The Anatomy of the BACI PCODE File

Bill Bynum/Tracy Camp
College of William and Mary/Colorado School of Mines

November 5, 2002

Contents

Chapter 1

Introduction

This document describes the form and function of the components of the BACI executable files, the files with `.pco` and `.pob` suffixes. These are the files produced by the BACI C−− and Concurrent Pascal compilers. The `.pob` file is produced by the BACI compilers when the user invokes the `-c` option at compilation, which induces the compilers to relax the requirement that the file produced by the compilation contain a main procedure. This option was added in 1996 to prepare for separate compilation. The form of the `.pob` file is identical to that of the `.pco` file. Both file types will be referred to in this manual as a "PCODE file".

The PCODE file consists of seven tables: the PCODE Table, containing the instructions executed by the BACI PCODE interpreter; the Identifier Table, containing the symbols used by the program; the Block Table, describing the static blocks of the program; the Array Table, describing the array variables of the program; the String Table, containing the strings used by the program; the Input File Table, listing the source files used to create the PCODE file; and the PCODE Debugging Information Table, used by the BACI PCODE interpreter to associate the source code line numbers with PCODE addresses.

These seven sections of the PCODE file will be described in the context of the C−− example shown in Figure 1.1 on page 3.

The entire PCODE file for this example is included in its entirety in Appendix A. We will discuss each of the seven tables of the PCODE file in the chapters that follow.

```
BACI System: C-- to PCODE Compiler, 13:40   7 Jun 2000
Source file: example.cm  Fri Jun 30 10:13:48 2000
 line  pc
   1    0  // Source file to produce a PCODE file for use as an example
   2    0
   3    0
   4    0  int i = 17;
   5    0
   6    0  string[20] s;
   7    0
   8    0  int q[10];        // a simple array
   9    0
  10    0  char r[3][6][4];  // a more complicated array
  11    0
  12    0  #include "example.inc"  // bring in the procedure called "included_proc"
>   1    0  // a proc for inclusion into the example program
>   2    0
>   3    0  void included_proc(int& i,int ival, int ix, int qixval)
>   4    0  {
>   5    0     int p = 3;
>   6    3     int w = 72;
>   7    6     cout << "p = " << p << "  w = " << w << endl;
>   8   13     i = ival;
>   9   16     q[ix] = qixval;
>  10   21  }
Returning to file example.cm
  13   22
  14   22  int main()
  15   23  {
  16   23     stringCopy(s,"a string for you");
  17   25     cout << "s = \"" << s << "\"" << endl;
  18   30     included_proc(i,44,4,77);
  19   37     cout << "i = " << i << "  q[4] = " << q[4] << endl;
  20   47     r[1][4][2] = 'A';
  21   56     r[0][1][3] = 'B';
  22   65     cout << "r[1][4][2]  = '" << r[1][4][2]
  23   73          << "'   r[0][4][3] =  " << r[0][1][3] << "'\n";
  24   86  }
```

Figure 1.1: Example C−− Source Program

Chapter 2

The PCODE Table

The PCODE Table is shown below in abbreviated form in Figure 2.1. The PCODE Table consists of the 91 PCODE instructions generated by the C−− compiler for the example source program.

The first two lines of Figure 2.1 are the "header lines" included in the PCODE file by the compiler to show the compiler version and the name of the primary source file.

The third line of Figure 2.1 gives the first (0) and last indices (90) of the entries in the PCODE table.

The fourth line of Figure 2.1 names the columns that follow. The `lc` label refers to the location counter value for each PCODE instruction.

Each PCODE instruction consists of an instruction opcode and an `x` and `y` operands. The exact meaning of each instruction is not important to understanding the format of the PCODE Table. More information about the PCODE for this example is contained in the disassembled version of the program PCODE included in Appendix B.

```
BACI System: C-- to PCODE Compiler, 13:40   7 Jun 2000
Source file: example.cm  Fri Jun 30 10:13:48 2000
0  90                PCODE table
 lc   f      x    y
  0   0    1    9
  1  24    0    3
  2  38    0    0
  3   0    1   10
  4  24    0   72
      ...
 84  29    0    3
 85  28    0   87
 86  31    0    0
 87   0    0    0
 88  24    0   17
 89  38    0    0
 90  81    0    0
```

Figure 2.1: The PCODE Table

Chapter 3

The Identifier Table

The Identifier Table for the example source program is shown in Figure 3.1.

```
1  13            IDENTIFIER table
index   identifier    link  obj  type  ref normal   lev   adr   mon atomic
   1  ++-outer-++       0    7     0    0     1     0    87     0     0
   2  i                 1    1     1    0     1     0     0     0     0
   3  s                 2    1    11    6     1     0     1     0     0
   4  q                 3    1     7    0     1     0     7     0     0
   5  r                 4    1     7    1     1     0    17     0     0
   6  included_pro      5    3     0    1     1     0     0     0     0
   7  i                 0    1     1    0     0     1     5     0     0
   8  ival              7    1     1    0     1     1     6     0     0
   9  ix                8    1     1    0     1     1     7     0     0
  10  qixval            9    1     1    0     1     1     8     0     0
  11  p                10    1     1    0     1     1     9     0     0
  12  w                11    1     1    0     1     1    10     0     0
  13  main              6    6     0    2     1     0    22     0     0
```

Figure 3.1: Identifier Table for Example Program

The first line of the Identifier Table gives the indices that the entries of the table will occupy in the interpreter's `tab` array used at runtime. In this case, the index for the `++-outer-++` symbol in the `tab` array will be `1`, and the index for the `main` symbol in the `tab` array will be `13`. Entry 0 of the Identifier Table is used by the two BACI compilers as a "sentinel" location for searching the Identifier Table and is not used by the BACI PCODE interpreter.

3.1 The `identifier` Field of the Identifier Table

The `identifier` field of an entry in the Identifier Table is the name of the given symbol, truncated to 12 characters. As you can see, the name of the `included_proc` function has been truncated to `included_pro`.

