C - Cedar/Mesa Interoperability

Author: Raymond T. Li (RLi:ESCP10:Xerox, MS: ESCN-102, Phone: 8*823 - 7370)

Date: Aug 7, 1992

Filed: [DeAnza:PARC:Xerox]/Integration Tools/C-CEDAR/MESA INTEROPERABILITY

Version: 1.0

Revision history:

XEROX

Advanced Technology & Competency Development
Systems Competency
Imaging Platform Development
El Segundo, Ca.

ACKNOWLEDGEMENTS

There were many people who provided the support needed to create this document. I would especially like to acknowledge the following individuals:
Hans Boehm, John Corwin, Jim Foote, Peter Hantos, Tom Horton, Alan Ishigo, Christian Jacobi, Swen Johnson, Sandy Kobayashi, Gene McDaniel, Michael Plass, Xavier Riley, Mike Spreitzer, Marvin Theimer, John Wenn, Marjan White and all the **PARC folks** who reviewed the material and provided critical feedback on the Pre-Release Draft of this document.

TABLE OF CONTENTS

LEGEND: For an explanation of the "⊕" and "⊘" symbols, please refer to Section 1.2.

C - Cedar/Mesa Interoperability

1.0 ⊕⊘ Introduction

There has been much effort towards developing mixed-language (Cedar/Mesa and C programming language) software systems within the Cedar environment. This emphasis can be attributed to three factors. First, management has decided to take advantage of the huge existing base of Mesa code. For example, there are approximately 1.5 - 2 million lines of Mesa code (400+ work years of labor) associated with DocuTech which could be reused. The use of this code necessitates the additional layer of runtime support provided by the Cedar environment. Second, usage of industry-standard programming languages like C for new code development have been emphasized over Xerox-proprietary Mesa language in recent years. Finally, systems have made increased use of commercially purchased C-based third-party software.

1.1 ⊕⊘ Purpose

The purpose of this document is to present issues involved in engineering mixed-language (Cedar/Mesa and C programming language) software systems within the Cedar environment. An example of such a system is PCL5 1.0. Designed to interpret and decompose HP PCL5 page description language (PDL) into a format for printing or display, this software combines the C-based PCL5 decomposer with Cedar/Mesa based imaging and font support facilities.

The development of software systems has often been driven by the creation of interface specifications between functional components. Although this activity plays an important role in the engineering process, it is not sufficient for development of software within the Cedar environment. Sole use of interfaces for system development neglects to recognize the large roles played by the Cedar runtime environment (PCR) and the UNIX operating system. The in-depth discussion of the Cedar environment should provide an understanding of the complexities of engineering such a software system.

The document will also provide application programmers with a methodology for porting C software to this platform.

1.2 ⊕⊘ Intended Audience

The intended audience of this document can be divided into two categories, architects and programmers. The former are responsible for the high-level system design while the latter perform the implementation.

Sections of this document may only be applicable to one category of readers. In order to allow readers quick access to material of interest, presence of the character ⊕ preceding the section header text denotes recommended reading for architects and ⊘ denotes material aimed at programmers. Note that the categorization of each section reflects the author's views and as such are subjective.

It is assumed that readers are familiar with the Cedar/Mesa and C programming languages and have knowledge of basic operating system concepts. They should also have access to the Cedar environment in order to facilitate access to referenced reading materials and examples.

1.3 ⊕⊘ Document Scope

Material covered in this document are geared toward the Cedar environment running atop SunOS 4.1. With the transition of SUN SPARC stations to SunOS 5.0 in the near future, the Cedar environment software is undergoing major revisions. Facilities provided by the new release will definitely change to take advantage of the capability of SunOS 5.0 to handle multiple threads. This may invalidate information provided in sections 2 and 4 of this document, at the very least.

1.4 ⊕⊘ Organization of this document

Section 2 provides an in-depth discussion of the Cedar environment and the underlying Portable Common Runtime software. Section 3 describes the syntax required for calling procedures written in another language, methods of incorporating C & Cedar/Mesa modules together (configurations, packages), Remote Procedure Calls and debugging of Cedar-based systems. Many examples are presented for reinforcement purposes, including a detailed case study of the Wasabi 1.0 Postscript Decomposer. Section 4 describes some of the problems involved with porting C code to the Cedar environment and provides a protocol for doing the porting process. The remainder of the document consists of these appendices:

Appendix A: Table of UNIX system call replacements
Appendix B: Table of unimplemented UNIX system calls
Appendix C: Selected reference materials

1.5 ⊕⊘ References

The asterisk character denotes the presence of this article within Appendix C.

-- Portable Cedar Basement

*[1] *An Introduction to the Portable Cedar Basement Documentation* Weiser,M. April 26, 1989 Documentation roadmap for PCR, CedarCore and BasicCedar
Filed as: [PCedar2.0]/Documentation/PCedarBasementIntroduction.tioga

*[2] *An Overview of the Portable Cedar Basement* Hauser,C. & Weiser,M. April 26, 1989 High-level introduction to the concepts and facilities of the Portable Cedar system.
Filed as: [PCedar2.0]/Documentation/PCedarOverview.tioga

*[3] *The Portable Common Runtime Approach to Interoperability* Weiser,M., Demers,A. & Hauser,C. May, 1989 Covers PCR design principles and implementation.
Filed as:
[PCedar2.0]/Documentation/PortableCommonRuntimeApproach.tioga

*[4] *Experiences Creating a Portable Cedar* Atkinson,R., Demers,A., Hauser,C., Jacobi,C., Kessler,P. & Weiser,M. March 23, 1989 Strategy used to make Cedar language portable across many different architectures
Filed as: [Cedar10.1]/CedarDoc/ExperiencesCreatingAPortableCedar.tioga

-- Cedar/Mesa Language

[5] *Mesa Language Manual. Version 6.0* October, 1988 Reference manual for Mesa, a subset of Cedar language
Filed as: [BitBucket:OSBU North:Xerox]/Release 6.0/Text/Mesa Language Manual

[6] *Cedar Language Overview* Foote,J., et al. May 13, 1992, Describes the Cedar language as an extension of Mesa
Filed as: [Cedar10.1]/CedarDoc/LanguageOverviewDoc.tioga

[7] *Cedar/Mesa Language Changes* Atkinson,R. Aug 26, 1991 Summarizes additions to the Cedar language to make it portable to other machine architectures, eg, SPARC/PCR platform
Filed as: [Cedar10.1]/MimosaOnly/LanguageChangesSummary.tioga

-- Cedar/C Interoperability

*[8] *C2CInterLanguageDoc.tioga* Jacobi,C. May 13, 1991 Discuss ways Cedar modules can access C procedures and vice versa.
Filed as: [Cedar10.1]/C2C/C2CInterLanguageDoc.tioga

*[9] *Salient Mesa and C Intercallability* Litman Jan 22, 1988 Describes intercallability between Mesa and C, as supported by Salient language tools

