according to their row-major order. If there are more than 64 PEs, Poker wraps back to node 0 and uses the next higher process number to map another PE to each of the nodes.

You may want to change this mapping for two reasons. One reason is if you want to run on a smaller cube. This requires that you change the cube size in the mapping file and change the node and process numbers of PEs mapped outside of this smaller cube.

The other reason is that you may want to preserve the locality of your interconnection. Namely, PEs that communicate with each other should be placed on adjacent nodes. In general finding such mappings is a difficult problem. See the "C Programmer's Guide to the Cosmic Cube" for information on the physical connectivity of the Cosmic Cube nodes.

[8]For more details see Lawrence Snyder and David Socha, Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment, *Proceedings of the International Conference on Parallel Processing*, pp. 628-635, 1986.

8. Execute the Cosmic Cube back end. The spawner acts as the interface to the Cosmic Cube back end. To directly execute this back end, you can either invoke the spawner alone and type in the commands to control it, or redirect the standard input from a prepared file of commands, as in

```
% cspawner < commandScript (ispawner for Hyper Cube)
```

The Cosmic Cube shares some of the same restrictions as the Cosmic Cube emulator. Namely, it prohibits use of interrupts while executing a phase, and does not allow use of the "go" or "execute" commands. While it does allow tracing, be warned that the increased message traffic associated with tracing may severely degrade performance. The timing results are reported in milliseconds and indicate the time taken by the last PE to finish.

Problems

Hints

The Cosmic Cube kernels are easily corrupted. If your program doesn't work on the Cosmic Cube but worked on the Cosmic Cube emulator, rebooting the Cosmic Cube kernels often will fix the problem. To do this type (the '%' is the UNIX prompt)

```
% getcube cosmic N  (ipsc for Hyper Cube)
% killcube
```

and wait for the kernel to be re-installed. This takes about a minute. Choose the "N" to be the dimensionality of the cube required by your program. In extreme cases you may have to restart the cubed deamon which controls the Cosmic Cube and its emulator. See Su, et. al.'s "C Programmer's Guide to the Cosmic Cube" for details.

If your Cosmic Cube program has problems and you have to kill it, afterwards type "peek" to make sure that you have freed the Cosmic Cube space for others to use.

How to report problems

If you have problems, please send a message describing the problem and containing the smallest example of it to poker@larry.cs.washington.edu.

How Poker interacts with the Cosmic Cube back end

Snyder and Socha, previously cited, discusses the general plan for composing the Cosmic Cube executables from the Poker sources. Briefly, cpkcc first calls cc2c to translate each of the .pc Poker C codes into a .pc.cc file. This .pc.cc file is written regular C. During compilation, the cc2c compiler also produces a .inter file from each of the .pc files. This .inter file contains a list of each of the interphase variables and its type. Interphase arrays are listed as pointers since their size is determined by the previous export. Once all of the .inter files have been created, they are compared to make sure that there are no inconsistences in the use of the interphase variables. If there are no inconsistencies, the merged set of interphase variables defines the interphase variable space of the PEs. Next, cpkcc calls cc to compile the .pc.cc files into .pc.cc.o files. These .pc.cc.o files are then linked in with a pre-defined main routine, as described in Snyder and Socha.

Poker Command Summary

The symbol $ abbreviates "escape key" and ^<char> indicates striking the <char> key while holding down the "control key."

Global Commands

$h	cHip parameters	$^a	Abort, no state save	^f	display File
$s	Switch settings	$^e	End, save state	^v	+page
$c	Code names	$^o	Output screen	$v	-page
$p	Port names	$d	reDraw screen	^u	eof
$i	Input/output names	^p	phase change	$u	bof
$r	command Request	^x	eXecute command line	$$$	noop
$t	Trace values	^h	backspace	^z	pause
				^]	UNIX Interrupt (abort)

Specific Commands

Switch Set	Code Names	Port Names	I/O Names	Cmd Request	Trace Values
^n Null	^b Buffer	^b Buffer	^b Buffer	$m Make	^e Event
^c Center	^c Center	^c Center	^c Center	^c Compile	^c Center
^d Draw	^d Deposit	^d Deposit	^d Deposit	^d Dump	^d Dump
^r Remove	^r Remove	^r Remove	^r Remove	^r Reload	^r Reload
^g Go for	^t Toggle	^t Toggle	^i Input	^k linK	^t Toggle
$^k Klear	$^k Klear	$^k Klear	^o Output	^l Load	^y displaY
	^y displaY	^y displaY		^g Go	^l Load
				^n conNections	^g Go
				^\ interrupt	^\ interrupt
				^b liBraries	
				^i Interface	
				^w Write	