3.2 The `link` Field of the Identifier Table

For each program block (procedure or function), the symbols accessible in that block are maintained in a linked list using the `link` field of the Identifier Table. This linked list is used by the BACI compilers and the PCODE intererpeter for searching the Identifier Table. The compilers must search the Identifier Table during compilation each time another symbol is seen in the input. The PCODE interpreter must the Identifier

Table when presenting runtime debugging information, such as the values of the variables of a procedure or function.

For a given block, the index in the `tab` array of the last symbol in the block is given by the `last` field of the entry for that block in the Block Table (see Section 4.1 on page 12 below). Starting at this entry of the `tab` array and using the `link` field to obtain the `tab` index of the next element of the list will traverse all of the symbols defined in the given block. The value of the `link` field for the last element in the linked list is zero.

As an example, the `last` field of the entry of the Block Table corresponding to the `include_proc` block has value 12 (see Section 4.1), the index of the local variable `w`. The value of the `link` field for the `w` entry is 11, the index of the `tab` entry for the `p` local variable (that is, `tab[12].link` is `11`). The value of the `link` field for the `p` entry is 10, the index of the `tab` entry for the `qixval` parameter. The value of the `link` field for the `qixval` entry is 9, the index of the `tab` entry for the `ix` parameter. The value of the `link` field for the `ix` entry is 8, the index of the `tab` entry for the `ival` parameter. The value of the `link` field for the `ival` entry is 7, the index of the `tab` entry for the `i` parameter. Finally, the value of the `link` field for the `i` entry is 0, indicating that there are no other symbols in this block. The `included_pro` symbol belongs to the `main` block, and thus appears in the linked list for the `main` block.

3.3 The `obj` Field of the Identifier Table

The `obj` field of the Identifier Table contains the value of a C enumerated type used to characterize what sort of object the symbol is. This field is used heavily by both compilers. The BACI PCODE interpreter uses the field to determine the stack addresses of program variables at runtime. The possible values are shown below in Figure 3.2.

value	name in enumerated type	description
0	`constant`	program constant
1	`variable`	program variable
2	`type`	user-defined type
3	`procedure`	void function
4	`function`	non-void function
5	`monitor`	Hoare monitor
6	`mainproc`	main procedure
7	`outerblock`	startup initialization code
8	`ext_procedure`	external void function
9	`ext_function`	external non-void function
10	`ext_monitor`	external Hoare monitor
11	`ext_variable`	external program variable

Figure 3.2: Possible Values of the `obj` Field of the Identifier Table

3.4 The `type` Field of the Identifier Table

The `type` field of the Identifier Table contains the value of a C enumerated type used to characterize the type of a symbol. This field is used heavily for type-checking by both compilers and the PCODE interpreter. The possible values are shown in Figure 3.3 on page 7.

value	name in enumerated type	description
0	notyp	untyped
1	ints	integer
2	bools	boolean (used only in Pascal)
3	chars	character
4	bsems	binary semaphore
5	sems	semaphore
6	conds	condition of a Hoare monitor
7	arrays	array
10	rawstrings	a quoted string (e.g., “xyz”)
11	strings	string variable

Figure 3.3: Possible Values of the type Field of the Identifier Table

3.5 The ref Field of the Identifier Table

The ref field of the Identifier Table is used as an index into one of the tables of the PCODE file. The table used depends on the type or obj field of the symbol.

For an array symbol (type field of 7), the ref field is the index of the symbol’s entry in the Array Table. For example, the value of the ref field for the q symbol (entry 4 in the Identifier Table) is 0, the index of the q array in the Array Table. The value of the ref field for the r symbol (entry 5 in the Identifier Table) is 1, the index of the r array in the Array Table.

For a symbol representing a procedure or function (an obj field of 3, 4, 6, 7, 8, or 9), the ref field is the index of the symbol’s entry in the Block Table. For example, the ref field for the outer block (entry 1 in the Identifier Table) is 0, the index of the entry for the outer block in the Block Table. The ref field for the included_proc procedure (entry 6 in the Identifier Table) is 1, the index of the entry for included_proc in the Block Table. The ref field for the main procedure (entry 13 in the Identifier Table) is 2, the index of the entry for main in the Block Table.

For a symbol representing a string variable (a type field of 11), the ref field is the size of the string, expressed as the number of 4-byte words required by the string on the PCODE interpreter’s runtime stack. The s symbol (at index 3 of the Identifier Table) was declared to be a string of length 20 (i.e., its C−− type is string[20]), so it will require a minimum of 21 bytes on the interpreter’s runtime stack (the extra byte is the space for the string’s null terminator byte 0x00). To store 21 bytes on the interpreter’s stack , 6 4-byte stack words will be required. Thus, the value of the ref field for the symbol s is 6.

The ref field is unused for all other members of the Identifier Table.

3.6 The normal Field of the Identifier Table

This normal field of the Identifier Table applies only to symbols in the table that are formal parameters of a procedure or function. The value of the normal field is 0 for a pass-by-reference parameter. The value of the normal field is 1 for all other entries in the Identifier Table. For example, the value of the normal field for the formal parameter i of the included_proc procedure (entry 7 of the Identifier Table) is 0, reflecting the fact that i is a reference parameter of the procedure. The value of the normal field for the other formal parameters of the included_proc procedure (entries 8, 9, and 10 in the Identifier Table) is 1, indicating that these parameters are pass-by-value parameters.

The field is used by the BACI compilers to generate appropriate PCODE for referencing procedure or function parameters. The BACI PCODE interpreter ignores the normal field.

3.7 The `lev` Field of the Identifier Table

The `lev` field of a symbol gives the static level of the program at which the symbol was declared. For example, `lev` fields of the global variables `i`, `s`, `q`, and `r` and the functions `included_proc` and `main` are 0, since these symbols were declared at the global level (level 0). The `lev` fields for the variables of the `included_proc` procedure, `i`, `ival`, `ix`, `qixval`, `p`, and `w` are 1, since these variables are declared in the `include_proc` block (level 1).