[10] *UXStrings.mesa* Cedar/Mesa interface for conversion of strings between UNIX format and Cedar representation, ROPE. Filed as: [Cedar10.1]/UXStrings/UXStrings.mesa

UXProcs.mesa Cedar/Mesa interface for conversion of procedure variables between Cedar and C programs
Filed as: [Cedar10.1]/UXStrings/UXProcs.mesa

InstallationComfortsDoc Hauser,C. & Kessler,P. May 13, 1991 Describe interfaces that provide the ability to bind to procedures in the run-time Cedar/Mesa interfaces.
Filed as:
[Cedar10.1]/InstallationComforts/InstallationComfortsDoc.tioga

-- Configurations, Packages

*[11] *Using Cinder* Swinehart, Subhana, Litman,A., Foote,J. May 20, 1991 Among other things, it describes C/Mesa language construct to facilitate building multiple language configurations.
Filed as: [Cedar10.1]/Cinder/CinderDoc.tioga

*[12] *Building A Packaged World* Weiser,M. Jan. 1, 1991 Provides a step-by-step description for building a packaged PCedar world
Filed as: [PCedar2.0]/Documentation/BuildingAPackagedWorld.tioga

*[13] *PCRDoc --Portable Common Runtime* Atkinson,R. May. 24, 1989 Describes PCR operating switches
Filed as: [PCedar2.0]/Documentation/PCRDoc.tioga

-- Remote Procedure Calls

[14] *Network Interfaces Programmer's Guide, chapter 2,3 & 4* Sun Microsystems Introduces the concepts of remote procedure calls and the use of the "rpcgen" compiler to help programmer's write RPC applications

*[15] *CedarRPCGenDoc* Theimer,M., Orr,WS May 26, 1992 Describes program that accepts interface specifications in Sun's RPC language and emits stubs written in Cedar. The stubs allows C and Cedar programs to communicate via remote procedure call using Sun's RPC protocol.
Filed as: [Cedar10.1]/CedarRPCGen/CedarRPCGenDoc.tioga

*[16] *Sun RPC Runtime* Demers,A., Hauser,C., Plass,M. Jan 7, 1992 Describes general structure of Cedar/Mesa-based Sun RPC remote programs.
Filed as: [Cedar10.1]/SunRPCRuntime/SunRPCRuntimeDoc.tioga

[17] *Example of a Cedar/Mesa-based Sun RPC remote program: SunEcho*
Filed as: [PCedar2.0]/Documentation/SunEchoDoc.tioga,[PCedar2.0]/SunEcho/*

-- Debugging

[18] *Debugging Tools Manual* Sun Microsystems Describes the standard UNIX dbx and dbxtool debuggers

*[19] *The Cirio Debugger* Sturgis,H., Theimer,M. and others June 12, 1992 User's guide to the Cirio debugger
Filed as: [Cedar10.1]/Cirio/CirioDoc.tioga

-- Porting process

*[20] *Experiences from the Salient project group on porting Viewpoint to run on UNIX-based SUN workstations*

*[21] *UnixSysCalls* Hauser,C., Demers,A., Weiser,M., Plass.M. Aug 7, 1991 Summarizes the UNIX system call interface for Cedar/Mesa programmers.
Filed as: [Cedar10.1]/UnixSys/UnixSysCallsDoc.tioga

[22] *Cedar/Mesa interfaces and implementation modules for Unix system calls.*
Filed as: [Cedar10.1]/UnixSys/UnixSysCalls.mesa,[PCedar2.0]/UnixSys/*.mesa

[23] *PCR C interfaces for Unix system calls, threads (monitors)*
Filed as: [PCR dir]/INCLUDE/xr/UIO.h, [PCR dir]/INCLUDE/xr/Threads.h

-- Miscellaneous

[24] *Cheops Principles of Operation version 1.71*
Filed as: [Enterprise:ESCP10:Xerox]/IPD Projects/ver 1.71 Cheops principles of Operation(ip).

2.0 ⊕⊘ Cedar Environment

The Cedar environment was built beginning in 1978 by the CSL organization to support the Cedar/Mesa programming language on the Dorado workstation. For many years this language and environment ran only on these D-machines; an effort later began to make it portable to different machine architectures. The resulting portable Cedar environment, named PCedar, is able to run on operating systems such as SUNOS Unix (version 4.0), Mach and on a bare homebuilt machine serving as the main operating system. Distinctions between PCedar and Cedar have now disappeared; references to the Cedar environment implies portability.

2.1 ⊕⊘ General Overview

The Cedar environment is a software layer which lies atop a SUN SPARC workstation running UNIX. It provides run-time support facilities for the Cedar/Mesa programming language, although it's services are language-independent. It has been used successfully with C, Scheme, Modula3 and Lisp.

This environment is composed of several layers. The lowest layer is called the Portable Common Runtime (PCR). It is written in primarily C code (with a small amount of assembly code) and provides a language and operating system independent base for modern languages, although it's primary use is to support the Cedar/Mesa programming language. The runtime support facilities offered by PCR include light-weight processes (multiple thread support), low-level I/O, garbage collection and dynamic loading of modules. A detailed discussion of the PCR implementation occurs later in this document; this knowledge is necessary in understanding the complexities of porting multi-language software systems to the Cedar environment.

The final layers of the Cedar environment specifically provide support for Cedar/Mesa programming. The software immediatedly above PCR, CedarCore, provides full functionality for the language itself. It supports language constructs such as SIGNALS, PROCESS, ROPE and low level arithmetic, logical and storage overlay operations. The top layer, BasicCedar, contains services considered essential for most Cedar-based applications. Services include debugging support, hash tables, time functions, IO, additional ROPE support and access to the UNIX operating system. In addition, there are many available software packages that provide specialized services to the user application program (referred to as Miscellaneous Cedar Run-Time code).

Figure 2.1 illustrates the Cedar environment sitting above the UNIX operating system. It shows the functional components comprising the Portable Common Runtime, CedarCore and BasicCedar layers.

