BACI System Separate Compilation Guide

Bill Bynum
College of William and Mary

Scott Mitchell
College of William and Mary

Tracy Camp
Colorado School of Mines

November 14, 2001

Contents

1 Introduction

This document describes use of the BACI System `bald` PCODE linker and `baar` PCODE archiver. These two tools, in conjunction with the two BACI compilers, implement separate compilation in the BACI system.

Programs of the BACI System

program	function	described in
`bacc`	BACI C−− to PCODE Compiler	cmimi.ps
`bapas`	BACI Pascal to PCODE Compiler	guidepas.ps
`bainterp`	command-line PCODE Interpreter	cmimi.ps, guidepas.ps
		disasm.ps
`bagui`	Graphical user interface to the	guiguide.ps
	PCODE Interpreter (UNIX systems only)	
`badis`	PCODE de-compiler	disasm.ps
`baar`	PCODE archiver	this guide (sepcomp.ps)
`bald`	PCODE linker	this guide (sepcomp.ps)

"Separate compilation" here refers to the process of compiling source programs into separate PCODE files with either of the BACI System compilers, `bacc` or `bapas`, creating libraries of PCODE files with the `baar` archiving program, and combining collections of separately compiled PCODE files into an executable PCODE file with the `bald` linker. Separate compilation was popularized by its use in the UNIX operating system in the 1970's. At the current time, almost all substantial C programs in almost every computing environment use separate compilation.

Separate compilation offers several significant advantages:

Modularity

The ability to combine object files produced from separate source files into a single application allows the programmer to decompose the source code for a large program into small, easily understood units. The programmer is never forced to treat the entirety of the source code for the application as one unit.

Multiple Authorship

Separate compilation allows different programmers to author separate parts of a large application. The separate pieces are compiled and linked together to create the application. Each programmer is freed from maintaining a detailed knowledge of the entire application, and can concentrate on the part of the code in the programmer's area of expertise.

Code Re-use

Software tasks that must be performed by several different applications can be isolated into subroutines that are compiled and then stored in code libraries. When the application is assembled by the linker, the common software can be retrieved by the linker from the appropriate code libraries into the executable being constructed. Compilation of the code for commonly occurring software tasks occurs only once.

Encapsulation, Information Hiding

Separate compilation allows the programmer to suppress details of the source code that its users should not know, such as the variables or data structures used. Since users of the code only have access to the object code, not the source code, users are prevented from making assumptions about the internal execution of the software, such as variable values and data structures used.

Although applications written in the BACI System are considerably less substantial than most of the separately compiled applications in a UNIX system, the advantages of separate compilation still apply to the PCODE of the BACI System.

2 Separate Compilation in the BACI System

The BACI System user can reap the benefits of separate compilation using the following steps:

1. Produce separate PCODE files with the BACI System compilers `bacc` or `bapas`, using the `-c` command-line option. This option keeps the compiler from complaining that the PCODE object code contains unresolved external references. The PCODE files so produced will have a `.pob` suffix, rather than the more familiar `.pco` suffix produced when the `-c` option isn't used.

2. Combine several `.pob` files into a single `.pco` file using the `bald` BACI System linker. The `bald` linker is an "incremental" linker, in that it will accept its own output subsequently as input. This means that the user can call `bald` repeatedly until all external references in the code have been resolved.

3. Create PCODE libraries using the BACI System archiver, `baar`. The `bald` linker can use these PCODE libraries to resolve external references. This saves the user from having to list on the `bald` command line all of the PCODE files that will be used in the link in step 2.

The exact steps that a user must go through will be illustrated through a sequence of examples in Section 4.

Separate compilation was added to the BACI System in a sequence of steps. The `-c` option and the `extern` and `EXTERNAL` keywords were added to the BACI compilers in the summer of 1997. In 2000, Scott Mitchell, a Masters student in computer science at the College of William and Mary, created the initial version of the `bald` linker that would link `.pob` files given on the command line. This was a significant achievement, especially in view of the fact that separate compilation was not a consideration when the BACI PCODE file was designed. Also in 2000, Bill Bynum created the `baar` archiver program from a similar program written in 1996 for the Motorola 6811 CPU. In spring and early summer of 2001, Scott Mitchell extended the `bald` linker to check `baar`-created libraries to resolve external references. In July and August of 2001, Tracy Camp and Bill Bynum integrated `bald` and `baar` into the BACI System, adding robustness and clarifying their user interfaces.

3 Command-line Options of `baar` and `bald`

3.1 `bald` Command-line Options

```
prompt% bald -h
BACI System:  PCODE Linker 12:57   3 Aug 2001

Usage: bald [options] List_of_files_to_link
   List_of_files_to_link is a list of .pob or .pco files to be linked.
   This list must contain at least one file and can contain up to 50 files.

Options (can occur in any order):
   -Ldirname
      Look in "dirname" for a BACI link library to use to resolve external
      references.  "dirname" should be a fully qualified directory name.
      The -L option can be repeated up to 30 times.

   -llibfile
      Look in the directories given by the -L options for the library file
      whose name is the concatenation of the 3 strings "lib", "libfile",
```

```
        and ".ba".  For example, "-lxu" specifies the library file "libxu.ba".
        BACI library files are created with the BACI archiver, baar.
        The -l option can be repeated up to 30 times.

   -o outfile_prefix
        Store the linked output in the file "outfile_prefix.pco" (if all externals
        externals were resolved) or "outfile_prefix.pob" (if external symbols
        remain).  Default output file is "aout.pco" or "aout.pob".

   -m
        Make a symbol map of the linked file.

   -v
        Do everything verbosely.

   -h
        Show this help information.

Example Usage:

   bald -o prog diskmon.pob diskusr.pob -L.. -L/home/baci/lib -ldisk -lfsem

Link the two files diskmon.pob disusr.pob, resolving external references with
the libraries libdisk.ba and libfsem.ba, located in the parent directory (..)
or in the directory /home/baci/lib.  The linked file will be named
prog.pco if all externals were resolved, or prog.pob, if not.
```

3.2 baar Command-line Options

```
prompt% baar h
BACI System: baar  PCODE archiver, 12:57   3 Aug 2001
Usage: baar [-]{drstwxh}[cosvV] archive-file file...
  commands (at least one must be present):
    d     delete named modules from the archive
    r     insert files into archive (with Replacement)
    s     install a symbol index in the archive
          (can be a modifier also -- see below)
    t     display list of files in the archive
    w     list all symbols in the archive
    x     extract named module from the archive
    h     show this help and exit
  optional modifiers:
    c     if archive didn't exist previously, create the new archive silently
    o     preserve the original dates of extracted files
    s     install a symbol index in the archive
    v     do everything verbosely
    V     show version number of program and exit
```

4 Examples of Separate Compilation in the BACI System

The examples presented in this section are written both in BACI C−− and in BACI Pascal. There are three examples in each language:

example	language	section	page
external variables and procedures	C−−	4.1	5
external array variables	C−−	4.2	11
external Hoare monitors	C−−	4.3	15
external variables and procedures	BACI Pascal	4.4	20
external array variables	BACI Pascal	4.5	24
external Hoare monitors	BACI Pascal	4.6	27

The examples in each language are self-contained. This enables a user interested only in one language to skip directly to the sections containing examples in that language.

4.1 External Variables and Procedures in C−−

The first example of separate compilation in C−− consists only of external variables and procedures. There are five source files involved:

extvars.cm

This source file simply declares three variables, an `int` variable, a `string[80]` variable, and a `char` variable.

```
//  example of external variables, procedures and functions
//  File:   extvars.cm
//              this file declares the variables

int theInt;
string[80] theString;
char theChar;
```

The `extvars.pob` file is created with the `bacc -c extvars` command.

extprocs.cm, extvars.h

This source file declares the procedures and functions that access the three variables declared in `extvars.cm`. The `.lst` file produced by the compiler is shown instead of the `.cm` file, because the `.lst` file shows the included header file `extvars.h` that declares the external variables. The `extprocs.lst` and `extprocs.pob` files are created with the `bacc -c extprocs` command.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: extprocs.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  //  example of external variables, procedures and functions
   2    0  //  File:   extprocs.cm
   3    0  //              this file declares the procs
   4    0
   5    0
   6    0  #include "extvars.h"       // bring in the references to the variables
>   1    0  //  example of external variables, procedures and functions
>   2    0  //  File:   extvars.h
>   3    0  //              this header file declares the external variables
>   4    0  //              for program units that use them
>   5    0
>   6    0  extern int theInt;
>   7    0  extern string[80] theString;
>   8    0  extern char theChar;
Returning to file extprocs.cm
   7    0
   8    0  void store_theInt(int q)
   9    0     // stores 'q' in theInt
  10    0  {
  11    0     theInt = q;
  12    3  }   // store_theInt
  13    4
  14    4  int  get_theInt()
  15    4     // returns current value of theInt
  16    4  {
  17    4     return theInt;
  18    8  }   // get_theInt
  19    9
  20    9  void store_theString(string u)
  21    9     // stores 'u' in theString
  22    9  {
  23    9     stringCopy(theString,u);
  24   12  }   // store_theString
  25   13
  26   13  void get_theString(string v)
  27   13     // returns current val of theString in the ref. variable v
```