Global variables, procedures, functions, and monitors are at level 0. Monitor procedures and functions are at level 1, but are linked into global level (level 0) through their `link` fields (see Section 3.2). This makes the monitor procs visible at the global level during compilation. Monitor variables are at level 1. Local variables of monitor procedures or functions are at level 2.

The parameters of a procedure or function in C−− or Pascal are at the same level as the procedure or function, so that they will be visible in the "outer" scope, just like the name of the procedure or function is. The level of the wocal variables of a procedure or function is one higher than the level of the procedure or function.

For a `.pob` file produced by C−−, the `lev` field of all symbols is either 0, 1, or 2, since nesting is not allowed. Because the Concurrent Pascal compiler allows static nesting up to seven levels, symbols with a `lev` field larger than 1 can occur.

Both compilers and the PCODE interpreter use the `lev` field for accessing program variables. See Section 3.8 for a discussion of variable referencing using the `lev` and `adr` fields.

3.8 The `adr` Field of the Identifier Table

The `adr` field of the Identifier Table is relevant for the symbols in the identifier table that are either program blocks (procedures or functions) or program variables.

For a program block, the `adr` field is the address (that is, the index in the Code Table) of the entry point of the block. For example, the `adr` field of the `++-outer-++` symbol (the outer block created by the compiler to hold initialization code, element 1 of the Identifier Table) is 87, the entry point of the outer block. The `adr` field of the `include_pro` symbol is 0, the entry point of the procedure. The `adr` field of the `main` symbol is 22, the entry point of the main program.

For a program variable, the BACI PCODE interpreter (and the compilers, too) use the `lev` and `adr` fields of the Identifier table to determine the runtime stack address of the variable. The PCODE interpreter uses a display to enforce static program structure at runtime. The `lev` entry of the display is the offset to the stack location of the variable from the base of the stack frame of the current program block at static level `lev`. Stated another way, the runtime stack address of a program variable is simply the sum

`display[lev] + adr`.

The program block at static level 0 is always the outer (global) scope, and the base of the stack frame for the global scope is always stack location 0 (that is, `display[0]` = 0). Thus, for a global variable (a symbol in the Identifier Table whose `lev` field is 0), the stack address of the variable is simply the `adr` field of the symbol.

For example, global variable `i` (at index 2 of the Identifier Table) is at stack location 0, global variable `s` (at index 3 of the Identifier Table) is at stack location 1, global variable `q` (at index 4 of the Identifier Table) is at stack location 7, and global variable `r` (at index 5 of the Identifier Table) is at stack location 17.

The stack address of the actual parameter corresponding to the formal parameter `i` of the `included_proc` procedure is determined by adding 5 (its `adr` field) to the base of the stack frame of the activation of the procedure (the value in `display[1]`). Since the parameter `i` is a reference parameter, the value stored at this stack location will be the (stack) address of the parameter, rather than the value of the actual parameter. The values of Since the actual parameters `ival`, `ix`, and `qixval` are passed by value, the values of these

parameters (and not the addresses of these parameters) will be located 6, 7, and 8 words, respectively, above the base of the stack frame of the activation of the procedure. The values of the local variables `p` and `w` will be located 8 and 9 words, respectively, above the base of the stack frame of the activation of the procedure.

3.9 The `mon` Field of the Identifier Table

The `mon` field of the Identifier Table is used by the BACI compilers and PCODE interpreter to implement the semantics of calls to procedures and functions of a Hoare monitor. The `mon` field of a procedure or function of a Hoare monitor contains the index in the Identifier Table of the Hoare monitor in which the procedure or function was declared. The `mon` field for all other entries in the Identifier Table is zero.

The PCODE interpreter uses the `mon` field to enforce the mutual exclusion required of calls to monitor procedures or functions. Only one call to a procedure or function of a monitor can be active at any time – all other simultaneously active threads of execution in the monitor must be suspended and subsequently revived when execution is again possible.

A short example to illustrate the use of the `mon` field is shown in Figure 3.4. This example contains a Hoare monitor to manage allocation and release of a single resource.

```
BACI System: C-- to PCODE Compiler, 13:40   7 Jun 2000
Source file: mon-alloc.cm  Fri Jun 30 14:40:53 2000
 line  pc
   1    0     // Hoare monitor to allocate a single resource
   2    0     // See Operating System Concepts, 4th ed, Silberschatz & Galvin p197
   3    0
   4    0  monitor resource_alloc {
   5    0     int busy;             // 1 if resource is busy, 0 otherwise
   6    0     condition free;       // condition on which to wait until resource is free
   7    0
   8    0     void acquire()
   9    1     {
  10    1        if (busy) {        // if resource is busy, then
  11    3           waitc(free);    //  .. wait until resource is free
  12    6           busy = 1;       // just awakened, mark the resource as busy
  13    9        }
  14    9     }
  15   11
  16   11     void release()
  17   12     {
  18   12        busy = 0;          // resource no longer needed
  19   15        signalc(free);     // awake someone waiting (possibly)
  20   17     }
  21   19
  22   19     init { busy = 0; }    // initially the resource is free
  23   23
  24   23  }  // resource_alloc monitor
```

Figure 3.4: A Hoare Monitor in C−− for Allocating a Single Resource

The Identifier and Block Tables for this example are shown in Figure 3.5 on page 10. The `resource_alloc` monitor has index 2 in the Identifier Table. The `mon` fields of the `acquire` and `release` procedures of the monitor at entries 6 and 7 of the Identifier Table have value 2 to indicate that these procedures belong to the `resource_alloc` monitor. The `mon` fields of all other entries of the Identifier Table are zero.

```
1  6            IDENTIFIER table
index   identifier    link  obj  type  ref normal   lev   adr   mon atomic
   1  ++-outer-++       0     7     0     0     1     0    23     0     0
   2  resource_all      1     5     0     1     1     0    19     0     0
   3  busy              0     1     1     0     1     1     5     0     0
   4  free              3     1     6     0     1     1     6     0     0
   5  acquire           2     3     0     2     1     1     0     2     0
   6  release           5     3     0     3     1     1    11     2     0
0  3         BLOCK table
index last lastpar psize vsize
   0    6       0    0    0
   1    4       2    5    7
   2    0       5    5    5
   3    0       6    5    5
```