PORTABLE CEDAR BASEMENT INTERNALS

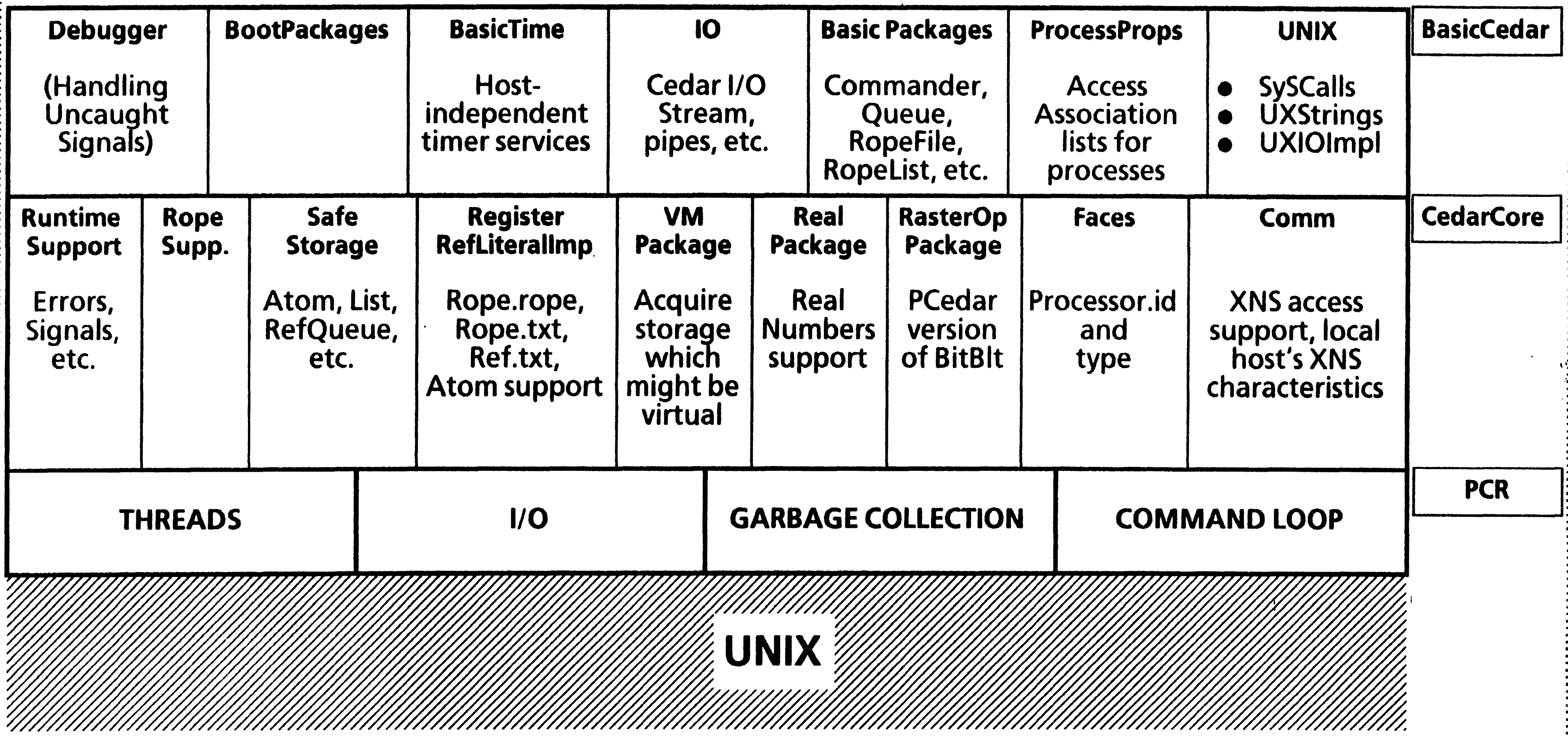


Figure 2.1

2.2 ⊕⊘ Portable Common Runtime

A process or task is defined as an instance of a program in execution. Associated with each process is an address space that represents the set of all addresses available to the program. In the traditional UNIX environment, each process (heavyweight process) consists of a single locus (thread) of control.

The Cedar environment, namely the PCR layer, implements the concept of threads (lightweight process) within the UNIX environment. The PCR approach is to have a small fixed number of processes, called "virtual processors" (VPs), that act like virtual CPUs and can support multiple threads of control. Although seen by the underlying UNIX kernel as a single process, each VP can spawn off multiple execution threads and schedule processing time for each thread in a time-sharing manner. Each thread runs strictly sequentially and has it's own program counter and stack information; however, all threads share the same address space (global variables are shared) and other process resources such as open files, UNIX timers, UNIX signals (signal handlers). It should be noted that threads aren't bound to a VP but are collectively supported by all VPs. When a thread becomes ready, it gets scheduled on the next available VP.

The ability for VPs to support multiple threads simulate parallel processing and are ideal for systems running on a multiprocessor hardware platform. It is also well suited for applications involving a high degree of concurrent processing such as in server/client relationships.

PCR supports automatic deallocation of dynamic storage (garbage collection). This functionality prevents problems associated with dangling pointers (pointer used after the referenced storage has been deallocated) and storage leaks (storage becomes inaccessible without being deallocated for reuse). In addition, symbol table management capabilities exist to support the static/dynamic linking and loading of modules.

PCR provides additional heavyweight processes called I/O processors (IOPs). Some are associated with file descriptors and used to circumvent the limitations imposed by UNIX on the maximum number of open files available to a process; these are classified as either Streams IOP or Standard IOP. The former is implemented with System V style streams while the latter possess the standard UNIX style device driver implementation. Other IOPs (Slave IOP) aren't associated with file descriptors; an example is the CirioNub program which supports debugging in the Cedar Environment.

The user can provide the PCR software with operating parameter values upon startup (a list of all available parameters is provided in [Ref 13]), including the number of virtual and IO processors. A minimal PCR environment should consist of 4 UNIX processes: 1 VP, 1 Slave IOP, 1 Standard IOP and 1 Stream IOP). More stream IOPs are needed if the application utilizes greater than 30 open streams simultaneously; likewise, additional standard IOPs are needed if you require more than 256 open file descriptors.

To summarize, PCR contains two types of processes, VPs and IOPs. Each VP is capable of running multiple threads; including threads to perform garbage collection, dynamic loading and execution of user application code. Threads aren't fixed to a particular VP and share a common virtual address space; this shared space significantly reduces the cost of complex interactions and data exchanges required by the PCR implementation.