```
28   13      // the 'string' type is passed by reference
29   13  {
30   13      stringCopy(v,theString);
31   16  }   // get_theString
32   17
33   17  void store_theChar(char a)
34   17      // store the char 'a' in theChar
35   17  {
36   17      theChar = a;
37   20  }   // store_theChar
38   21
39   21  char get_theChar()
40   21      // returns the current value of theChar
41   21  {
42   21      return theChar;
43   25  }   // get_theChar
```

Use of a header file to declare external references, as is done here with **extvars.h**, is highly desirable. If every program unit that needs to refer to the external variables uses the same header file, then referencing mistakes will be minimized.

Most of the source code in this file is straightforward and needs no comment. Usage of the procedure `stringCopy` in `get_theString` and `store_theString` is required because the `string` type is passed by reference (that is, the address of the `string` variable is passed to the subroutine, rather than the contents of the `string` variable).

extmain.cm, extprocs.h

This file contains the `main` proc for the example. The `extmain.lst` compiler listing file is shown below, rather than the `extmain.cm` source file, because the `extmain.lst` file also shows the inclusion of the `extprocs.h` header file. The `extmain.lst` and `extmain.pob` files are created with the `bacc -c extmain` command.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: extmain.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  //  example of external variables, procedures and functions
   2    0  //  File:   extmain.cm
   3    0  //            this file contains the main program for the example
   4    0
   5    0  #include "extprocs.h"   // declare the external procs
>   1    0  //  example of external variables, procedures and functions
>   2    0  //  File:   extprocs.h
>   3    0  //            this header file declares the external procs
>   4    0  //            for program units that use them
>   5    0
>   6    0  extern void store_theInt(int yy);        // stores 'yy' in theInt
>   7    0  extern int  get_theInt();                // returns current value of theInt
>   8    0
>   9    0  extern void store_theString(string ss); // stores 'ss' in theString
>  10    0  extern void get_theString(string tt);
>  11    0     // returns current val of theString in 'tt' (strings are pass-by-reference)
>  12    0
>  13    0  extern void store_theChar(char cc);        // stores 'cc' in theChar
>  14    0  extern char get_theChar();               // returns current value of theChar
Returning to file extmain.cm
   6    0                           // Note that the external variables are not
   7    0                           // referred to here
   8    0  main()
   9    1  {
  10    1      string[20] u;
  11    1      string[95] x;
  12    1      store_theInt(77);
  13    5      cout << "Current value of theInt is " << get_theInt() << endl;
  14   11      stringCopy(u,"Mareseatoats"); // string variables are passed by reference
  15   13      store_theString(u);           // the raw string can't appear in the call
  16   17      get_theString(x);             // string variables are passed by reference
```

```
17   21     cout << "Current value of theString is \"" << x << "\"" << endl;
18   26     store_theChar('Z');
19   30     cout << "Current value of theChar is '" << get_theChar() << "'" << endl;
20   37  }
```

Note that the parameter names used in the procedure and function declarations in the `extprocs.h` file need not agree with the actual parameter names in the `extprocs.cm`. For example, the `store_theInt` procedure is declared in `extprocs.cm` file as

```
void store_theInt(int q)
```

but in the `extprocs.cm` file, the `store_theInt` procedure is declared as

```
extern void store_theInt(int yy);
```

The `bald` linker checks only the types of procedure parameters, and not their names, so the parameter names used in a header file need not agree with the source code file that defines the procedure. In practice, most users prefer to make the parameters in the declaration of a procedure in a header file agree with the parameters in the declaration of the procedure in its source code file, because this is simply easier than making them different.

4.1.1 Using `bald` To Link The `.pob` Files

The three object files, `extmain.pob`, `extvars.pob`, and `extprocs.pob` are linked together with the command:

```
prompt% bald -v -m -o extexample extmain.pob extvars.pob extprocs.pob
++ Global symbols in the link file extmain.pob
  ('E' = external, 'D' = defined)
  name         object        type
get_theChar       function   char      E
get_theInt        function   int       E
get_theStrin      procedure  void      E
main              main proc  void      D
store_theCha      procedure  void      E
store_theInt      procedure  void      E
store_theStr      procedure  void      E
++ Global symbols in the link file extvars.pob
  ('E' = external, 'D' = defined)
  name         object        type
theChar           variable   char      D
theInt            variable   int       D
theString         variable   string    D
++ Global symbols in the link file extprocs.pob
  ('E' = external, 'D' = defined)
  name         object        type
get_theChar       function   char      D
get_theInt        function   int       D
get_theStrin      procedure  void      D
store_theCha      procedure  void      D
store_theInt      procedure  void      D
store_theStr      procedure  void      D
theChar           variable   char      E
theInt            variable   int       E
theString         variable   string    E
Output link file stored in extexample.pco
Map file stored in extexample.map
```

The `-v` (verbose) option on the command line generates most of the output coming from the command. If the `-v` option were not present, then only the last two lines would appear. The verbose output lists the symbols in each `.pob` file as the `bald` program encounters them. For example, the `extmain.pob` file contains external references to the symbols `get_theChar`, `get_theInt`, `get_theStrin` (actually, `get_theString`, but only 12 characters are significant to the compiler and linker), `store_theCha`, `store_theInt`, `store_theStr` and defines the `main` procedure.

The `-o` (lower case "oh") option provides the prefix of the name of the output file of the link. In this case, the `-o` option specifies that the linked file should be named `extexample.pco`, if all external references are resolved (as happened here), or `extexample.pob`, if one or more external references remain after the link.

The `-m` option produces a "symbol map" for the linked file that, in this case, is stored in the file `extexample.map`:

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the extexample.pco link file
Files included in the link
index     file
0      extmain.pob
1      extvars.pob
2      extprocs.pob
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object        type           references
get_theChar       function   char       0 E  2 D
get_theInt        function   int        0 E  2 D
get_theStrin      procedure  void       0 E  2 D
main              main proc  void       0 D
store_theCha      procedure  void       0 E  2 D
store_theInt      procedure  void       0 E  2 D
store_theStr      procedure  void       0 E  2 D
theChar           variable   char       1 D  2 E
theInt            variable   int        1 D  2 E
theString         variable   string     1 D  2 E
```

This map file describes every reference to every symbol that appears in the linked file. For example, the symbol map listing indicates that the three variables `theChar`, `theInt`, and `theString` were defined in the file with index `1`, `extvars.pob`, were referenced as externals in the file with index `2`, `extprocs.pob`, and were not referenced in the file with index `0`, `extmain.pob`. Of course, this information is not new to us, since we have seen the source files in this case. Because the source files may not always be available, the symbol map information can be useful in determining which external references have not been resolved.

The order in which the `.pob` files are listed on the command line does not matter. In this case, there are six different orders in which the `.pob` files could be specified on the command line. Certainly, these six `.pco` files produced from the different orderings will differ, but the execution of the six `.pco` files should be identical (if not, then another linker bug has been discovered).

4.1.2 Using the `baar` Archiver To Simplify the Link

The `baar` archiver program can create and manage a library consisting of one or more `.pob` files. The `bald` linker can be required to search a collection of libraries to resolve external references in a collection of `.pob` files being linked, so that most of the `.pob` files required for the link need not be specifically mentioned on the `bald` command line.

In our running example, let us suppose that we decide to use the `baar` program to create a link library that `bald` can use to satisfy the external references in the `extmain.pob` program. We create a library, named `libvapro.ba`, and place the `extvars.pob` and `extprocs.pob` files in it with the command:

```
prompt% baar rcv libvapro.ba extvars.pob extprocs.pob
int variable theInt
```

```
string variable theString
character variable theChar
r- extvars.pob
external int variable theInt
external string variable theString
external character variable theChar
void function store_theInt
int function get_theInt
void function store_theStr
void function get_theStrin
void function store_theCha
character function get_theChar
r- extprocs.pob
```

The `v` flag causes the verbose output shown. If it were not present, then only the lines that begin with the `r-` symbol would appear. These lines indicate that the named file has been "replaced" in (in this case, added to) the library file. The verbose output lists the symbols in the `.pob` file being added to the library.

As we shall soon see, the `bald` linker expects the name of a library to be of the form `lib*.ba`; that is, the string `lib`, followed by zero or more characters, and terminated by the string `.ba`.

The `w` flag of `baar` can be used to list the symbols in a library:

```
prompt% baar vw libvapro.ba
Archive index
Declared symbols
int variable theInt in extvars.pob
string variable theString in extvars.pob
character variable theChar in extvars.pob
void function store_theInt in extprocs.pob
int function get_theInt in extprocs.pob
void function store_theStr in extprocs.pob
void function get_theStrin in extprocs.pob
void function store_theCha in extprocs.pob
character function get_theChar in extprocs.pob
External symbols
external int variable theInt in extprocs.pob
external string variable theString in extprocs.pob
external character variable theChar in extprocs.pob
```

The `t` flag of `baar` can be used to list information about the `.pob` files comprising a library:

```
prompt% baar vt libvapro.ba
rw-r--r-- 34/20       283 Jul 25 07:55 2001 __.SYMDEF
rw-r--r-- 34/20       931 Nov 14 17:25 2001 extvars.pob
rw-r--r-- 34/20      2572 Nov 14 17:25 2001 extprocs.pob
```

The leftmost column gives the access permissions for the file. These permissions only have meaning in a UNIX operating system. They won't be meaningful in the MS-DOS version of the program. The `34/20` gives the userid and groupid of the user who created the file. These values are zero in the MS-DOS version of the program. The next number over is the size (in bytes) of the file. The next four fields give the modification time and date for the file. The rightmost column gives the name of the file.

The `__.SYMDEF` entry of the library is simply the "table of contents" for the library. This member of the library lists the symbols occurring in the files of the library. The `bald` linker uses this information to decide which files from the library should be incorporated into the file being linked.

The `baar` program has two other options that might occasionally be useful to you. The `x` option extracts the `.pob` file you name from the named library into the current directory. For example, `baar xv libvapro.ba extvars.pob` would extract the copy of `extvars.pob` in the `libvapro.ba` into the current directory. The `d` option can be used in a similar way to delete the file you name from a given library. The "usage" output (the output of the command `baar -h`) shown in Section 3.2 gives a little more information about command-line options of the `baar` program.

We can then use this library to simplify the `bald` link of the `extmain.pob` file:

```
prompt% bald -v -m -o extexample2 extmain.pob -lvarproc
++ Global symbols in the link file extmain.pob
  ('E' = external, 'D' = defined)
  name         object        type
get_theChar       function   char      E
get_theInt        function   int       E
get_theStrin      procedure  void      E
main              main proc  void      D
store_theCha      procedure  void      E
store_theInt      procedure  void      E
store_theStr      procedure  void      E
++ Global symbols in the link file extprocs.pob (./libvapro.ba)
  ('E' = external, 'D' = defined)
  name         object        type
get_theChar       function   char      D
get_theInt        function   int       D
get_theStrin      procedure  void      D
store_theCha      procedure  void      D
store_theInt      procedure  void      D
store_theStr      procedure  void      D
theChar           variable   char      E
theInt            variable   int       E
theString         variable   string    E
++ Global symbols in the link file extvars.pob (./libvapro.ba)
  ('E' = external, 'D' = defined)
  name         object        type
theChar           variable   char      D
theInt            variable   int       D
theString         variable   string    D
Output link file stored in extexample2.pco
Map file stored in extexample2.map
```

The `bald` linker goes through the following steps in response to the previous command. First, the linker notes that the `extmain.pob` file contains external references. The linker then forms a list of valid library file names by combining the directory names specified by the user through the `-L` option (see section 4.2.2) with the library file names specified by the user through the `-l` option, and then opens all library files on the list of valid filenames.

The `bald` linker always checks the current directory (`./`) for library files. The `libvapro.ba` library file that contains `extvars.pob` and `extprocs.pob` is in the current directory, so the `-L.` option is not required in this case. In this example, the only valid library name supplied by the user is `-lvapro`. The linker first expands the `vapro` string of the `-l` option into the library name `libvapro.ba`, and then prepends the library directory `./` to obtain the fully qualified library pathname `./libvapro.ba`.

The `bald` linker then enters the following loop:

1. Get the next unresolved external reference from the link file being constructed. In this case, the first such external reference will be the proc `get_theChar`, because the linker goes through the symbol table from last to first, but you don't really need to know this fact.

2. The linker consults the `__.SYMDEF` table of contents of each of the open libraries (in this case, the `./libvapro.ba` is the only open library) to see if any of them contains a `.pob` file that defines (or "resolves") the external reference. In this case, the `bald` linker will discover that the `extprocs.pob` file contains a `get_theChar` proc with `void` return type.

3. The `bald` linker reads that file into memory from the library and links it into the current working linkfile.

4. If, after the link, the working linkfile contains external references that haven't been checked against all open libraries, then the linker goes to step 1 and repeats the process; otherwise, the linker writes the current working linkfile to disk and terminates.

In this particular case, the above loop executes twice, because the `extprocs.pob` file contains external references, too; namely, `theInt`, `theString`, and `theChar`. On step 2 of the second time through the loop, the linker will discover that the `extvars.pob` file contains definitions of these symbols. After the linking that occurs in step 3, the linker will realize that all external references have been resolved and terminate.

The symbol map for the link, produced by the `-m` option, is:

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the extexample2.pco link file
Files included in the link
index     file
0      extmain.pob
1      extprocs.pob (./libvapro.ba)
2      extvars.pob (./libvapro.ba)
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object        type           references
get_theChar       function   char       0 E  1 D
get_theInt        function   int        0 E  1 D
get_theStrin      procedure  void       0 E  1 D
main              main proc  void       0 D
store_theCha      procedure  void       0 E  1 D
store_theInt      procedure  void       0 E  1 D
store_theStr      procedure  void       0 E  1 D
theChar           variable   char       1 E  2 D
theInt            variable   int        1 E  2 D
theString         variable   string     1 E  2 D
```

4.2 External Array Variables in C−−

These example source files illustrate how to use arrays with separate compilation. There are six source files involved:

arrdef.cm, tdarr23.h

This source file simply declares two dimensional array variable to be used. We show below the `arrdef.lst` file from the compilation, because it also shows the included file `tdarr23.h`.

Because the C−− compiler grew out of the BACI Pascal compiler, some vestiges of Pascal strong typing remain, such as the way that array variables must be declared as procedure or function parameters. The `tdarr23.h` header file, shown between lines 5 and 6 of the `arrdef.cm` file provide the necessary `typedef` to define the two-dimensional array type `Array23Parm`. Two constants, `dim1` and `dim2`, are used to define the dimensions of the array. The declaration on line 7 defines the array variable `A`.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: arrdef.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // C-- Array Example
   2    0  // File: arrdef.cm
   3    0  //          define an Array23Parm array
   4    0
   5    0  #include "tdarr23.h"    // import the typedef for Array23Parm
>   1    0  // C-- Array Example
>   2    0  // File: tdarr23.h
>   3    0  //          provides Array23Parm typedef
>   4    0
>   5    0  const int dim1 = 2;
```

```
>   6    0  const int dim2 = 3;
>   7    0
>   8    0  typedef int Array23Parm[dim1][dim2];
Returning to file arrdef.cm
   6    0
   7    0  Array23Parm A;
```

The `arrdef.pob` and `arrdef.lst` files are created with the `bacc -c arrdef` command.

arrprocs.cm, tdarr23.h

This source file declares the two procedures that access the array, `storeArray` and `showArray`. The `arrprocs.lst` file produced by the compiler is shown instead of the `arrprocs.cm` file, because this file shows (once again) the included header file `tdarr23.h` that provides the array `typedef`. The `arrprocs.lst` and `arrprocs.pob` files are created with the `bacc -c arrprocs` command.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: arrprocs.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // C-- Array Example
   2    0  // File: arrprocs.cm
   3    0  //          showArray and storeArray
   4    0
   5    0  #include "tdarr23.h"    // import the typedef for Array23Parm
>   1    0  // C-- Array Example
>   2    0  // File: tdarr23.h
>   3    0  //          provides Array23Parm typedef
>   4    0
>   5    0  const int dim1 = 2;
>   6    0  const int dim2 = 3;
>   7    0
>   8    0  typedef int Array23Parm[dim1][dim2];
Returning to file arrprocs.cm
   6    0
   7    0
   8    0  void showArray(Array23Parm u)
   9    0     // show u on stdout (but call it A)
  10    0  {
  11    0     int i, j;
  12    0     for (i = 0; i < dim1; i++) {
  13   14        for (j = 0; j < dim2; j++)
  14   28           cout << "A[" << i << "][" << j << "] = " << u[i][j] << "  ";
  15   44        cout << endl;
  16   45     }
  17   46  }  // showArray
  18   47
  19   47
  20   47  void storeArray(Array23Parm& a, int a00, int a01, int a02,
  21   47     int a10, int a11, int a12)
  22   47     // store the 6 values in the Array23Parm array a
  23   47  {
  24   47     a[0][0] = a00;
  25   54     a[0][1] = a01;
  26   61     a[0][2] = a02;
  27   68     a[1][0] = a10;
  28   75     a[1][1] = a11;
  29   82     a[1][2] = a12;
  30   89  }  // storeArray
```

The `Array23Parm` parameter `u` of the `showArray` procedure on lines 8 through 17 is passed by value (that is, a copy of the array is placed on the stack at the time of the call). The `Array23Parm` parameter `a` of the `storeArray` procedure on lines 20 through 30 is a reference parameter (that is, the address of the array is passed to the subroutine, so that the actual values stored in the array will be changed). The six values to be stored in the array, `a00` through `a12`, are passed by value.

arrmain.cm, arrprocs.h, tdarr23.h

This file contains the `main` proc for the example. The `arrmain.lst` compiler listing file is shown below because it provides the two header files used, too. Note that the `arrprocs.h` header file includes the `tdarr23.h` header file. The `Array23Parm` array `A` is declared as external in line 7 of `arrmain.lst`.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: arrmain.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // C-- Array Example
   2    0  // File: arrmain.cm
   3    0  //          the main program
   4    0
   5    0  #include "arrprocs.h"    // import the declarations of the array procs
>   1    0  // C-- Array Example
>   2    0  // File: arrprocs.h
>   3    0  //          header file for storeArray usage
>   4    0
>   5    0  #include "tdarr23.h"    // import the typedef for Array23Parm
>>   1    0  // C-- Array Example
>>   2    0  // File: tdarr23.h
>>   3    0  //          provides Array23Parm typedef
>>   4    0
>>   5    0  const int dim1 = 2;
>>   6    0  const int dim2 = 3;
>>   7    0
>>   8    0  typedef int Array23Parm[dim1][dim2];
Returning to file arrprocs.h
>   6    0
>   7    0  extern void storeArray(Array23Parm& a, int a00, int a01, int a02,
>   8    0     int a10, int a11, int a12);
>   9    0     // store the 6 values in the Array23Parm array a
>  10    0
>  11    0  extern void showArray(Array23Parm u);
>  12    0     // show u on stdout (but call it A)
Returning to file arrmain.cm
   6    0
   7    0  extern Array23Parm A;
   8    0
   9    0  main()
  10    1  {
  11    1     cout << "Storing values ... \n";
  12    2     storeArray(A,11,22,33,99,88,77);
  13   12     cout << "Showing what was stored ... \n";
  14   13     showArray(A);
  15   18  }  // main
```

4.2.1 Using `bald` To Link The `.pob` Files

The three object files, `arrdef.pob`, `arrprocs.pob`, and `arrmain.pob` are linked together with the command:

```
prompt% bald -v -m -o arrexample arrdef.pob arrmain.pob arrprocs.pob
Output link file stored in arrexample.pco
Map file stored in arrexample.map
```

The symbol map produced by the `-m` option is much like the previous symbol map that you have seen:

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the arrexample.pco link file
Files included in the link
index     file
0       arrmain.pob
1       arrdef.pob
2       arrprocs.pob
```

```
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object       type         references
A                 variable   array      0 E  1 D
Array23Parm       type       array      0 D  1 D  2 D
dim1              constant   int        0 D  1 D  2 D
dim2              constant   int        0 D  1 D  2 D
main              main proc  void       0 D
showArray         procedure  void       0 E  2 D
storeArray        procedure  void       0 E  2 D
```

When the linked program is executed, the results are as you would expect:

```
prompt% bainterp arrexample
Storing values ...
Showing what was stored ...
A[0][0] = 11  A[0][1] = 22  A[0][2] = 33
A[1][0] = 99  A[1][1] = 88  A[1][2] = 77
```

4.2.2 Using the `baar` Archiver To Simplify the Link

For this example, we decide to use the `baar` program to create two different link libraries. One library `libarr.ba` will be in the current directory and will contain the `arrdef.pob` file. The other library `libaproc.ba` will be in the `lib` subdirectory of the current directory and will contain the `arrprocs.pob` file.