Figure 3.5: Block and Identifier Tables for a Hoare Monitor in C−−

3.10 The `atomic` Field of the Identifier Table

The `atomic` field of a symbol in the Identifier Table is 1 only if the symbol corresponds to a procedure or function that has been declared as `atomic` in the program source. The PCODE interpreter uses the `atomic` field of the Identifier Table to ensure that a procedure or function so declared is non-preemptible when the program is executed.

A short example to illustrate the use of the `atomic` field is shown in Figure 3.6. This example contains a `compare_and_swap` synchronization primitive.

```
BACI System: C-- to PCODE Compiler, 13:40   7 Jun 2000
Source file: comp-swap.cm  Fri Jun 30 15:26:58 2000
 line  pc
   1    0  // define a compare_and_swap synchronization primitive
   2    0  // if 'w' and 'old' are the same, then store 'new'
   3    0  // at the reference parameter 'w' and return 1;
   4    0  // otherwise return 0.
   5    0  atomic int compare_and_swap(int& w, int old, int new)
   6    0  {
   7    0     if (w == old)
   8    4     {
   9    4        w = new;
  10    7        return 1;
  11   11     }
  12   11     else
  13   12     {
  14   12        return 0;
  15   16     }
  16   16  }  // compare_and_swap
```

Figure 3.6: An Atomic Compare-and-Swap Synchronization Primitive in C−−

The Identifier and Block Tables for this example are shown in Figure 3.7 on page 11.

The `atomic` field of the `compare_and_swap` symbol (entry 2) of the Identifier Table is 1 to mark the procedure to the PCODE interpreter as non-preemptible. Each time this function is called, all of its PCODE instructions will be executed, from the entry point of the call to the return, with no possibility of an interruption by a context switch.

```
1  5            IDENTIFIER table
index   identifier    link  obj  type  ref normal   lev   adr   mon atomic
   1  ++-outer-++       0     7     0     0     1     0    17     0     0
   2  compare_and_      1     4     1     1     1     0     0     0     1
   3  w                 0     1     1     0     0     1     5     0     0
   4  old               3     1     1     0     1     1     6     0     0
   5  new               4     1     1     0     1     1     7     0     0
0  1         BLOCK table
index last lastpar psize vsize
   0    2       0    0    0
   1    5       5    8    8
```

Figure 3.7: Block and Identifier Tables for an Atomic Compare-and-Swap Function

Chapter 4

Block Table

The Block Table of the PCODE file contains the information needed by the compilers and the PCODE interpreter to describe the static blocks of the program.

The Block Table for the main example source program is shown in Figure 4.1.

```
0  2          BLOCK table
index last lastpar psize vsize
   0   13      13    0   89
   1   12      10    9   11
   2   13      13    5    5
```

Figure 4.1: Block Table for Example Program

The first line indicates that 0 and 2 the smallest and largest indices, respectively, used in the Block Table. As we know from examining the `ref` fields of the entries in the Identifier Table in Section 3.5, entry 0 of the Block Table corresponds to the global (outer) block, entry 1 of the Block Table corresponds to the `included_proc` block, and entry 2 corresponds to the `main` block.

4.1 The `last` Fields of the Block Table

The `last` field of the Block Table was previously mentioned in Section 3.2 on page 5 when discussing the `link` field of the Identifier Table. As previously stated, for a given entry of the Block Table, the `last` field of the Block Table contains the index in the Identifier Table of the last symbol of the given block.

The `last` fields of the outer block and the main block (entries 0 and 2 of the Block Table) have the same value: namely 13, the index in the Identifier Table of the `main` symbol. The `last` field of the `included_proc` block (entry 1 of the Block Table) is 12, the index in the Identifier Table of the variable `w` defined in the procedure.

4.2 The `lastpar` Field of the Block Table

The `lastpar` field of the Block Table contains the index in the Identifier Table of the last parameter of the block. This field is used by the PCODE interpreter to create an activation record for the block when the block is called.

As with the `last` fields, the `lastpar` fields of the outer block and the main block are again 13, the index in the Identifier Table of the `main` symbol. For the main block, this makes sense, but for the outer block it doesn't, since the outer block has no parameters. For this reason, the `lastpar` field for the outer block is

used to hold the index in the `tab` table of the main proc. The PCODE interpreter uses this field to locate the entry point of the program at execution time.

The `lastpar` field of the `included_proc` block (entry 1 of the Block Table) is 10, the index in the Identifier Table of the right-most parameter in the `included_proc` function prototype, the `qixval` parameter. In the PCODE file for an example program in Appendix A, the value of `btab[0].laspar` (the `lastpar` field of the outer block) is 13, the index of `main` in the `tab` array.

4.3 The `psize` and `vsize` Fields of the Block Table

The `psize` and `vsize` fields of describe two sizes (in 4-byte stack words) associated with the stack frame of the block. Each BACI stack frame contains five 4-byte words of linkage information in addition to the storage space for the parameters and local variables of the block. The `psize` field is the sum the space for the linkage information (5 words) plus the size (in 4-byte words) of the space needed for the parameters of the block. The `vsize` field is the sum the space for the linkage information (5 words), the size (in 4-byte words) of the space needed for the parameters of the block, and the size (in 4-byte words) of the space needed for local variables of the block; that is, `vsize` is `psize` plus the size (in 4-byte words) of the space needed for local variables of the block.