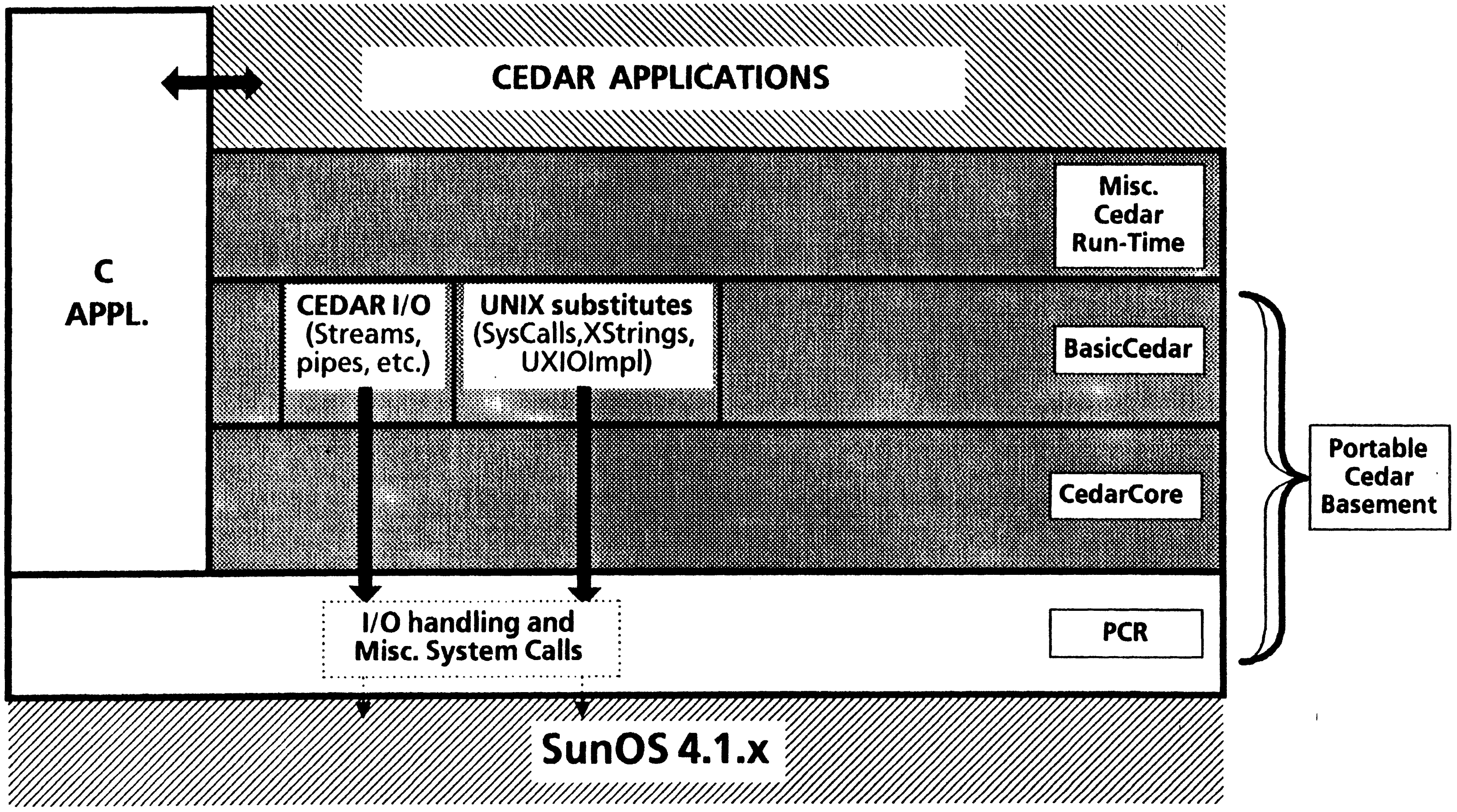
2.3 ⊕⊘ Applications within the Cedar environment

Figure 2.2 illustrates the configuration occupied by a Cedar/Mesa and C software application within the Cedar environment.

Besides providing a pictorial representation of the layers comprising the Cedar environment, it shows that Cedar/Mesa code is supported by the CedarCore, BasicCedar and Misc. Cedar Run-Time layers. Implementation within these modules are ultimately handled by the PCR and SunOS software lying below (vertical black arrows). On the other hand, C-based code are supported by the facilities that PCR and SunOS provides.

With the ability for C and Cedar/Mesa modules to interact using communication techniques discussed in section 3.1 and 3.2, C code can also access the Cedar/Mesa based services offered by the CedarCore, BasicCedar and Misc. Cedar Run-Time layers. This ability to interact extends to Cedar/Mesa programs utilizing C based services; in fact, most Cedar/Mesa based BasicCedar routines for accessing UNIX system calls are basically veneers for the underlying C-based PCR function. Despite the wide opportunities for interoperability using system services, most such interactions will exist between Cedar/Mesa and C based user-application programs, as shown by the horizontal black arrow.

PCR-BASED SOFTWARE ARCHITECTURE


CEDAR APPLICATIONS
C APPL.
Misc. Cedar Run-Time
CEDAR I/O (Streams, pipes, etc.)
UNIX substitutes (SysCalls,XStrings, UXIOImpl)
BasicCedar
CedarCore
Portable Cedar Basement
I/O handling and Misc. System Calls
PCR
SunOS 4.1.x

CODE SIZES: Misc. Cedar Run-Time/BasicCedar/CedarCore: ≈ 70,000 lines of **Cedar**,
PCR: ≈ 60,000 lines of **C** code

Figure 2.2

3.0 ⊕⊘ Syntax/Mechanics of Cedar/C Interoperability

In order to fully understand the mechanics of the Cedar / C interoperability issue, it is necessary to explain that Cedar based code is translated by the Mimosa compiler into machine-dependent C code that is very efficient, difficult to read and normally unseen by the programmer [Ref 4]. This intermediate C code is then fed to the target machine's C compiler to generate the executable object file. This translation process provides the means by which Cedar/C interoperability can occur.

3.1 ⊘ Calling C procedures from within a Cedar/Mesa module

Support for calling a C procedure within Cedar/Mesa is provided by the inclusion of "MACHINE CODE" procedures within the language. The syntax for declaring a "MACHINE CODE" procedure is quite similar to that of other procedures and includes the following elements (If unfamiliar with declaring procedures, refer to chapter 5 of the Mesa Language Manual version 6.0 [Ref 5]):

a. MACHINE CODE (MC) procedure name

b. Procedure Type: includes parameters required and results returned by the C procedure. Field specifications for the parameter and result records follow the standard Cedar/Mesa language syntax. However, each field type must correspond bitwise to the type expected by the underlying C procedure.

   There are several cases where this correspondence isn't trivial. First, if the C procedure expects a "UNIX string" (pointer to a NUL terminated array of chars, char *), the MC procedure must declare the corresponding parameter as type "UXStrings.UnixString". Second, if the C procedure expects a procedure variable, the MC procedure must declare the corresponding parameter as type "UXProcs.CProc".

c. Immediatedly following the "=" character with the string "TRUSTED MACHINE CODE" initializes and defines the procedure

d. Procedure Body: consists of a series of one or more interpreted string literals separated by a semicolon. There are numerous valid constructs within a string literal; a detailed discussion can be found in the C2CInterLanguageDoc.tioga document [Ref 8].

Illustrated below are examples of procedure bodies for a Cedar/Mesa procedure Proc to access C procedure C_Proc.

```
void C_Proc (character, num)
   char character;   int num;
{ /* This is the C procedure which is being called */ }


-- Cedar/Mesa procedure for calling C_Proc
-- This should account for the vast majority of cases
Proc: PROC [c: CHAR, n:INT] = TRUSTED MACHINE CODE {
  "C_Proc"
};

-- Cedar/Mesa procedure for calling a C_Proc requiring an
-- include file "inc.h" from Cedar include directory
-- /usr/include/cedar
Proc: PROC [c: CHAR, n:INT] = TRUSTED MACHINE CODE {
  "inc.C_Proc"
};

-- Cedar/Mesa procedure for calling a C_Proc requiring an
-- include file "inc.h" from UNIX include directory
Proc: PROC [c: CHAR, n:INT] = TRUSTED MACHINE CODE {
  "<inc.h>.C_Proc"C_Proc
};

-- Cedar/Mesa procedure for calling a C_Proc requiring an
-- include file "inc.h" from current working directory
Proc: PROC [c: CHAR, n:INT] = TRUSTED MACHINE CODE {
  " \"inc.h\".C_Proc"
};
```

There are many valid constructs supported within the procedure body of a 'MACHINE CODE' procedure (a detailed discussion of the procedure body syntax can be found in the C2CInterLanguageDoc.tioga document [Ref 8]). These allow the Cedar programmer to call C procedures, include C header files, define symbolic constants and macros, insert C code into the intermediate machine-based C code produced during translation (eg, introduce variables, procedures), invoke defined macros and introduce constant and variable values into Cedar expressions.