Execute Commands

alias <is> <name1> <name2>
bind <stream> <file>
continue [<condition>] [trace|notrace]
copy <from structure>, <to structure>
display<type>
flushbuffers
help
if <i> <j> <var> relop <val> skip <l>
log <file>
peek <is> <name>[<file>]
poke <is> <name> <value>
replace <old string> <new string>
reset [<phase>]
run[<phase>] [<condition>] [trace|notrace]
script <file>
set <param><value>
shell <UNIX cmd>
test <predicate>
unbind
unalias <is> <name>

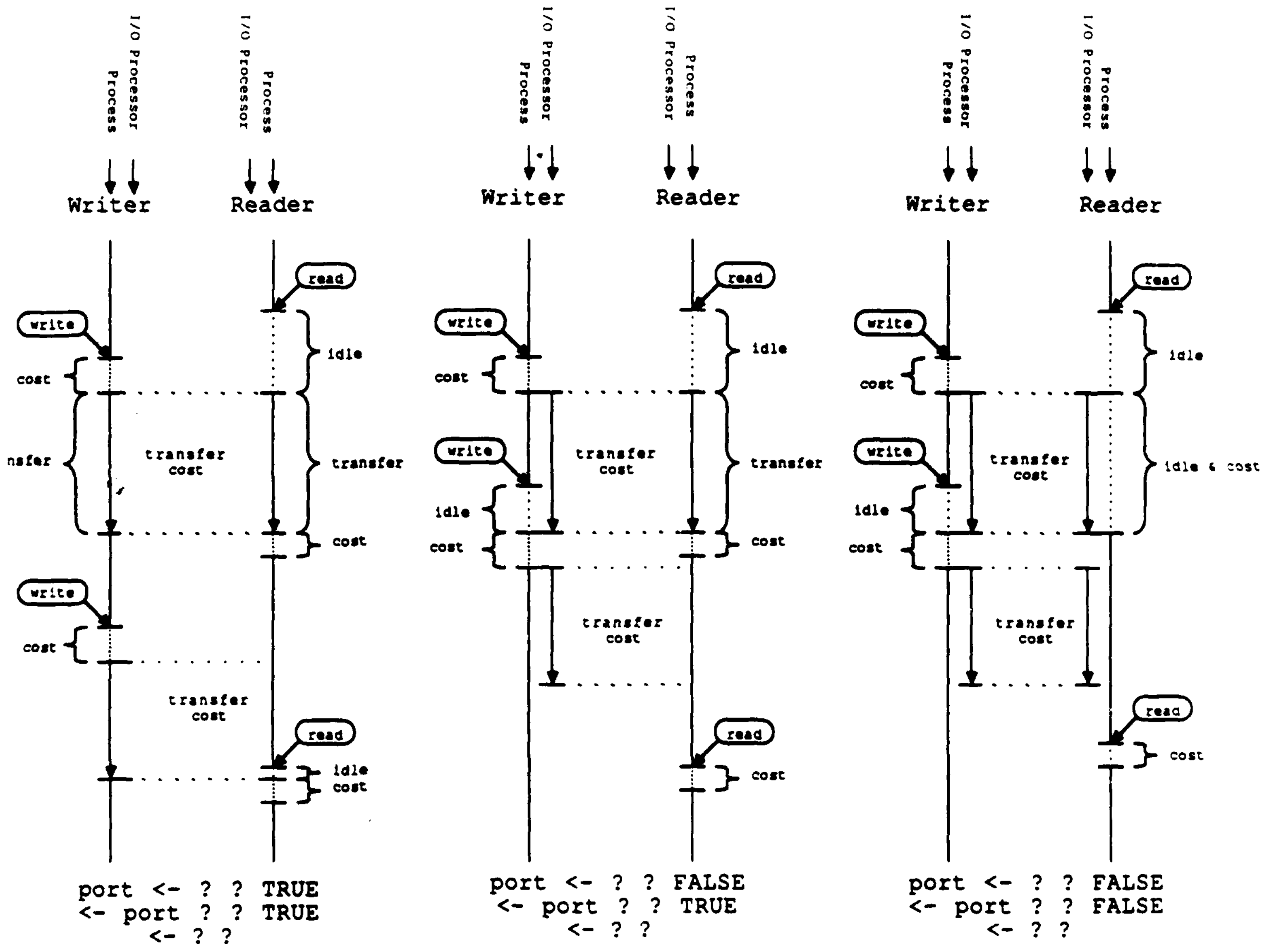


Figure 9
Poker time model showing effect of varying port I/O parameters.