Logically, the `psize` and `vsize` fields for the outer block and main block have no meaning, since neither of these blocks is endowed with a stack frame. The `psize` field of both blocks can safely be ignored. The `vsize` of the main block can also be ignored. However, the `vsize` field of the outer block (always entry 0 of the Block Table) is used by the interpreter to retrieve the amount of space (in 4-byte stack words) to allocate on the stack for the global variables of the program.

The `vsize` field of entry 0 of the Block Table shown in Figure 4.1 on page 12 is 89. This number is indeed the size of the global variable area of the example program. The global integer `i` occupies 1 word of stack space. As discussed in Section 3.5, the `s` string occupies 6 stack words. As we will see in Section 5.5, the `q` array requires 10 words of stack and the `r` array requires 72 stack words. The sum of these sizes is $1+6+10+72=89$, the number of words needed for the global variables of the example program.

Chapter 5

The Array Table

The Array Table for the main example source program is shown in Figure 5.1. The first line indicates that 0 and 3 the smallest and largest indices, respectively, used in the Array Table.

```
0  3           ARRAY table
index inxtype eltyp elref  low  high  elsize  size
   0     1     1      0     0     9     1    10
   1     1     7      2     0     2    24    72
   2     1     7      3     0     5     4    24
   3     1     3      0     0     3     1     4
```

Figure 5.1: Array Table for Example Program

As mentioned in Section 3.5, the `ref` field of an entry in the Identifier Table that corresponds to an array variable is the variable's index in the Array Table. According the Identifier Table shown in Figure 3.1 on page 5, entry 0 of the Array Table describes the `q` array (this is the value of the `ref` field of entry 4 (`q`'s entry) in the Identifier Table), and entry 1 of the Array Table describes the `r` array (this is the value of the `ref` field of entry 5 (`r`'s entry) in the Identifier Table).

5.1 The `elref` Field of the Array Table

Each dimension of each array is described by an entry in the Array Table. The declaration of a multidimensional array, like `r` in our running example:

```
char r[3][6][4];
```

is treated by C−− compiler as if the array were declared in three separate steps:

```
typedef char dim1[4];
typedef dim1 dim2[6];
dim2 r[3];
```

Thus, in this example, the `r` array occupies 3 entries of the Array Table. Entry 1 (as indicated by the `ref` field of the entry for `r` in the Identifier Table) corresponds to the `dim2 r[3]` declaration above. The `elref` field of the Array Table is used to refer to the entry number in the Array Table occupied by an "anonymous" array type, like the types created by the `typedef` declarations for `dim1` and `dim2` above. The `elref` field of entry 1 of the Array Table (the entry for the `r` array) has value 2, the entry corresponding to the `typedef dim1 dim2[6]` declaration. The `elref` field of entry 2 of the Array Table (the entry for the `dim2` dimension) has value 3, the entry corresponding to the `typedef char dim1[4]` declaration.

The machinery of the declaration of a multidimensional array guarantees that no "anonymous" array dimension can occupy entry 0 of the Array Table. Thus, an `elref` field of 0 for an entry of the Array Table indicates unambiguously that the type of the elements of the corresponding array is one of the built-in types and does not refer to the Array Table.

5.2 The `inxtype` Field of the Array Table

The `inxtype` field of the Array Table lists the type (as a member of the C enumerated type used in the `type` field of the Identifier Table – see Figure 3.3 on page 7 for a listing of the different possible values) used by the index of the array. In this example, the `inxtype` field of every entry of the Array Table has value 1, indicating that all indices are of integer type. This will be true of all examples produced by the C−− compiler.

The BACI Pascal compiler has two built-in discrete types, integer and boolean (TRUE, FALSE). In PCODE files created by the BACI Pascal compiler, the `inxtype` field in the Array Table can contain the boolean type (having value 2). For example, the following Pascal declaration:

```
VAR
   x : ARRAY [TRUE..FALSE] OF INTEGER;
```

would generate an entry in the Array Table whose `inxtype` field has the value 2 (for the boolean type).

At the present time, neither BACI compiler allows user-defined enumerated types, so the appearance of user-defined type in the `inxtype` field is not currently possible.

5.3 The `eltyp` Field of the Array Table

The `eltyp` field of the Array Table lists the type of the elements of the array (as a member of the C enumerated type used in the `type` field of the Identifier Table – see Figure 3.3 on page 7 for a listing of the different possible values).

For a multi-dimensional array like the `r` variable in our running example, the `eltyp` field of the "innermost" or "last" or "rightmost" dimension of the array will have the enumerated type value for type of the elements of the array. The `eltyp` field of the Array Table entries of all other ("outer") dimensions of the array will have value 7 (the `arrays` type). In the case of the three-dimensional array `r` in our running example, the entry for the inner array dimension (the `dim1` dimension, element 3 in the Array Table of Figure 5.1 on page 14) has a value 3 (for type `char`) in its `eltyp` field, while the "outer dimensions", the `dim2` and `char` dimensions in entries 2 and 1 of the Array Table, have `eltyp` fields of 7, the value for an array type.

5.4 The `low` and `high` Fields of the Array Table

The `low` and `high` fields of the Array Table describe the smallest and largest values that the index of the array will have. Since the `q` array, represented by element 0 of the Array Table, is declared to be of size 10, its array index will range from 0 to 9, the `low` and `high` values of element 0 in the Array Table. The "inner" dimension of the `r` array, the so-called `char` dimension represented by element 1 of the Array Table, has 3 entries, so its array index will range from 0 to 2, yielding the `low` and `high` fields of element 1. The entries 2 and 3 of the Array Table corresponding to the `dim2` and `dim1` dimensions, respectively, of the `r` array, are similarly determined to have `low` and `high` values of 0 and 5, and 0 and 3, respectively.

Since each array in C begins at 0, the `low` value of any array declared in a C−− program will be 0. However, Pascal allows arbitrary integer subranges for array indices. The following Pascal declaration

```
VAR
   w : ARRAY [-11..-3] OF CHAR;
```

will produce a corresponding entry in the Array Table having a `low` field of -11 and a `high` field of -3.