The following examples illustrate some permutations that the procedure body can assume:

- o File: /PCedar2.0/basicpackages/CommanderImpl.mesa
  "MACHINE CODE' procedure name: getenv
- o File: /PCedar2.0/bootpackages/TerminationImpl.mesa
  "MACHINE CODE' procedure name: XRExitWorld
- o File: /PCedar2.0/CedarPreBasicsExtras/XRLoaderUtils.mesa
  "MACHINE CODE' procedure name: RequireFrom
- o File: /PCedar2.0/CFontSolution/CFontSolutionImpl.mesa
  "MACHINE CODE' procedure name: GetMaskInner
- o File: /PCedar2.0/DReal/DRealFnsImpl.mesa
  "MACHINE CODE' procedure name: DRealExpl
- o File: /PCedar2.0/FloatingPoint/DRealSupportImpl.mesa
  "MACHINE CODE' procedure name: help

To invoke the C procedure 'C_Proc' defined by the example on the previous page, these statements must be included within the Cedar/Mesa program:

```
int: INT;
character: CHAR;
Proc[character, number];
```

Invoking the C procedure involves specifying the "MACHINE CODE" procedure name and supplying arguments for each parameter. Arguments are passed by value and not reference. The type of each argument value must be compatible with it's corresponding parameter type. In most cases, this presents no problem; in some situations, the supplied argument value must be converted to a compatible form beforehand. The next few paragraphs present cases where this conversion is required.

C procedure expects a "Unix compatible string" (char *) as an argument: Since Cedar/Mesa programs deal with the "ROPE" construct instead, there must exist a "transducer" which converts any ROPE values to an equivalent Unix-compatible value prior to the invocation process. This conversion process is handled by utilizing the "Create" procedure within the "UXStrings.mesa" interface. Likewise, any "Unix compatible strings" returned by the resulting MC procedure must be transformed by the UXStrings.ToRope procedure before this data can be used by the remainder of the Cedar/Mesa program. Refer to these Cedar modules from the /PCedar2.0 directory for real, on-line examples:
- o Cirio/CirioDeltaFaceSunImpl.mesa
- o LoadstateTool/LoadstateToolImpl.mesa
- o NetworkStream/NetworkStreamImpl.mesa

C procedure expects a procedure variable argument: The Cedar procedure argument must be compatible with the procedure type expected by the invoked C function in terms of the parameters and results. It must be transformed by the "FromCedarProc" procedure defined within the "UXProcs.mesa" interface into a form (UXProc.CProc) which can then utilized by the invoked C procedure. Likewise, the UXProcs interface defines a procedure, "ToCedarProc" that converts C procedure variables into Cedar procedure values; clients can then LOOPHOLE this into the right procedure type. Refer to these Cedar modules from the /PCedar2.0 directory for real, on-line examples:
- o Compiler/ADotOutExtrasAccessImpl.mesa
- o GCOps/GCCallBackImpl.mesa
- o TCL/TCL.mesa

Here is an example of a Cedar/Mesa procedure calling a C routine; it is source code for the Cedar debugger, Cirio.

Cedar/Mesa file: CirioDeltaFaceSunImpl.mesa (caller)

```
DIRECTORY
 -- This interface converts UNIX strings to ROPE and vice versa
 UXStrings USING [ToRope, UnixString];

CirioDeltaFaceSunImpl: CEDAR PROGRAM
IMPORTS UXStrings -- list of imported interfaces
EXPORTS -- list of exported interfaces -- =
BEGIN
-- Cedar/Mesa definition of record passed to C proc
-- CirioNubLocalPCtoInfo. It overlays the CirioNubPCInfo
-- structure defined in CirioNubTypes.h and included by C prog
-- that implements this procedure.
SWPCInfoPtr: TYPE = POINTER TO SWPCInfo;
SWPCInfo: TYPE = MACHINE DEPENDENT RECORD [
 procName,fileName,
  guessEmbeddedFileName: UXStrings.UnixString,
 procSymID, fileSeqNum, guessEmbeddedFileSymID: CARD ];

-- Target definition of record structure containing processed
-- data returned from calling C proc CirioNubLocalPCtoInfo
PCInfo: TYPE = REF PCInfoBody;
PCInfoBody: TYPE = RECORD [
 procName: Rope.ROPE,
 procSymID, fileSeqNum, guessEmbeddedFileSymID: CARD,
 fileName, guessEmbeddedFileName: PFSNames.PATH ];

SameWorldPCtoInfo: PUBLIC PROC[pc: CARD]
 RETURNS[PCInfo] = TRUSTED
BEGIN
 -- Definition of MACHINE CODE procedure, PCtoInfoInner
 PCtoInfoInner: PROC [pc: CARD, buf: SWPCInfoPtr]
 RETURNS [INT] = TRUSTED MACHINE CODE {
   "CirioNubLocalPCtoInfo"
 }; -- of procedure PCtoInfoInner

 info: PCInfo;
 buf: SWPCInfo;
 info ← NEW[PCInfoBody];
 THROUGH [1..3] DO
   -- Invoking the C procedure CirioNubLocalPCtoInfo
   result: INT ← PCtoInfoInner[pc, @buf];
 ENDLOOP;
 -- Processing data returned by C proc CirioNubLocalPCtoInfo
 -- including converting variables of UNIX string type to ROPE
 info.procName ← UX.ToRope[buf.procName];
 info.procSymID ← buf.procSymID;
 info.fileName ←
   PFS.PathFromRope[UXStrings.ToRope[buf.fileName]];
 info.fileSeqNum ← buf.fileSeqNum;
 info.guessedEmbeddedFileName ←
   PFS.PathFromRope[
   UXStrings.ToRope[buf.guessedEmbeddedFileName]];
 info.guessedEmbeddedFileSymID ←
   buf.guessedEmbeddedFileSymID;
 RETURN[info];
END; -- of procedure SameWorldPCtoInfo
END. -- of program module CirioDeltaFaceSunImpl
```