These two libraries were created using `baar` in the manner you have seen previously:

```
prompt% baar rcv libarr.ba arrdef.pob
constant dim1
constant dim2
array variable A
r- arrdef.pob

prompt% baar rcv libarr.ba arrprocs.pob
Adding __.SYMDEF member (the archive symbol table)
constant dim1
constant dim2
void function showArray
void function storeArray
r- arrprocs.pob
```

Finally, we link the `arrmain.pob` file using the following command:

```
prompt% bald -v -m -o arrexample2 arrmain.pob -larr -laproc -L./lib
Output link file stored in arrexample2.pco
Map file stored in arrexample2.map
```

The symbol map produced by the `-m` flag is as follows:

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the arrexample2.pco link file
Files included in the link
index     file
0      arrmain.pob
1      arrdef.pob (./libarr.ba)
2      arrprocs.pob (./lib/libarr.ba)
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object       type         references
A                 variable   array      0 E  1 D
Array23Parm       type       array      0 D  1 D  2 D
dim1              constant   int        0 D  1 D  2 D
dim2              constant   int        0 D  1 D  2 D
main              main proc  void       0 D
showArray         procedure  void       0 E  2 D
storeArray        procedure  void       0 E  2 D
```

During the link, the linker pairs each library name given with an `-l` option with each library directory given with the `-L` option (plus the current directory `./`) to obtain a candidates for the the fully qualified filenames of valid library files. The `bald` linker tries to open each of the candidate library files. The candidate files that open successfully are the valid library files. In this case, there are two valid library files, `./libarr.ba` and `./lib/libaproc.ba`.

Note that the `bald` linker has retrieved the `arrdef.pob` file from the `./libarr.ba` library and the `arrprocs.pob` file from the `./lib/libaproc.ba` library to complete the link. The execution of the linked file is identical to what you have seen before.

4.3 External Hoare Monitors in C−−

The example that we use to illustrate external Hoare monitors is the "classical" producer/consumer problem, in which producer processes store characters into a bounded buffer and consumer processes consume characters from the bounded buffer. Mutually exclusive access to the bounded buffer is managed by a Hoare monitor.

This application is implemented in seven files:

bbuff.cm

This file contains the bounded buffer management code, implemented in a Hoare monitor, `bounded_buffer`. The ten-character buffer uses two `int` variables, `nextIn` and `nextOut`, that give the locations in the buffer of the next character to be stored and the next character to be retreived. The monitor uses two conditions, `notFull` and `notEmpty`, to regulate the access of calling processes to the bounded buffer. There are two externally visible monitor procedures, `append` and `retrieve`, that the producer and consumer processes can call.

Let us briefly go though the code for the `append` procedure. The monitor variable `bufferCount` holds the number of characters currently in the buffer. On entry to the `append` procedure, if the buffer is full (`bufferCount == bufferSize`), then the calling process is put to sleep on the `notFull` condition. If there is space in the buffer (or when space becomes available in the buffer through the action of `retrieve` and a sleeping `append` caller is awakened by a signal to the `notFull` condition), the input parameter `c` is stored into the buffer. The `nextIn` index is moved to the next index in the buffer, and the `bufferCount` variable is incremented.

```
// C-- Monitor Example
// File: bbuff.cm
//          Bounded buffer monitor

monitor bounded_buffer {
   const int bufferSize = 10;
   char buffer[bufferSize];

   int nextIn;       // index of next character coming into buffer
   int nextOut;      // index of next character going out of buffer
   int bufferCount;  // number of characters in the buffer
   condition   notEmpty;   // signalled when buffer is not empty
   condition   notFull;    // signalled when buffer is not full

   void append(char c)
      // append the character 'c' to the buffer
   {
      if (bufferCount == bufferSize) waitc(notFull);
      buffer[nextIn] = c;
      nextIn = (nextIn + 1) % bufferSize;
      bufferCount++;
      signalc(notEmpty);
   }  // append

   void retrieve(char& c)
      // retrieve a character from the buffer into 'c'
```

```
   {
      if (bufferCount == 0) waitc(notEmpty);
      c = buffer[nextOut];
      nextOut = (nextOut + 1) % bufferSize;
      bufferCount--;
      signalc(notFull);
   }  // retrieve

   init { nextIn = nextOut = bufferCount = 0; }
}  // boundedBuffer monitor
```

Finally, the `notEmpty` condition is signaled, in case processes calling `retrieve` are suspended on the `nonEmpty` condition. This signal is postponed to the last statement of `append` because of the "Immediate Resumption Requirement" used by the BACI PCODE interpreter. This requirement means that a process signaling a condition in a monitor is suspended so that any process sleeping on the signaled condition can be resumed immediately (recall that only one thread of execution at a time can be active inside a Hoare monitor). Use of Immediate Resumption is based on the assumption that the condition that is signaled needs to be taken care of, so any process sleeping on the condition should have a higher priority of execution in the monitor than the signaling process.

The source code for `retrieve` is almost identical to the code of `append`, with only minor modifications. A reference variable is used to transfer the character removed from the buffer to the caller, rather than changing `retrieve` to a function of type `char` and returning the character as the function's return value. With the reference variable, the transfer occurs immediately with the reference variable, while the return statement, if it were added, would be delayed by the Immediate Resumption Requirement.

prod.cm, bbuff.h

The `prod.lst` file, produced by the compiler, is shown instead of the `prod.cm` file, because the compilation listing shows the `bbuff.h` include file. Note that the `bbuff.h` include file declares only the externally visible part of the `bounded_buffer` monitor, the `append` and `retrieve` procedures.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: prod.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // C-- Monitor Example
   2    0  // File: prod.cm
   3    0  //          the character producer
   4    0
   5    0  #include "bbuff.h"      // bring in the monitor
>   1    0  // C-- Monitor Example
>   2    0  // File: bbuff.h
>   3    0  //          bounded buffer monitor
>   4    0
>   5    0  extern monitor bounded_buffer {
>   6    0     void append(char c);      // append the character 'c' to the buffer
>   7    0     void retrieve(char& c);   // retrieve a character from the buffer into 'c'
>   8    0  }  // bounded_buffer monitor
Returning to file prod.cm
   6    0
   7    0  extern binarysem mutex; // unscramble screen output
   8    0
   9    0  void producer(char c)
  10    0     // appends 'c' to the buffer forever
  11    0  {
  12    0     while(1) {
  13    2        append(c);
  14    6        p(mutex);
  15    8        cout << "producer " << c << " appended " << c << endl;
  16   15        v(mutex);
```

```
 17   17     }
 18   18  }  // producer
```

The **producer** procedure simply calls **append** to adds its **char** parameter **c** to the buffer forever. Because the main program (**prodcons.cm**, see below) starts multiple **producer** and **consumer** processes, screen output by either process has to be serialized through the use of a binary semaphore, **mutex**, declared and initialized in the main program.

cons.cm,bbuff.h

The code for the **consumer** procedure is just like the code for the **producer** procedure, except that **retrieve** is called, instead of **append**.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: cons.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // C-- Monitor Example
   2    0  // File: cons.cm
   3    0  //          the character consumer
   4    0
   5    0  #include "bbuff.h"   // bounded buffer monitor declarations
>   1    0  // C-- Monitor Example
>   2    0  // File: bbuff.h
>   3    0  //          bounded buffer monitor
>   4    0
>   5    0  extern monitor bounded_buffer {
>   6    0     void append(char c);      // append the character 'c' to the buffer
>   7    0     void retrieve(char& c);   // retrieve a character from the buffer into 'c'
>   8    0  }  // bounded_buffer monitor
Returning to file cons.cm
   6    0
   7    0  extern binarysem mutex; // unscramble screen output
   8    0
   9    0  void consumer(char c)
  10    0     // consumes characters from the buffer  (has ID 'c')
  11    0  {
  12    0     char d;
  13    0     while (1) {
  14    2        retrieve(d);
  15    6        p(mutex);
  16    8        cout << "consumer " << c << " retrieved " << d << endl;
  17   15        v(mutex);
  18   17     }
  19   18  }  // consumer
```

prodcons.cm,prod.h,cons.h

This file contains the source code for the main program. The two include files declare the prototypes of the **producer** and **consumer** procedures.

```
BACI System: C-- to PCODE Compiler, 12:57   3 Aug 2001
Source file: prodcons.cm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // C-- Monitor Example
   2    0  // File: prodcons.cm
   3    0  //          the producer-consumer main program
   4    0
   5    0  #include "prod.h"
>   1    0  // C-- Monitor Example
>   2    0  // File: prod.h
>   3    0  //          the character producer
>   4    0  extern void producer(char c);
>   5    0     // appends 'c' to the buffer forever
Returning to file prodcons.cm
   6    0  #include "cons.h"
>   1    0  // C-- Monitor Example
>   2    0  // File: cons.h
>   3    0  //          the character consumer
```

```
>   4    0  extern void consumer(char c);
>   5    0     // consumes characters from the buffer  (has ID 'c')
Returning to file prodcons.cm
   7    0
   8    0  binarysem mutex;  // to unscramble screen output
   9    0
  10    0  main()
  11    1  {
  12    1     initialsem(mutex,1);
  13    4     cobegin{
  14    5        producer('A'); producer('B'); producer('C');
  15   17        producer('D'); producer('E'); producer('F');
  16   29        consumer('1'); consumer('2'); consumer('3');
  17   41     }
  18   42  }
```

The main program declares and initializes the `mutex` binary semaphore that the `producer` and `consumer` procedures use to keep their output to the terminal screen from intermixing.

In the `cobegin` block, the program starts six `producer` processes and three `consumer` processes. One could add debugging output to the `bounded_buffer` monitor to discover that in this case, the buffer will stay full most of the time, with `producers` waiting to add their characters. If there were more `consumer` processes than `producer` processes, then the buffer would be empty most of the time, with `consumer` processes waiting to remove characters.

4.3.1 Using `bald` To Link The `.pob` Files

The four object files, `bbuff.pob`, `prod.pob`, `cons.pob`, and `prodcons.pob` are linked together with the command:

```
prompt% bald -m -o proconex bbuff.pob prod.pob cons.pob prodcons.pob
Output link file stored in proconex.pco
Map file stored in proconex.map
```

Execution of the linked program yields:

```
prompt% bainterp proconex |less
Linked files: bbuff.pob prod.pob cons.pob prodcons.pob
Executing PCODE ...
producer A appended A
producer D appended D
producer C appended C
producer A appended A
producer B appended B
consumer 3 retrieved A
producer E appended E
producer B appended B
producer C appended C
producer D appended D
producer A appended A
producer C appended C
producer F appended F
consumer 3 retrieved D
producer A appended A
consumer 1 retrieved C
producer F appended F
producer E appended E
producer C appended C
consumer 3 retrieved B
consumer 1 retrieved E
consumer 2 retrieved A
consumer 3 retrieved B
producer C appended C
producer F appended F
producer E appended E
consumer 2 retrieved D
```

```
producer B appended B
consumer 3 retrieved A
consumer 1 retrieved C
producer A appended A
consumer 2 retrieved C
producer C appended C
producer A appended A
producer D appended D
consumer 2 retrieved F
consumer 3 retrieved F
consumer 1 retrieved A
producer E appended E
consumer 2 retrieved E
producer B appended B
producer F appended F
producer B appended B
consumer 3 retrieved C
consumer 1 retrieved F
consumer 2 retrieved C
...
```

You may find the above output confusing, because characters seem to leave the buffer in a different order than they were entered. If you start from the top of the output, the first three characters appended to the buffer are `A`, `D`, and `C`. The first three characters retrieved from the buffer match these characters exactly. However, the fourth, fifth, and sixth characters added to the buffer are `A`, `B`, and `E`, in that order, yet the the fourth, fifth, and sixth characters retrieved from the buffer are `B`, `E`, and `A`.

You can add debugging output to the `bounded_buffer` monitor (as we did) to discover that the program output can occur in a different order from the order that the characters enter and leave the buffer. In this particular case, the order that the fourth, fifth, and sixth characters were added to the buffer is exactly the same as the order in which they were retrieved. However, the `append` call from the `A` producer reached the `mutex` critical section before the `append` calls from the `B` and `E` producers did, so the `A` producer's output appeared first. Consequently, the `B` producer process was awakened from the `mutex` queue before the `E` process was, which matches the order that the corresponding characters were appended to the queue. This preservation of order doesn't always happen, because the BACI semaphore queues are not necessarily FIFO.

Recalling the source code above, the `producer` and `consumer` procedures both have non-terminating loops, so this program must be terminated by some external influence. In this case, the program output was piped (in LINUX) into the `less` pager, so that the `bainterp` program could be terminated by terminating the pager.

4.3.2 Using `baar` To Simplify The Link

In this example, we combine the `bbuff.pob`, `prod.pob`, and `cons.pob`, into a library file `libprcon.ba` with the command:

```
prompt% baar rcv libprcon.ba bbuff.pob prod.pob cons.pob
monitor bounded_buff
monitor void function append
monitor void function retrieve
r- bbuff.pob
external monitor bounded_buff
external monitor void function append
external monitor void function retrieve
external binary semaphore variable mutex
void function producer
r- prod.pob
external monitor bounded_buff
external monitor void function append
external monitor void function retrieve
external binary semaphore variable mutex
```

```
void function consumer
r- cons.pob
```

The execution of the `proconx2.pco` linker output file is much like what you have seen above.

4.4 External Variables and Procedures in BACI Pascal

This is the BACI Pascal example of external variables and procedures that corresponds to the C−− example in Section 4.1. A Pascal programmer may find find this example mildly surprising, because the BACI Pascal compiler differs in the syntax of external references from the "standard" Pascal described by Niklaus Wirth.

The five BACI Pascal source files are:

pxtvars.pm

This source file simply declares three variables, an `INTEGER` variable, a `STRING[80]` variable, and a `CHAR` variable. The capitalization of the keywords is merely a personal preference, and is not required by the BACI Pascal compiler.

```
//  BACI Pascal example of external variables, procedures and functions
//  File:   pxtvars.pm
//              this file declares the variables

VAR
   theInt    : INTEGER;
   theString : STRING[80];
   theChar   : CHAR;
```

The `pxtvars.pob` file is created with the `bapas -c pxtvars` command.

pxtprocs.pm, pxtvars.h

This source file declares the procedures and functions that access the three variables declared in `pxtvars.pm`. The `.lst` file produced by the compiler is shown instead of the `.pm` file, because the `.lst` file shows the included header file `pxtvars.h` that declares the external variables). The `pxtprocs.lst` and `pxtprocs.pob` files are created with the `bapas -c pxtprocs` command.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: pxtprocs.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  //  BACI Pascal example of external variables, procedures and functions
   2    0  //  File:   pxtprocs.pm
   3    0  //             this file declares the procs
   4    0
   5    0  #INCLUDE "pxtvars.h"       // bring in the references to the variables
>   1    0  {   BACI Pascal example of external variables, procedures and functions
>   2    0      File:   pxtvars.h
>   3    0                 this header file declares the external variables
>   4    0                 for program units that use them
>   5    0  }
>   6    0
>   7    0  EXTERNAL VAR
>   8    0     theInt    : INTEGER;
>   9    0     theString : STRING[80];
>  10    0     theChar   : CHAR;
Returning to file pxtprocs.pm
   6    0
   7    0  PROCEDURE store_theInt(q : INTEGER);
   8    0     //  stores 'q' in theInt
   9    0  BEGIN
  10    0     theInt := q;
  11    3  END; // store_theInt
  12    4
```

```
13    4  FUNCTION get_theInt : INTEGER;
14    4     // returns current value of theInt
15    4  BEGIN
16    4     get_theInt := theInt;
17    7  END;  // get_theInt
18    8
19    8  PROCEDURE store_theString(u : STRING);
20    8     // stores 'u' in theString
21    8  BEGIN
22    8     stringCopy(theString,u);
23   11  END;  // store_theString
24   12
25   12  PROCEDURE get_theString(v : STRING);
26   12     // returns current val of theString in the ref. variable v
27   12     // the 'string' type is passed by reference
28   12  BEGIN
29   12     stringCopy(v,theString);
30   15  END; // get_theString
31   16
32   16  PROCEDURE store_theChar(a : CHAR);
33   16     // store the char 'a' in theChar
34   16  BEGIN
35   16     theChar := a;
36   19  END; // store_theChar
37   20
38   20  FUNCTION get_theChar: CHAR;
39   20     // returns the current value of theChar
40   20  BEGIN
41   20     get_theChar := theChar;
42   23  END;  // get_theChar
```

External variables are noted in BACI Pascal with the combination of the `EXTERNAL VAR` keywords. This differs from the Wirth Pascal syntax for external variables. The chief virtue of the `EXTERNAL VAR` usage is that it is easy to parse.

Most of the source code in this file is straightforward and needs no comment. As in the C−− example, usage of the procedure `stringCopy` in `get_theString` and `store_theString` is required because the `string` type is passed by reference (that is, the address of the `string` variable is passed to the subroutine, rather than the actual `string` variable itself).

pxtmain.pm, pxtprocs.h

This file contains the main procedure for the example.

In BACI Pascal, all external declarations, both local and external, must occur at the "global" level, outside of the Pascal block delimited by the `PROGRAM` and `END.` tokens. This is why the `pxtprocs.h` header file is included above the `PROGRAM` statement.

In addition, the `EXTERNAL` keyword must precede the declaration of the function or procedure here, rather than trailing it, as in in Wirth's Pascal.

Because of the Pascal `PROGRAM` statement, the main procedure is required to be named. In this case the main procedure is named `pxtmain`. The `pxtmain.lst` compiler listing file is shown below, rather than the `pxtmain.pm` source file, because the `pxtmain.lst` file also shows the inclusion of the `pxtprocs.h` header file. The `pxtmain.lst` and `pxtmain.pob` files are created with the `bapas -c pxtmain` command.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: pxtmain.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  //  BACI Pascal example of external variables, procedures and functions
   2    0  //  File:   pxtmain.pm
   3    0  //             this file contains the main program for the example
   4    0
   5    0  #INCLUDE "pxtprocs.h"   // declare the external procs
>   1    0  {  BACI Pascal example of external variables, procedures and functions
```

```
>   2    0     File:   pxtprocs.h
>   3    0               this header file declares the external procs
>   4    0               for program units that use them
>   5    0  }
>   6    0
>   7    0  EXTERNAL PROCEDURE store_theInt(yy : INTEGER);
>   8    0     { stores 'yy' in theInt }
>   9    0  EXTERNAL FUNCTION get_theInt : INTEGER;
>  10    0     { returns current value of theInt }
>  11    0
>  12    0  EXTERNAL PROCEDURE store_theString(ss : STRING);
>  13    0     { stores 'ss' in theString }
>  14    0  EXTERNAL PROCEDURE get_theString(tt : STRING);
>  15    0     { returns current val of theString in 'tt' }
>  16    0     { (strings are pass-by-reference)         }
>  17    0
>  18    0  EXTERNAL PROCEDURE store_theChar(cc : CHAR);
>  19    0     { stores 'cc' in theChar }
>  20    0  EXTERNAL FUNCTION get_theChar : CHAR;
>  21    0     { returns current value of theChar }
Returning to file pxtmain.pm
   6    0                         // Note that the external variables are not
   7    0                         // referred to here
   8    0  PROGRAM pxtmain;
   9    0
  10    0  VAR
  11    0     u  : STRING[20];
  12    0     x  : STRING[95];
  13    0
  14    0  BEGIN
  15    1     store_theInt(77);
  16    5     WRITELN("Current value of theInt is ",get_theInt);
  17   11     stringCopy(u,"Mareseatoats"); // string variables are passed by reference
  18   13     store_theString(u);           // the raw string can't appear in the call
  19   17     get_theString(x);             // string variables are passed by reference
  20   21     WRITELN("Current value of theString is \"",x, "\"");
  21   26     store_theChar('Z');
  22   30     WRITELN("Current value of theChar is '", get_theChar, "'");
  23   37  END.
```

Note that the parameter names used in the procedure and function declarations in the `pxtprocs.h` file need not agree with the actual parameter names in the `pxtprocs.pm`. For example, the `store_theInt` procedure is declared in `pxtprocs.pm` file as