5.5 The `elsize` and `size` Fields of the Array Table

Recall from Section 5.1 that the declaration of a multidimensional array, like `r` in this example:

```
char r[3][6][4];
```

is treated by C−− compiler as if the array were declared in three separate steps:

```
typedef char dim1[4];
typedef dim1 dim2[6];
dim2 r[3];
```

As noted in Section 5.1, the `char` dimension of the array (entry 3 of the Array Table) has an element size of one 4-byte word (the BACI PCODE interpreter uses an entire 4-byte stack word to store a single one-byte character). There are 4 elements in inner `char` dimension of the array, so the size of a member of this array type is 4. In fact, the following relationship holds between the `size`, `elsize`, `low` and `high` fields of any entry of the Array Table:

$$\texttt{size} = \texttt{elsize} \cdot (1 + \texttt{high} - \texttt{low})$$

The `elsize` value of entry 2 of the Array Table corresponding to the `typedef dim1 dim2[6]` declaration is 4, the value of the `size` field of entry 3 corresponding to the `typedef char dim1[4]` declaration. Since there are 6 elements in the `dim2` "anonymous" array, the `size` field of this entry of the Array Table will be 24.

In like manner, the `elsize` value of entry 1 of the Array Table corresponding to the `dim2 r[3]` declaration is 24, the `size` field of entry 2 of the Array Table. Since there are 3 elements in this dimension of the array, the `size` field for entry 1 of the Array Table will be $3 \cdot 24 = 72$.

Chapter 6

The String Table

The String Table for the main example source program is shown in Figure 6.1. The first line indicates that the smallest index used in the string table is 0. The second integer, 90 in this case, is **the number of characters (or bytes) in the string table**. This usage of the second integer is different from the usage for any other table in the PCODE file. For any other table in the PCODE file, the second integer is the largest index of any element in the given table. In the string table, the second integer on the initial line of the table is **one more** than the largest index used in the String Table.

The third integer on the initial line of the String Table, 60 in this case, is the number of characters of the String Table stored on each line of the PCODE file. In Figure 6.1, the null terminator character (the byte `0x00`) is represented by the two characters `^@`.

```
0  90  60                  STRING table
p = ^@  w = ^@a string for you^@s = "^@"^@i = ^@  q[4] = ^@r[1][4][
2]  = '^@'   r[0][4][3] =  ^@'
^@
```

Figure 6.1: String Table for Example Program

With this information, one can verify that the first line of the String Table does indeed correctly contain 60 characters. The String Table appears to span three lines of the PCODE file. However, the BACI PCODE interpreter will read the newline at the end of the second line of the String Table as a part of the String Table. To put it another way, after reading the 60 characters on the first line of the String Table, the BACI PCODE interpreter will read an additional 30 characters of the PCODE file to obtain the 90 characters that the initial line has indicated are in the String Table. By counting characters, one can verify that the 29-th character of the second line is the newline at the end of the second line, so that the 30-th character stored into the String Table by the interpreter will be the null terminator character that it reads after the newline. Recall that the last string of the source program (see Figure 1.1 on page 3) contains a newline character (`\n`).

Incidentally, both BACI compilers check to see whether a newly encountered string already appears in the String Table before storing it. Each user-defined raw string will appear in the String Table exactly once.

Chapter 7

The Input File Table

The Input File Table (or Array) is simply the list of the names of the input files that were used by the compiler to produce the PCODE file. In the case of our running example, the Input File Table is shown in shown in Figure 7.1. The first line indicates that 0 and 1 the smallest and largest indices, respectively, used in the Input File Table.

```
0  1           Input File array
index parent      file name
   0    -1   example.cm
   1     0   example.inc
```

Figure 7.1: Input File Table for Example Program

The `parent` field of an entry of the Input File Table contains the index of the entry in the Input File Table that included the current entry. The `file name` field of an entry of the Input File Table is the name of the file.

In Figure 7.1, the `parent` field of entry 0 of the Input File Table is -1, because the file corresponding to this entry was the first file that the compiler encountered.nd has no parent The name of this file was `example.cm`.

The `parent` field of entry 1 of the Input File Table corresponding to the file named `example.inc` is 0, the index of the entry in the Input File Table that included the `example.inc` file, namely the `example.cm` file.

Chapter 8

The PCODE Debugging Information Table

The PCODE Debugging Information Table contains the information that the PCODE interpreter needs to correspond addresses in the PCODE to source lines in the one or more source files used by the compiler to produce the PCODE file.

Figure 8.1 shows the PCODE Debugging Information Table for our main example program. The first line of the table gives the smallest and largest indices used in the table, 0 and 18 in this case.

The `flineno` field of an entry of the PCODE Debugging Information Table is the largest line number of the input file given by the `findex` field for which the value of the location counter during compilation had the value shown in the `lc` field. For example, the third line of the table

```
   0     0    12
```

indicates that the location counter value, `lc`, was 0 for lines 1 through 12, inclusive, of the `example.cm` file.

```
0  18           PCODE debugging information
   lc findex flineno
    0     0    12
    0     1     5
    3     1     6
    6     1     7
   13     1     8
   16     1     9
   21     1    10
   22     1   -10
   22     0    14
   23     0    16
   25     0    17
   30     0    18
   37     0    19
   47     0    20
   56     0    21
   65     0    22
   73     0    23
   86     0    24
   91     0   -24
```

Figure 8.1: PCODE Debugging Information Table for Example Program

The next line

```
   0     1     5
```

indicates that `lc` remained at 0 for the first five lines of the included file, `example.inc`.

The following line

```
3     1     6
```

indicates that the `lc` variable increased to 3 at line 6 of the included file, `example.inc`.

The rest of the PCODE Debugging Information Table can be explained similarly. A negative `flineno` field indicates that the corresponding file was closed after that line was compiled.