Here is the C procedure being invoked and it's include file

C header file: CirioNubTypes.h (include file for callee)

```
/* other definitions of other structures and variables */
/* Defines structure passed to C proc CirioNubLocalPCtoInfo */
typedef struct CirioNubPCInfoRep {
 char *procName;
 unsigned procSymID;
 char *filename;
 unsigned fileSeqNum;
 char *guessedEmbeddedFileName;
 unsigned guessedEmbeddedFileSymID;
} *CirioNubPCInfo;
```

C file: CirioNubLocalProcs.c (callee)

```
/* other includes */
#include "xr/CirioNubTypes.h"
/* other processing */
/* This is the C procedure being called within Cedar/Mesa proc
/* SameWorldPCtoInfo shown on previous page */
CirioNubLocalPCtoInfo(pc, buf)
 unsigned pc;
 CirioNubPCInfo buf;
{
 /* Process information */
}
```

3.2 ⊘ Calling Cedar/Mesa procedures from within a C module

There are two ways to invoke a Cedar/Mesa procedure from a C procedure. The recommended method involves forcing the compiled Cedar/Mesa procedure to both assume a programmer defined name and become publicly accessible. A rather obscure method utilizes the InstallationComforts procedures to access Cedar/Mesa procedures via the DEFINITIONs module interface.

3.2.1 ⊘ Access Cedar/Mesa procedures via External Names

This method involves imposing programmer-defined procedure names during the process when a Cedar/Mesa program is translated into machine-dependent C code by the Mimosa compiler and making it publicly accessible. Unless specified within the language syntax construct described below, CedarMesa procedures are translated by Mimosa into static C functions (function inaccessible/invisible to other modules) with compiler-generated names.

The following steps are performed on the encompassing Cedar/Mesa module.

1. Define a "MACHINE CODE" procedure "ExternalNames" with the following syntax (**boldface** symbols are included literally).

ExternalNames: **PROC [ ] = TRUSTED MACHINE CODE** {
 "^ExternalNames\n";
 <Proc name equiv list>
};

<Proc name equiv list> : =
  <Proc name equiv> <Proc name equiv list> | empty
<Proc name equiv> : =
  "<Cedar proc name> <C proc name>**\n";**
<Cedar proc name> : = Cedar/Mesa procedure called
<C proc name> : = Functionally-equiv C proc created by the Mimosa compiler. This name is used by C programs to call procedure.

For example, the code below enables Cedar/Mesa procedures MesaProc1 and MesaProc2 to be accessible to C code via invoking Proc1, Proc2 respectively.

```
ExternalNames: PROC[ ] = TRUSTED MACHINE CODE {
 "^ExternalNames\n";
 "MesaProc1 Proc1\n";
 "MesaProc2 Proc2\n";
};
```

The only restrictions are that the Cedar/Mesa procedure must be unique and exist on the top level (not a nested procedure).

2. Invoke "ExternalNames" procedure within the mainline code of the Cedar/Mesa module. This step must be carried out in order to make the Cedar/Mesa procedure accessible.

3. Compile the Cedar/Mesa module.

Invoke a Cedar/Mesa procedure by supplying <C proc name> along with it's arguments and in certain situations (discussed below), a pointer to locations for storing the return results. Arguments are passed by value. The type of each argument value must be compatible with it's corresponding parameter type; the same rule applies for the procedure results.

Determining the exact syntax of the procedure call is complicated due to restrictions imposed by the Mimosa compiler under two conditions.

First, Cedar/Mesa procedures that expect a parameter record occupying more than 16 machine words (due to a combination of large number of parameters/data structures) are invoked by placing procedure arguments within a C structure with corresponding data types and passing the pointer address instead.

Second, means of specifying the result record varies according to whether the Cedar/Mesa procedure return results record occupying more than 1 machine word.

o If the return data occupies one machine word or less (at most 1 return variable of integer, character or pointer type), the caller C code should utilize an assignment statement. The return variable should be to the left of the assignment and the procedure name along with the arguments on the right side.

o If the return data occupies more than one machine word (more than 1 return variable, records), the caller C code should allocate sufficient room to store the return data and provide this address within the procedure call, immediatedly preceding input arguments.

Return data of type character occupies exactly one machine word (32 bits), although only the least significant byte contains the actual character. In order to access the return value, the C program can define a structure, occupying 1 word, that utilizes the "bit field" construct within the C programming language to extract the character data. This is shown by the sample program shown on the next page (refer to defined type CHAR_TYPE and the call to "record_ops" within the C program).

For real on-line examples of C accessing Cedar/Mesa procedures, reference files within the /PCedar/CFontSolution directory. It contain numerous Cedar/Mesa procedures being made externally visible in addition to the C modules that reference these functions.

The following is an example of a C procedure calling a Cedar/Mesa procedure using ExternalNames.

Cedar/Mesa program

```
CedarMesaProg: CEDAR PROGRAM =
BEGIN

ArgumentRecord: TYPE = RECORD[
 int1, int2, int3: INT
 ];

Multiply: PROCEDURE [op1, op2: INT] RETURNS [INT] =
{
 RETURN [op1*op2];
};

Change: PROC [op1, op2: REF INT] =
{
 op1^ ← op1^ + 10;
 op2^ ← op2^ + 20;
};

Record: PROCEDURE [op1, op2: INT, value: CHAR]
 RETURNS [int1,int2: INT,
             char1: CHAR, int3: INT, char2: CHAR] =
{
 RETURN [(op1*op2), (op1+op2), value+1, 89, 'L];
};

RecordPtr: PROCEDURE [argument: ArgumentRecord]
 RETURNS [answer: REF ArgumentRecord] =
{
 answer ← NEW[ArgumentRecord ←
 [argument.int1+argument.int2+argument.int3,
 argument.int1-argument.int2-argument.int3,
 argument.int1*argument.int2*argument.int3]];
};

-- Four Cedar/Mesa procedures made accessible to C
-- programs; Multiply, Change, Record & RecordPtr.
-- They are accessed as multiply_ops, change_ops,
-- record_ops and recordptr_ops respectively */
ExternalNames: PROC [] = TRUSTED MACHINE CODE {
 "^ExternalNames\n";
 "Multiply      multiply_ops\n";
 "Change        change_ops\n";
 "Record        record_ops\n";
 "RecordPtr     recordptr_ops\n";
};

-- The procedure defining the ExternalNames must be invoked
-- in order for things to take effect */
ExternalNames[];

END.
```