```
PROCEDURE store_theInt( q : INTEGER );
```

but in the `extprocs.pm` file, the `store_theInt` procedure is declared as

```
EXTERNAL PROCEDURE store_theInt( yy : INTEGER );
```

The `bald` linker checks only the types of procedure parameters, and not their names, so the parameter names used in a header file need not agree with the source code file that defines the procedure. In practice, most users prefer to make the parameters in the declaration of a procedure in a header file agree with the parameters in the declaration of the procedure in its source code file, because this is simply easier than making them different.

4.4.1 Using `bald` To Link The `.pob` Files

The three object files, `pxtmain.pob`, `pxtvars.pob`, and `pxtprocs.pob` are linked together with the command:

```
prompt% bald -v -m -o pxtexample pxtmain.pob pxtvars.pob pxtprocs.pob
++ Global symbols in the link file pxtmain.pob
  ('E' = external, 'D' = defined)
  name         object        type
get_thechar       function   char      E
get_theint        function   int       E
get_thestrin      procedure  void      E
pxtmain           main proc  void      D
store_thecha      procedure  void      E
store_theint      procedure  void      E
store_thestr      procedure  void      E
++ Global symbols in the link file pxtvars.pob
  ('E' = external, 'D' = defined)
  name         object        type
thechar           variable   char      D
theint            variable   int       D
thestring         variable   string    D
++ Global symbols in the link file pxtprocs.pob
  ('E' = external, 'D' = defined)
  name         object        type
get_thechar       function   char      D
get_theint        function   int       D
get_thestrin      procedure  void      D
store_thecha      procedure  void      D
store_theint      procedure  void      D
store_thestr      procedure  void      D
thechar           variable   char      E
theint            variable   int       E
thestring         variable   string    E
Output link file stored in pxtexample.pco
Map file stored in pxtexample.map
```

The `-v` (verbose) option on the command line generates most of the output coming from the command. If the `-v` option were not present, then only the last two lines would appear. The verbose output lists the symbols in each `.pob` file as the `bald` program encounters them. For example, the `extmain.pob` file contains external references to the symbols `get_thechar`, `get_theint`, `get_thestrin` (actually, `get_thestring`, but only 12 characters are significant to the compiler and linker), `store_thecha`, `store_theint`, `store_thestr` and defines the `pxtmain` main procedure.

Because Pascal is not case-sensitive, the BACI Pascal compiler lower-cases all program identifiers during compilation. This simplifies considerably the task of finding things in the symbol table during compilation. For this reason, the symbols that you see during the linking process, because they are lower-cased, may differ from the appearance of the symbols in the source file.

The `-o` (lower case "oh") option provides the prefix of the name of the output file of the link. In this case, the `-o` option specifies that the linked file should be named `pxtexample.pco`, if all external references are resolved (as happened here), or `pxtexample.pob`, if one or more external references remain.

The `-m` option produces a "symbol map" for the linked file that, in this case, is stored in the file `pxtexample.map`:

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the pxtexample.pco link file
Files included in the link
index     file
0      pxtmain.pob
1      pxtvars.pob
2      pxtprocs.pob
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object        type           references
get_thechar       function   char       0 E  2 D
get_theint        function   int        0 E  2 D
get_thestrin      procedure  void       0 E  2 D
pxtmain           main proc  void       0 D
store_thecha      procedure  void       0 E  2 D
```

```
store_theint       procedure  void       0 E  2 D
store_thestr       procedure  void       0 E  2 D
thechar            variable   char       1 D  2 E
theint             variable   int        1 D  2 E
thestring          variable   string     1 D  2 E
```

The map file produced in this step is almost the same as the map file produced in Section 4.1.1, except that all symbol names have been lower-cased.

4.4.2 Using `baar` To Simplify The Link

The only difference between linking `.pob` files produced by `bapas` and linking `.pob` files produced by `bacc` stems from the fact that Pascal is not case-sensitive and C is. The interaction with `baar` and `bald` is exactly as described in Section 4.1.2.

4.5 External Array Variables in BACI Pascal

This is the BACI Pascal example of external an external array that corresponds to the C−− example in Section 4.2.

The six BACI Pascal source files required by the example are:

pardef.pm, ptarr23.h

This source file simply declares two dimensional array variable to be used. We show below the `pardef.lst` file from the compilation, because it also shows the included file `ptarr23.h`.

Because Pascal is strongly typed, arrays that will be used as subroutine parameters require a type declaration. The `ptarr23.h` header file, shown between lines 6 and 7 of the `pardef.pm` file provide the necessary definition of the two-dimensional array type `Array23Parm`. Two constants, `dim1` and `dim2`, are used to define the dimensions of the array. To mimic the behavior of the C−− program exactly, two additional constants, `dim1m1` (= `dim1` − 1) and `dim2m1` (= `dim2` − 1), are needed.

The declaration on line 9 defines the array variable `A`.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: pardef.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // BACI Pascal Array Example
   2    0  // File: pardef.pm
   3    0  //          define an Array23Parm array
   4    0
   5    0  #INCLUDE "ptarr23.h"    // import the typedef for Array23Parm
>   1    0  // BACI Pascal Array Example
>   2    0  // File: ptarr23.h
>   3    0  //          provides Array23Parm type definition
>   4    0
>   5    0  CONST
>   6    0     dim1   = 2;
>   7    0     dim1m1 = 1;
>   8    0     dim2   = 3;
>   9    0     dim2m1 = 2;
>  10    0
>  11    0  TYPE
>  12    0     Array23Parm = ARRAY [0..dim1m1 , 0..dim2m1] OF INTEGER;
Returning to file pardef.pm
   6    0
   7    0  VAR
   8    0     A : Array23Parm;
```

The `pardef.pob` and `pardef.lst` files are created with the `bapas -c pardef` command.

parprocs.cm, ptarr23.h

This source file declares the two procedures that access the array, `storeArray` and `showArray`. The `parprocs.lst` file produced by the compiler is shown instead of the `parprocs.pm` file, because the `parprocs.lst` file shows the included header file `ptarr23.h` that provides the necessary type definition. The `parprocs.lst` and `arrprocs.pob` files are created with the `bapas -c parprocs` command.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: parprocs.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1     0  // BACI Pascal Array Example
   2     0  // File: parprocs.pm
   3     0  //         showArray and storeArray
   4     0
   5     0  #INCLUDE "ptarr23.h"    // import the typedef for Array23Parm
>   1     0  // BACI Pascal Array Example
>   2     0  // File: ptarr23.h
>   3     0  //         provides Array23Parm type definition
>   4     0
>   5     0  CONST
>   6     0     dim1   = 2;
>   7     0     dim1m1 = 1;
>   8     0     dim2   = 3;
>   9     0     dim2m1 = 2;
>  10     0
>  11     0  TYPE
>  12     0     Array23Parm = ARRAY [0..dim1m1 , 0..dim2m1] OF INTEGER;
Returning to file parprocs.pm
   6     0
   7     0  PROCEDURE showArray( u : Array23Parm);
   8     0     //  show u on stdout (but call it A)
   9     0  VAR
  10     0     i, j : INTEGER;
  11     0  BEGIN
  12     0     FOR i := 0 TO dim1m1 DO
  13     4     BEGIN
  14     4        FOR j := 0 TO dim2m1 DO
  15     8           WRITE("A[",i,",",j,"] = ",u[i,j],"  ");
  16    24        WRITELN;
  17    25     END;
  18    26  END;  // showArray
  19    27
  20    27  PROCEDURE storeArray(VAR a : Array23Parm;
  21    27      a00, a01, a02, a10, a11, a12 : INTEGER);
  22    27     // store the 6 values in the Array23Parm array a
  23    27  BEGIN
  24    27     a[0,0] := a00;
  25    34     a[0,1] := a01;
  26    41     a[0,2] := a02;
  27    48     a[1,0] := a10;
  28    55     a[1,1] := a11;
  29    62     a[1,2] := a12;
  30    69  END;  // storeArray
```

The `Array23Parm` parameter `u` of the `showArray` procedure on lines 9 through 20 is passed by value (that is, a copy of the array is placed on the stack at the time of the call). The `Array23Parm` parameter `a` of the `storeArray` procedure on lines 23 through 33 is a reference parameter (that is, the address of the array is passed to the subroutine, so that the actual values stored in the array will be changed). The six values to be stored in the array, `a00` through `a12`, are passed by value. The `dim1m1` and `dim2m1` constants are useful in the `FOR` loops of lines 14 and 16 of `showArray`.

parmain.pm, parprocs.h, ptarr23.h

This file contains the main program for the example. The `parmain.lst` compiler listing file is shown below, because it provides the two header files used, too. Note that the `parprocs.h` header file includes the `ptarr23.h` header file. The `Array23Parm` array `A` is declared as external in lines 8 and 9 of `parmain.lst`.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: parmain.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // BACI Pascal Array Example
   2    0  // File: parrmain.pm
   3    0  //         the main program
   4    0
   5    0  #INCLUDE "parprocs.h"    // import the declarations of the array procs
>   1    0  {  BACI Pascal Array Example
>   2    0     File: parprocs.h
>   3    0              header file for storeArray usage
>   4    0  }
>   5    0
>   6    0  #INCLUDE "ptarr23.h"    { import the defintion of Array23Parm type }
>>   1    0  // BACI Pascal Array Example
>>   2    0  // File: ptarr23.h
>>   3    0  //          provides Array23Parm type definition
>>   4    0
>>   5    0  CONST
>>   6    0     dim1   = 2;
>>   7    0     dim1m1 = 1;
>>   8    0     dim2   = 3;
>>   9    0     dim2m1 = 2;
>>  10    0
>>  11    0  TYPE
>>  12    0     Array23Parm = ARRAY [0..dim1m1 , 0..dim2m1] OF INTEGER;
Returning to file parprocs.h
>   7    0
>   8    0  EXTERNAL PROCEDURE storeArray(VAR a : Array23Parm;
>   9    0     a00, a01, a02, a10, a11, a12 : INTEGER);
>  10    0     { store the 6 values in the Array23Parm array a }
>  11    0
>  12    0  EXTERNAL PROCEDURE showArray(u : Array23Parm);
>  13    0     { show u on stdout (but call it A) }
Returning to file parmain.pm
   6    0
   7    0  EXTERNAL VAR
   8    0     A : Array23Parm;
   9    0
  10    0  PROGRAM mainarray;
  11    0  BEGIN
  12    1     WRITELN("Storing values ... ");
  13    3     storeArray(A,11,22,33,99,88,77);
  14   13     WRITELN("Showing what was stored ... ");
  15   15     showArray(A);
  16   20  END.  // main
```

Note once again that the external declarations occur at the "global level", before the `PROGRAM` token occurs on line 10 of the `parmain.pm` file.

4.5.1 Using `bald` To Link The `.pob` Files

The three object files, `pardef.pob`, `parprocs.pob`, and `parmain.pob` are linked together with the command:

```
prompt% bald -m -o parexample pardef.pob parmain.pob parprocs.pob
Output link file stored in parexample.pco
Map file stored in parexample.map
```

The symbol map produced by the `-m` option is much like the previous symbol map that you have seen (including the lower-cased symbols):

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the parexample.pco link file
```

```
Files included in the link
index     file
0      parmain.pob
1      pardef.pob
2      parprocs.pob
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object        type          references
a                 variable   array      0 E  1 D
array23parm       type       array      0 D  1 D  2 D
dim1              constant   int        0 D  1 D  2 D
dim1m1            constant   int        0 D  1 D  2 D
dim2              constant   int        0 D  1 D  2 D
dim2m1            constant   int        0 D  1 D  2 D
mainarray         main proc  void       0 D
showarray         procedure  void       0 E  2 D
storearray        procedure  void       0 E  2 D
```

When the linked program is executed, the results are as shown in Section 4.2.1.

4.5.2 Using the `baar` Archiver To Simplify the Link

The only difference between linking `.pob` files produced by `bapas` and linking `.pob` files produced by `bacc` stems from the fact that Pascal is not case-sensitive and C is. The interaction with `baar` and `bald` is exactly as described in Section 4.2.2.

4.6 External Hoare Monitors in BACI Pascal

The example that we use to illustrate external Hoare monitors is the "classical" producer/consumer problem, in which producer processes store characters into a bounded buffer and consumer processes consume characters from the bounded buffer. Mutually exclusive access to the bounded buffer is managed by a Hoare monitor. The C−− version is discussed in Section 4.3.

This application is implemented in seven files:

pbbuff.pm

This file contains the bounded buffer management code, implemented in a Hoare monitor, `bounded_buffer`. The ten-character buffer uses two `INTEGER` variables, `nextIn` and `nextOut`, that give the locations in the buffer of the next character to be stored and the next character to be retreived. The monitor uses two conditions, `notFull` and `notEmpty`, to regulate the access of calling processes to the bounded buffer. an `INTEGER` variable. There are two externally visible monitor procedures, `append` and `retrieve` that the producer and consumer processes can call.

Let us briefly go though the code for the `append` procedure. The monitor variable `bufferCount` holds the number of characters currently in the buffer. On entry to the `append` procedure, if the buffer is full (`bufferCount = bufferSize`), then the calling process is put to sleep on the `notFull` condition. If there is space in the buffer (or when space becomes available in the buffer through the action of `retrieve` and a sleeping `append` caller is awakened by a signal to the `notFull` condition), the input parameter `c` is stored into the buffer. The `nextIn` index is moved to the next index in the buffer, and the `bufferCount` variable is incremented.

```
// BACI Pacal Monitor Example
// File: pbbuff.pm
//          Bounded buffer monitor

MONITOR bounded_buffer;
   CONST
      bufferSize    = 10;
```

```
      bufferSizem1  =  9;
   VAR
      buffer : ARRAY [0..bufferSizem1] OF CHAR;   // the character buffer
      nextIn : INTEGER;   // index of next character coming into buffer
      nextOut : INTEGER;  // index of next character going out of buffer
      bufferCount : INTEGER;  // number of characters in the buffer
      notEmpty : CONDITION;   // signalled when buffer is not empty
      notFull  : CONDITION;   // signalled when buffer is not full

   PROCEDURE append(c : CHAR);
      // append the character 'c' to the buffer
   BEGIN
      IF (bufferCount = bufferSize) THEN WAITC(notFull);
      buffer[nextIn] := c;
      nextIn := (nextIn + 1) MOD bufferSize;
      bufferCount := bufferCount + 1;
      SIGNALC(notEmpty);
   END;  // append

   PROCEDURE retrieve(VAR c : CHAR);
      // retrieve a character from the buffer into 'c'
   BEGIN
      IF (bufferCount = 0) THEN WAITC(notEmpty);
      c := buffer[nextOut];
      nextOut := (nextOut + 1) MOD bufferSize;
      bufferCount := bufferCount - 1;
      SIGNALC(notFull);
   END;  // retrieve

   BEGIN // init block
      nextIn := 0;
      nextOut := 0;
      bufferCount := 0;
END;  // bounded_buffer monitor
```

Finally, the `notEmpty` condition is signaled, in case processes calling `retrieve` were suspended on the `nonEmpty` condition. This signal is postponed to the last statement of `append` because of the "Immediate Resumption Requirement" used by the BACI PCODE interpreter. This requirement means that a process signaling a condition in a monitor is suspended so that any process sleeping on the condition can be resumed immediately (recall that only one thread of execution at a time can be active inside a Hoare monitor). Use of Immediate Resumption is based on the assumption that the condition that is signaled needs to be taken care of, so any process sleeping on the condition should have a higher priority of execution in the monitor than the signaling process.

The source code for `retrieve` is almost identical to the code of `append`, with only minor modifications. A reference variable is used to transfer the character removed from the buffer to the caller, rather than changing `retrieve` to a function of type `CHAR` and returning the character as the function's return value, because with the reference variable, the transfer occurs immediately with the reference variable, while the return statement, if it were added, would be delayed by the Immediate Resumption Requirement.

pprod.cm, pbbuff.h

The `pprod.lst` file, produced by the compiler, is shown instead of the `pprod.pm` file, because the compilation listing shows the `pbbuff.h` include file. Note that the `pbbuff.h` include file declares only the externally visible part of the `bounded_buffer` monitor, the `append` and `retrieve` procedures.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: pprod.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // BACI Pascal Monitor Example
   2    0  // File: pprod.pm
```

```
   3    0  //          the character producer
   4    0
   5    0  #INCLUDE "pbbuff.h"     // bring in the monitor
>   1    0  // BACI Pascal Monitor Example
>   2    0  // File: pbbuff.h
>   3    0  //          bounded buffer monitor
>   4    0
>   5    0  EXTERNAL MONITOR bounded_buffer;
>   6    0     PROCEDURE append(c : CHAR);       // append the character 'c' to the buffer
>   7    0     PROCEDURE retrieve(VAR c : CHAR); // retrieve a character from the buffer into 'c'
>   8    0  END;  // bounded_buffer monitor
Returning to file pprod.pm
   6    0
   7    0  EXTERNAL VAR
   8    0     mutex : BINARYSEM;   // unscramble screen output
   9    0
  10    0  PROCEDURE producer(c : CHAR);
  11    0     // appends 'c' to the buffer forever
  12    0  BEGIN
  13    0     WHILE (TRUE) DO
  14    2     BEGIN
  15    2        append(c);
  16    6        P(mutex);
  17    8        WRITELN("producer ",c," appended ",c);
  18   15        V(mutex);
  19   17     END;
  20   18  END;  // producer
```

The `producer` procedure simply calls `append` to add its `CHAR` parameter `c` to the buffer forever. Because the main program (`pprodcon.pm`, see below) starts multiple `producer` and `consumer` processes, screen output by either process has to be serialized through the use of a binary semaphore, `mutex`, declared and initialized in the main program.

pcons.pm,pbbuff.h

The code for the `consumer` procedure is just like the code for the `producer` procedure, except that `retrieve` is called, instead of `append`.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: pcons.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // BACI PascalMonitor Example
   2    0  // File: pcons.pm
   3    0  //          the character consumer
   4    0
   5    0  #INCLUDE "pbbuff.h"     // bounded buffer monitor declarations
>   1    0  // BACI Pascal Monitor Example
>   2    0  // File: pbbuff.h
>   3    0  //          bounded buffer monitor
>   4    0
>   5    0  EXTERNAL MONITOR bounded_buffer;
>   6    0     PROCEDURE append(c : CHAR);       // append the character 'c' to the buffer
>   7    0     PROCEDURE retrieve(VAR c : CHAR); // retrieve a character from the buffer into 'c'
>   8    0  END;  // bounded_buffer monitor
Returning to file pcons.pm
   6    0
   7    0  EXTERNAL VAR
   8    0     mutex : BINARYSEM;   // unscramble screen output
   9    0
  10    0  PROCEDURE consumer(c: CHAR);
  11    0     // consumes characters from the buffer  (has ID 'c')
  12    0     VAR
  13    0        d : CHAR;
  14    0  BEGIN
  15    0     WHILE (TRUE) DO
  16    2     BEGIN
  17    2        retrieve(d);
  18    6        P(mutex);
  19    8        WRITELN("consumer ",c," retrieved ",d);
```

```
 20   15        V(mutex);
 21   17     END;
 22   18  END;  // consumer
```

pprodcon.pm,pprod.h,pcons.h

This file contains the source code for the main program. The two include files declare the prototypes of the `producer` and `consumer` procedures.

```
BACI System: BenAri Pascal PCODE Compiler, 12:57   3 Aug 2001
Source file: pprodcon.pm  Wed Nov 14 17:25:36 2001
 line  pc
   1    0  // BACI Pascal Monitor Example
   2    0  // File: pprodcon.pm
   3    0  //          the producer-consumer main program
   4    0
   5    0  #INCLUDE "pprod.h"
>   1    0  // BACI Pascal Monitor Example
>   2    0  // File: pprod.h
>   3    0  //          the character producer
>   4    0  EXTERNAL PROCEDURE producer(c : CHAR);
>   5    0     // appends 'c' to the buffer forever
Returning to file pprodcon.pm
   6    0  #INCLUDE "pcons.h"
>   1    0  // BACI Pascal Monitor Example
>   2    0  // File: pcons.h
>   3    0  //          the character consumer
>   4    0  EXTERNAL PROCEDURE consumer(c : CHAR);
>   5    0     // consumes characters from the buffer  (has ID 'c')
Returning to file pprodcon.pm
   7    0
   8    0  VAR
   9    0     mutex : BINARYSEM;   // to unscramble screen output
  10    0
  11    0  PROGRAM ProdCons;
  12    0  BEGIN
  13    1     INITIALSEM(mutex,1);
  14    4     COBEGIN
  15    5        producer('A'); producer('B'); producer('C');
  16   17        producer('D'); producer('E'); producer('F');
  17   29        consumer('1'); consumer('2'); consumer('3');
  18   41     COEND;
  19   42  END.  // ProdCons
```

The main program declares and initializes the `mutex` binary semaphore that the `producer` and `consumer` procedures use to keep their output to the terminal screen from intermixing.