The BACI PCODE disassembler, `badis`, uses the PCODE Debugging Information Table to produce its output. You may find it instructive to compare the PCODE Debugging Information Table shown in Figure 8.1 with the disassembly listing contained in Appendix B to see how this information was used.

Appendix A

Entire PCODE File for Example Program

This is the entire PCODE file for the example given in Section 1. In the String Table, the two characters ^@ have been substituted for the zero string termination byte 0x00 each time that it occurs. Other than this modification, the PCODE file shown is exactly as produced by the bacc compiler.

```
BACI System: C-- to PCODE Compiler, 13:40   7 Jun 2000
Source file: example.cm  Fri Jun 30 10:13:48 2000
0  90                PCODE table
 lc   f      x    y
  0   0    1    9
  1  24    0    3
  2  38    0    0
  3   0    1   10
  4  24    0   72
  5  38    0    0
  6  28    0    0
  7   1    1    9
  8  29    0    1
  9  28    0    5
 10   1    1   10
 11  29    0    1
 12  63    0    0
 13   1    1    5
 14   1    1    6
 15  38    0    0
 16   0    0    7
 17   1    1    7
 18  21    0    0
 19   1    1    8
 20  38    0    0
 21  32    0    0
 22  80    0   87
 23   0    0    1
 24 111    0   12
 25  28    0   29
 26   0    0    1
 27 110    0    0
 28  28    0   35
 29  63    0    0
 30  18    0    6
 31   0    0    0
 32  24    0   44
 33  24    0    4
 34  24    0   77
 35  19    0    8
```

```
 36   3    0    1
 37  28    0   37
 38   1    0    0
 39  29    0    1
 40  28    0   42
 41   0    0    7
 42  24    0    4
 43  21    0    0
 44  34    0    0
 45  29    0    1
 46  63    0    0
 47   0    0   17
 48  24    0    1
 49  21    0    1
 50  24    0    4
 51  21    0    2
 52  24    0    2
 53  21    0    3
 54  24    0   65
 55  38    0    0
 56   0    0   17
 57  24    0    0
 58  21    0    1
 59  24    0    1
 60  21    0    2
 61  24    0    3
 62  21    0    3
 63  24    0   66
 64  38    0    0
 65  28    0   52
 66   0    0   17
 67  24    0    1
 68  21    0    1
 69  24    0    4
 70  21    0    2
 71  24    0    2
 72  21    0    3
 73  34    0    0
 74  29    0    3
 75  28    0   68
 76   0    0   17
 77  24    0    0
 78  21    0    1
 79  24    0    1
 80  21    0    2
 81  24    0    3
 82  21    0    3
 83  34    0    0
 84  29    0    3
 85  28    0   87
 86  31    0    0
 87   0    0    0
 88  24    0   17
 89  38    0    0
 90  81    0    0
1  13            IDENTIFIER table
index   identifier    link  obj  type  ref normal   lev   adr   mon atomic
   1  ++-outer-++       0    7     0    0     1     0    87     0     0
   2  i                 1    1     1    0     1     0     0     0     0
   3  s                 2    1    11    6     1     0     1     0     0
   4  q                 3    1     7    0     1     0     7     0     0
```

```
   5  r                4     1     7     1     1     0    17     0     0
   6  included_pro     5     3     0     1     1     0     0     0     0
   7  i                0     1     1     0     0     1     5     0     0
   8  ival             7     1     1     0     1     1     6     0     0
   9  ix               8     1     1     0     1     1     7     0     0
  10  qixval           9     1     1     0     1     1     8     0     0
  11  p               10     1     1     0     1     1     9     0     0
  12  w               11     1     1     0     1     1    10     0     0
  13  main             6     6     0     2     1     0    22     0     0
0  2         BLOCK table
index last lastpar psize vsize
   0   13      13    0   89
   1   12      10    9   11
   2   13      13    5    5
0  3           ARRAY table
index inxtype eltyp elref  low  high  elsize  size
   0     1     1      0     0     9     1    10
   1     1     7      2     0     2    24    72
   2     1     7      3     0     5     4    24
   3     1     3      0     0     3     1     4
0  90  60                  STRING table
p = ^@  w = ^@a string for you^@s = "^@"^@i = ^@  q[4] = ^@r[1][4][
2]  = '^@'   r[0][4][3] =  ^@'
^@
0  1           Input File array
index parent      file name
   0    -1   example.cm
   1     0   example.inc
0  18           PCODE debugging information
   lc findex flineno
   0     0    12
   0     1     5
   3     1     6
   6     1     7
  13     1     8
  16     1     9
  21     1    10
  22     1   -10
  22     0    14
  23     0    16
  25     0    17
  30     0    18
  37     0    19
  47     0    20
  56     0    21
  65     0    22
  73     0    23
  86     0    24
  91     0   -24
```