C Program, cprog.c

```
void XR_run_cprog( )
{
int answer;
char char_result;

/* Used to capture return results of type char , where data is
   stored in the least significant byte of a machine word  */
typedef struct {
 unsigned int dummy : 24;
 unsigned int character : 8;
} CHAR_TYPE;

/* Defines structure storing the output results from calling
    Cedar/Mesa procedure record_ops */
struct Result_Struct {
 int int1, int2;
 CHAR_TYPE char1;
 int int3;
 CHAR_TYPE char2;
} *result;

/* Defines structure holding the input arguments passed
    into Cedar/Mesa procedure recordptr_ops */
struct ArgRecord {
 int int1, int2, int3;
} argument, *resultptr, *recordptr_ops( );

/* Access Cedar/Mesa procedure multiply_ops */
op1 = 8; op2 = 11;
answer = multiply_ops(op1, op2);

/* Create storage for output results from calling record_ops */
result = (struct Result_Struct *) XR_malloc(
 sizeof(struct Result_Struct));
/* Access Cedar/Mesa procedure record_ops */
record_ops(result, op1, op2, 'A');
/* Access the returned character data, char1 */
char_result = (char) result->char1.character;
XR_free(result);

/* Access Cedar/Mesa procedure recordptr_ops */
argument.int1 = 6; argument.int2 = 8; argument.int3 = 4;
resultptr = recordptr_ops(argument);

/* Access Cedar/Mesa procedure change_ops */
change_ops(&op1, &op2);
}
```

3.2.2 ⊘ Access Cedar/Mesa procedures via DEFINITIONS interface

The second way of calling a Cedar/Mesa procedure within C is provided by a collection of procedures within /PCedar/InstallationComforts that allows a C module to access Cedar/Mesa procedures through it's exported DEFINITIONS module interface [Ref 10].

This example illustrates how access to proc MesaProc (defined in interface MesaInterface.mesa) is attained.

```
#include <xr/BasicTypes.h>
c_proc( )
{
XR_MesaProc mesa_proc, XR_ProcFromNamedInterface();
mesa_proc = XR_ProcFromNamedInterface(
  "MesaInterface", "MesaProc", 0);
}
```

Once access is attained, invoking the Cedar/Mesa procedure is done by specifying the string
(*(mesa_proc->mp_proc)) (args, mesa_proc->mp_x)

3.3 ⊕⊘ Incorporating C and Cedar/Mesa modules together

In order for C and Cedar/Mesa procedures to successfully call one another, all "cross-language external" symbols must be made accessible to the callers at run-time. There are several ways to ensure that symbols are loaded onto the Cedar environment prior to establishing access.

The best way of incorporating mixed-language modules together would be to utilize the Cedar/Mesa concept of a configuration; users unfamiliar with this construct should refer to Mesa Language Manual chapter 7 [Ref 5]. The "STATIC REQUESTS" feature was added to the C/Mesa language to enable configurations containing modules of different languages. This feature follows the "EXPORTS" clause within a C/Mesa language syntax and consists of the words "STATIC REQUESTS" followed by a list of one or more C object files to be linked into the configuration. Each C object filename should be enclosed within double quotes.

Example:
```
Foo: CONFIGURATION
IMPORTS -- list of imported Cedar/Mesa interface modules
EXPORTS -- list of exported Cedar/Mesa interface modules
-- Files foo.c and foo1.c are C module components
STATIC REQUESTS "foo.o", "foo1.o"
CONTROL -- list of Cedar/Mesa implementation modules =
BEGIN
 -- list of Cedar/Mesa program modules or configurations
END.
```

Configurations are the preferred means of building large multi-module software systems because they present a simple, easily reproducible description and the ability to nest configurations together encourages development along functional lines.

Two other methods of incorporating these modules exist, although their use is not recommended. First, the modules can be compiled separately and independently run. The one caveat is that modules containing called procedures must be run before any of the calling modules. This requires the modules to be executed in an ordered fashion. Second, modules may be compiled separately, however the caller program initiates dynamic loading of necessary support modules via PCR calls prior to making the call.

Inaccessible "cross-language" symbols are not detected until run-time when these external references are resolved. During execution of the "Run" command from the Cedar Commander viewer, either an error message is generated expressing the inability to resolve an undefined UNIX-level symbol or the program crashes and immediatedly brings up the Cirio debugger.

3.4 ⊕⊘ Delivering software as a package

Many application software projects developed within the Cedar environment are delivered to customers who are unfamiliar with and have no need for a user-interactive Cedar environment running atop the SUN workstation. In these situations, the software should be developed and delivered as a "package". It consists of a single UNIX object file (a.out) which contains all the necessary code needed to run the application within a foreign host environment.

The following software elements are found within a packaged world:

1. Cedar Basement
   - o PCR : Sits atop the SUN OS and provides runtime support for garbage collection, dynamic loading and lightweight processes (threads)
   - o CedarCore: Supports the Cedar programming language
   - o BasicCedar: Provides useful facilities for Cedar programs
2. Miscellaneous Runtime code
3. Application software

Here are the steps needed to create an optimized package named FooPackage using the INSTALLED version of the PCR software. Assume that the user application is composed of Cedar/Mesa program modules FooImpl1.mesa, FooImpl2.mesa and C module foo_c.c. (**Boldface** symbols are included literally)

1. Retrieve [XRHome]<INSTALLED>LIB>OptThreads-sparc>

DebugNub.o	SystemDaemon.o
Interp.o	XRRoot.o
LibrarySearchingLoad.o	xr.a
symfind.o	libxrc.a

2. Create FooWorld.config (defines package components)

```
FooWorld: CONFIGURATION
EXPORTS ALL
STATIC REQUESTS     "XRRoot.o", "DebugNub.o",
                    "SystemDaemon.o", "symfind.o",
                    "LibrarySearchingLoad.o", "Interp.o"
CONTROL CedarCore, BasicCedar,
          <Misc Cedar Runtime configs/modules>,
          FooImpl1, FooImpl2, FooIWorldImpl =
BEGIN
CedarCore;
BasicCedar;
<Misc Cedar Runtime configs/modules>;
FooImpl1;
FooImpl2;
FooWorldImpl;
END.
```

<Misc Cedar Runtime configs/modules> represents all the external Cedar software packages which provide a service that is ultimately referenced by the user's applications code.

Restrictions on configuration file specifications:

- o Within the CONTROL section: packages are listed in the order in which they are to be started. CedarCore and BasicCedar must precede all others. User application modules are usually placed near the end of the list with FooWorldImpl (see step 3) being last.
- o Within the configuration body:
  - CedarCore must be listed first. The order of the other modules doesn't matter.
  - All module names appearing in the CONTROL section must appear here and vice versa.
  - No duplicate names allowed

3. Create FooWorldImpl.mesa (defines PCR operating params)

```
FooWorldImpl: CEDAR PROGRAM =
BEGIN
   DefineDefaultArgs: PROC = TRUSTED MACHINE CODE {
      "+char defaultArgs[] = \"-msgs 0 -slaveiop 1 -mem
      550000 -stack 90000 \\\n";
      "-tmpdir . -nodbxscript -- -h 4000000 -
      install_and_run_package --\";\n.";
   };
   DefineDefaultArgs[];
END.
```

The PCR operating parameters can be adjusted by changing the string item shown within DefineDefaultArgs procedure; the list of possible switch/value pairs are defined in [Ref 13].