In the `COBEGIN` block, the program starts six `producer` processes and three `consumer` processes. One could add debugging output to the `bounded_buffer` monitor to discover that in this case, the buffer will stay full most of the time, with `producers` waiting to add their characters. If there were more `consumer` processes than `producer` processes, then the buffer would be empty most of the time, with `consumer` processes waiting to remove characters.

4.6.1 Using `bald` To Link The `.pob` Files

The four object files, `pbbuff.pob`, `pprod.pob`, `pcons.pob`, and `pprodcon.pob` are linked together with the command:

```
prompt% bald -m -o pproconx pbbuff.pob pprod.pob pcons.pob pprodcon.pob
Output link file stored in pproconx.pco
Map file stored in pproconx.map
```

Execution of the linked program yields:

```
prompt% bainterp pproconx | less
Linked files: pbbuff.pob pprod.pob pcons.pob pprodcon.pob
Executing PCODE ...
producer E appended E
producer B appended B
consumer 3 retrieved E
producer A appended A
producer D appended D
consumer 2 retrieved B
producer E appended E
producer B appended B
consumer 3 retrieved A
producer F appended F
producer C appended C
producer B appended B
consumer 2 retrieved D
producer E appended E
consumer 3 retrieved E
producer F appended F
producer D appended D
producer E appended E
producer D appended D
consumer 1 retrieved B
producer A appended A
producer D appended D
producer A appended A
producer F appended F
consumer 3 retrieved F
producer C appended C
consumer 1 retrieved B
producer A appended A
consumer 3 retrieved E
consumer 2 retrieved C
producer F appended F
consumer 3 retrieved F
producer E appended E
producer B appended B
consumer 2 retrieved D
consumer 3 retrieved E
producer A appended A
producer B appended B
producer F appended F
consumer 3 retrieved D
consumer 1 retrieved D
consumer 2 retrieved A
producer A appended A
producer E appended E
consumer 3 retrieved A
...
```

You may find the above output confusing, because characters seem to leave the buffer in a different order than they were entered. If you start from the top of the output, the first seven characters appended to the buffer are `E`, `B`, `A`, `D`, `E`, `B`, and `F`. The first seven characters retrieved from the buffer match these characters exactly. However, the eighth, ninth, and tenth characters added to the buffer are apparently `C`, `B`, and `E`, in that order, yet the the eighth, ninth, and tenth characters retrieved from the buffer are `B`, `E`, and `C`.

You can add debugging output to the `bounded_buffer` monitor (as we did) to discover that the program output can occur in a different order from the order that the characters enter and leave the buffer. In this particular case, the order that the eighth, ninth, and tenth characters were added to the buffer is exactly the same as the order in which they were retrieved. However, the `append` call from the `C` producer reached the `mutex` critical section before the `append` calls from the `B` and `E` producers did, so the `C` producer's output appeared first. Consequently, the `B` producer process was restarted from the `mutex` queue before the `E` was, even though these producers had appended their characters in the opposite order.

Recalling the source code above, the `producer` and `consumer` procedures both have non-terminating loops, so this program must be terminated by some external influence. In this case, the program output was piped (in LINUX) into the `less` pager, so that the `bainterp` program could be terminated by terminating the pager.

4.6.2 Using baar To Simplify The Link

In this example, we combine the `pbbuff.pob`, `pprod.pob`, and `pcons.pob`, into a library file `libpprco.ba` with the command:

```
prompt% bald -m -o pprocox2 pprodcon.pob -lpprco
Output link file stored in pproconx.pco
Map file stored in pproconx.map
```

The execution of the `proconx2.pco` linker output file is much like what you have seen above.

5 Linking .pob Files Produced by BACI Pascal and C−−

The `.pob` files produced by the BACI Pascal and C−− compilers can be combined, but to do so successfully, you must be aware of two important facts:

1. **All identifiers in a BACI Pascal .pob file are in lower case**.

   Since Pascal is not case-sensitive (unlike C), the BACI Pascal compiler lower-cases all identifiers to facilitate parsing an input file. So, if you plan to call routines written in BACI Pascal from a C−− program, the subroutine name that you use should be in lower case, no matter what the subroutine name looks like in the BACI Pascal source. If you plan to call routines written in C−− from a BACI Pascal program, then you can use any combination of upper and lower case in the header files declaring the external references, since the BACI Pascal compiler will lower-case the names during compilation.

2. **The index of the first element in a C array is always zero**.

   As you probably know, the C language uses zero-indexing for all arrays (the index of the first element in the array is zero), whereas in Pascal, the index set of an array can be any finite set of consecutive integers. This means that in a BACI Pascal program, zero-indexing must be used with any array declared in a C−− source file. This also means that in a C−− program, you cannot use arrays declared in BACI Pascal that are not zero-indexed.

5.1 Using a Library Produced by BACI Pascal With a C−− Program

Because of the way the bounded buffer examples in C−− in Section 4.3 and BACI Pascal in Section 4.6 are constructed, we will be able to use the `libpprco.ba` library produced in Section 4.6.2 when linking the `prodcons.pob` file produced from the `prodcons.cm` source file in Section 4.3.

The following `bald` command performs the link:

```
prompt% bald -m -o cxplib prodcons.pob -lpprco
Output link file stored in cxplib.pco
Map file stored in cxplib.map
```

The `-m` option in the command produces the following symbol map:

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the cxplib.pco link file
Files included in the link
index     file
0      prodcons.pob
```

```
1      pcons.pob (./libpprco.ba)
2      pprod.pob (./libpprco.ba)
3      pbbuff.pob (./libpprco.ba)
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object        type          references
append       mon. procedure  void       1 E  2 E  3 D
bounded_buff      monitor    void       1 E  2 E  3 D
consumer          procedure  void       0 E  1 D
main              main proc  void       0 D
mutex             variable   binarysem  0 D  1 E  2 E
producer          procedure  void       0 E  2 D
retrieve     mon. procedure  void       1 E  2 E  3 D
```

As you can see from the symbol map, the two external references in the `prodcons.cm` file, `producer` and `consumer`, were found in the files `pprod.pob` and `pcons.pob` in the BACI Pascal library `libpprco.ba`. Construction of this library is described in Section 4.6.2. These two files contain external references to the `bounded_buff` monitor and its two procedures `append` and `retrieve`, so the linker has also incorporated the file `pbbuff.pob` from the library `libpprco.ba` to resolve those references. The external references to the `mutex` binary semaphore in the `pprod.pob` and `pcons.pob` files were resolved by the definition of `mutex` in the `prodcons.pob` file.

This link is successful because the external references in the C−− program are in lower case, which is how the BACI Pascal has compiled the files included in the `libpprco.ba` library. Zero array indexing doesn't really figure in this example, because array used in the BACI Pascal implementation of the bounded buffer monitor is an internal data structure of the monitor and is not externally visible. This particular array does use zero-indexing to simplify the array index calculations, rather than to supply compatibilty.

Execution of the linked program, `cxplib.pco`, produces output similar to the executions shown in Sections 4.3.1 and 4.6.1.

5.2 Using a Library Produced by C−− with a BACI Pascal Program

If you have read the previous section, you have probably guessed correctly that it is possible to link the producer/consumer main program written in BACI Pascal with the `libprcon.ba` library composed of `.pob` files created by C−−.

The following `bald` command performs the link:

```
prompt% bald -m -o pxclib pprodcon.pob -lprcon
Output link file stored in pxclib.pco
Map file stored in pxclib.map
```

The `-m` option in the command produces the following symbol map:

```
BACI System:  PCODE Linker 12:57   3 Aug 2001
Symbol map for the pxclib.pco link file
Files included in the link
index     file
0      pprodcon.pob
1      cons.pob (./libprcon.ba)
2      prod.pob (./libprcon.ba)
3      bbuff.pob (./libprcon.ba)
List of symbol references
  ('E' = external, 'D' = defined, 'U' = unknown)
  the integer shown is an index into the link file list above
  name         object        type          references
append       mon. procedure  void       1 E  2 E  3 D
bounded_buff      monitor    void       1 E  2 E  3 D
consumer          procedure  void       0 E  1 D
mutex             variable   binarysem  0 D  1 E  2 E
prodcons          main proc  void       0 D
producer          procedure  void       0 E  2 D
retrieve     mon. procedure  void       1 E  2 E  3 D
```

The analysis of the symbol map is much like the analysis in the previous section and won't be repeated here.

As in the previous section, this link is successful because the external references in the BACI Pascal program get lower-cased by the compiler. These symbols are defined in lower case in the C−− source files used to create the the `prod.pob`, `cons.pob`, and `bbuff.pob` files of the `libprcon.ba` library. Construction of this library file is described in Section 4.3.2. As before, zero array indexing doesn't really figure here, because the array used in the C−− implementation of the bounded buffer monitor is an internal data structure of the monitor and is not externally visible.

Execution of the linked program, `pxclib.pco`, produces output similar to the executions shown in Sections 4.3.1 and 4.6.1.