Appendix B

Disassembled PCODE for Example Program

This is the disassembly of the PCODE of the example program produced by the BACI PCODE disassembler, `badis`.

```
BACI System: BenAri PCODE Disassembler, 13:40   7 Jun 2000
PCODE file: example.pco  Fri Jun 30 10:13:55 2000

BACI System: C-- to PCODE Compiler, 13:40   7 Jun 2000
Source file: example.cm  Fri Jun 30 10:13:48 2000
Reading from source file 'example.cm'

      1 // Source file to produce a PCODE file for use as an example
      2
      3
      4 int i = 17;
      5
      6 string[20] s;
      7
      8 int q[10];        // a simple array
      9
     10 char r[3][6][4];  // a more complicated array
     11
     12 #include "example.inc"  // bring in the procedure called "included_proc"

Reading from source file 'example.inc'

>      1 // a proc for inclusion into the example program
>      2
>      3 void included_proc(int& i,int ival, int ix, int qixval)
>      4 {
>      5    int p = 3;

  lc    f    x    y     PCODE
   0    0    1    9  LOAD_ADDR, push &p
   1   24    0    3  PUSH_LIT 3
   2   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)

>      6    int w = 72;

   3    0    1   10  LOAD_ADDR, push &w
   4   24    0   72  PUSH_LIT 72
   5   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
```

```
>      7    cout << "p = " << p << "  w = " << w << endl;

   6   28    0    0  WRITE_RAWSTRING stab[0] to stdout
   7    1    1    9  LOAD_VALUE, push p
   8   29    0    1  WRITE (int) s[t] to stdout, pop(1)
   9   28    0    5  WRITE_RAWSTRING stab[5] to stdout
  10    1    1   10  LOAD_VALUE, push w
  11   29    0    1  WRITE (int) s[t] to stdout, pop(1)
  12   63    0    0  WRITELN

>      8    i = ival;

  13    1    1    5  LOAD_VALUE, push i
  14    1    1    6  LOAD_VALUE, push ival
  15   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)

>      9    q[ix] = qixval;

  16    0    0    7  LOAD_ADDR, push &q
  17    1    1    7  LOAD_VALUE, push ix
  18   21    0    0  INDEX atab[0], pop(1)
  19    1    1    8  LOAD_VALUE, push qixval
  20   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)

>     10 }

  21   32    0    0  EXIT_PROC
Returning to file 'example.cm'

     13
     14 int main()

  22   80    0   87  SHORTCALL to 87, shortcall_reg = pc, pc = 87

     15 {
     16    stringCopy(s,"a string for you");

  23    0    0    1  LOAD_ADDR, push &s
  24  111    0   12  COPY_RAWSTRING from stab[12] to s[s[t]], pop(1)

     17    cout << "s = \"" << s << "\"" << endl;

  25   28    0   29  WRITE_RAWSTRING stab[29] to stdout
  26    0    0    1  LOAD_ADDR, push &s
  27  110    0    0  WRITE_STRING at s[s[t]] to stdout, pop(1)
  28   28    0   35  WRITE_RAWSTRING stab[35] to stdout
  29   63    0    0  WRITELN

     18    included_proc(i,44,4,77);

  30   18    0    6  MARKSTACK included_pro
  31    0    0    0  LOAD_ADDR, push &i
  32   24    0   44  PUSH_LIT 44
  33   24    0    4  PUSH_LIT 4
  34   24    0   77  PUSH_LIT 77
  35   19    0    8  CALL, psize-1 = 8
  36    3    0    1  UPDATE_DISPLAY from level 1 out to level 0

     19    cout << "i = " << i << "  q[4] = " << q[4] << endl;

  37   28    0   37  WRITE_RAWSTRING stab[37] to stdout
```

```
38    1    0    0  LOAD_VALUE, push i
39   29    0    1  WRITE (int) s[t] to stdout, pop(1)
40   28    0   42  WRITE_RAWSTRING stab[42] to stdout
41    0    0    7  LOAD_ADDR, push &q
42   24    0    4  PUSH_LIT 4
43   21    0    0  INDEX atab[0], pop(1)
44   34    0    0  VALUE_AT, s[t] = s[s[t]]
45   29    0    1  WRITE (int) s[t] to stdout, pop(1)
46   63    0    0  WRITELN

   20    r[1][4][2] = 'A';

47    0    0   17  LOAD_ADDR, push &r
48   24    0    1  PUSH_LIT 1
49   21    0    1  INDEX atab[1], pop(1)
50   24    0    4  PUSH_LIT 4
51   21    0    2  INDEX atab[2], pop(1)
52   24    0    2  PUSH_LIT 2
53   21    0    3  INDEX atab[3], pop(1)
54   24    0   65  PUSH_LIT 65
55   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)

   21    r[0][1][3] = 'B';

56    0    0   17  LOAD_ADDR, push &r
57   24    0    0  PUSH_LIT 0
58   21    0    1  INDEX atab[1], pop(1)
59   24    0    1  PUSH_LIT 1
60   21    0    2  INDEX atab[2], pop(1)
61   24    0    3  PUSH_LIT 3
62   21    0    3  INDEX atab[3], pop(1)
63   24    0   66  PUSH_LIT 66
64   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)

   22    cout << "r[1][4][2]  = '" << r[1][4][2]

65   28    0   52  WRITE_RAWSTRING stab[52] to stdout
66    0    0   17  LOAD_ADDR, push &r
67   24    0    1  PUSH_LIT 1
68   21    0    1  INDEX atab[1], pop(1)
69   24    0    4  PUSH_LIT 4
70   21    0    2  INDEX atab[2], pop(1)
71   24    0    2  PUSH_LIT 2
72   21    0    3  INDEX atab[3], pop(1)

   23         << "'   r[0][4][3] =  " << r[0][1][3] << "'\n";

73   34    0    0  VALUE_AT, s[t] = s[s[t]]
74   29    0    3  WRITE (char) s[t] to stdout, pop(1)
75   28    0   68  WRITE_RAWSTRING stab[68] to stdout
76    0    0   17  LOAD_ADDR, push &r
77   24    0    0  PUSH_LIT 0
78   21    0    1  INDEX atab[1], pop(1)
79   24    0    1  PUSH_LIT 1
80   21    0    2  INDEX atab[2], pop(1)
81   24    0    3  PUSH_LIT 3
82   21    0    3  INDEX atab[3], pop(1)
83   34    0    0  VALUE_AT, s[t] = s[s[t]]
84   29    0    3  WRITE (char) s[t] to stdout, pop(1)
85   28    0   87  WRITE_RAWSTRING stab[87] to stdout
```

```
    24 }

  86   31    0    0  HALT

++-outer-++:
  87    0    0    0  LOAD_ADDR, push &i
  88   24    0   17  PUSH_LIT 17
  89   38    0    0  STORE, s[s[t-1]] = s[t], pop(2)
  90   81    0    0  SHORTRET, pc = shortcall_reg
```