4. Create sun4-o3/FooPackage.MakeIt (drives package creation)

```
-- sun4-o3/FooPackage.MakeIt
-- { sun4-o3/FooWorld.c2c.o }
ComplexRsh -cmd "/bin/cc -Qpath
/project/pcedar2.0/lang/ld/sparc/ -Bstatic -o sun4-
o3/FooPackage sun4-o3/FooWorld.c2c.o
/pseudo/xrhome/INSTALLED/LIB/OptThreads-sparc/xr.a
/pseudo/xrhome/INSTALLED/LIB/OptThreads-sparc/libxrc.a -
lm"
```

The second line of the file, --{ sun4-o3/FooWorld.c2c.o }, indicates that the package is defined by the FooWorld configuration file and that the object file must be built before the package can be generated.

5. Create FooWorld.mob.switches (switch for package creation)

```
-hm ~g
```

6. Initiate creation of package by entering this command:
   MakeDo sun4-o3/FooPackage

For additional assistance on building a packaged world, refer to "BuildingAPackagedWorld.tioga" (Ref [[12]) and examples for viewer packages in [PCedar2.0]<PackagedWorlds>

3.5 ⊕⊘ Remote Procedure Calls

Remote Procedure Calls (RPC) provide a means of communication between two UNIX processes. RPC implements a logical client to server communications system designed specifically for the support of network applications. The model supported includes a client that initiates a request for service by the server. Depending upon whether the RPC implementation is synchronous or asynchronous, the client may wait (block) for notification on completion of the service. This communication occurs between processes on the same or different machines. As Figure 3.1 and the Wasabi example (discussed in section 3.7) illustrate, remote procedure calls can be used between applications differing in implementation language and run-time software platform.

There are several reasons why Remote Procedure Calls can be used when creating a large software system. First, since it provides communications across a network, the software system can be distributed across more than one machine. Second, it enables the system's functional components to become more autonomous; the engineering process becomes simplified because separate components can be treated more independently.

The PPO system (Production PDL Option) provides a good example. The Postscript decomposer,WASABI, is written in Cedar/Mesa and running in the Cedar environment. A second component, EGRET is written in C and running on the standard UNIX environment. Developed separately by two different organizations, the two components interact through the use of remote procedure calls. Otherwise, the EGRET software would have to be ported into the Cedar environment. This loose coupling of two "stand-alone" components by RPCs provides the required system functionality while eliminating the need to port EGRET's foreign C code into the Cedar environment.

The RPC mechanism adds additional layers of communication protocol to the interaction between server and client processes. The overhead incurred for RPC calls generated and serviced locally (no network access required) by the same SPARCstation 2 running under SunOS 4.1.1 is a minimum of 2.2 msec. In addition, numerous potential problems are introduced by using RPCs including inability to locate remote server, unsupported remote program/procedure/transport protocol, server/client crashes and unreliable data transport.

RPC-BASED SOFTWARE ARCHITECTURE

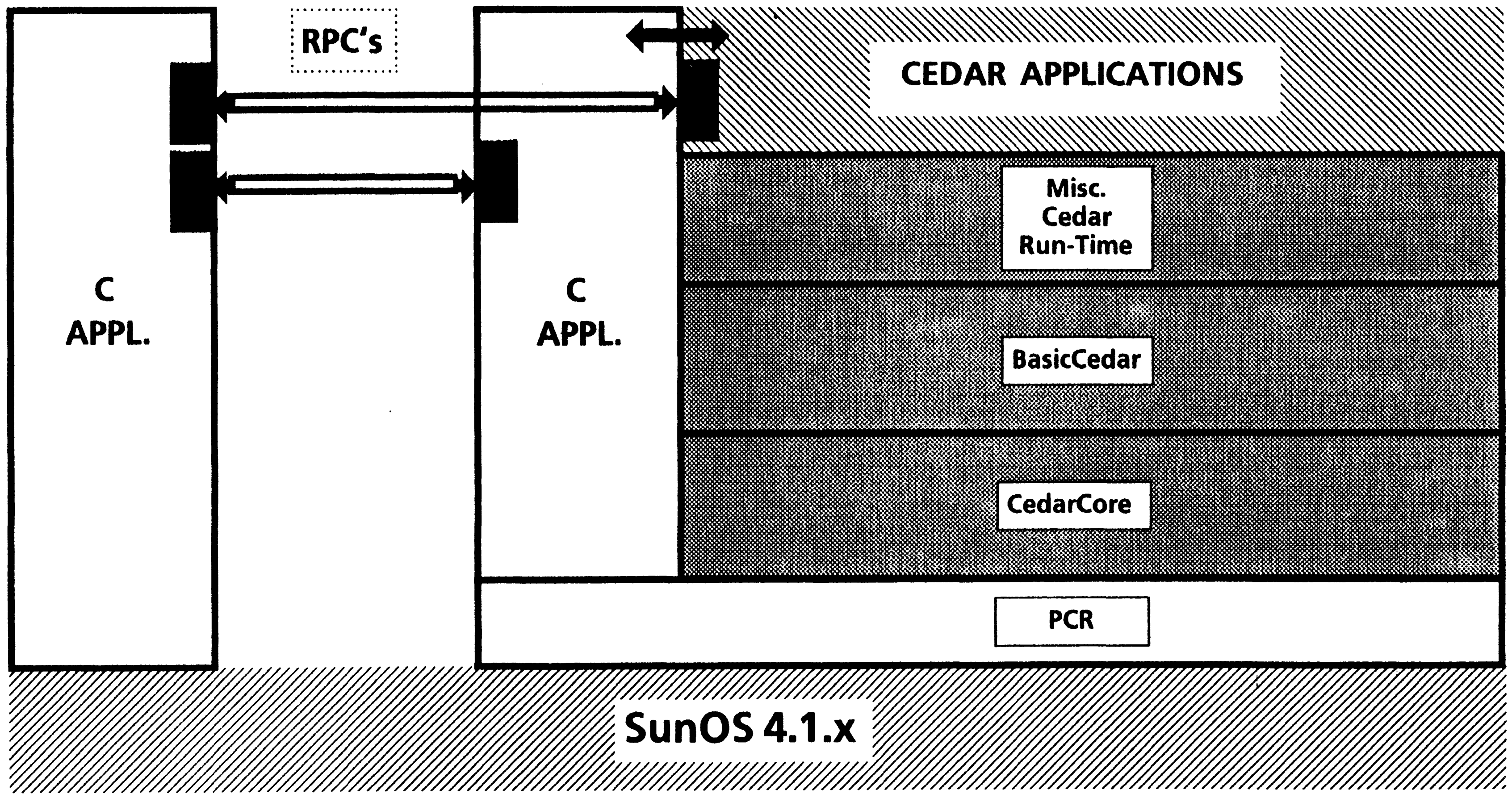


Figure 3.1

There are UNIX C and Cedar facilities, namely rpcgen and CedarRPCGen, which simplify the process of programming applications to use the Remote Procedure Calls; applications generated using CedarRPCGen are implemented in the Cedar/Mesa language while rpcgen generates C based code. Both facilities are essentially compilers which accept a remote program interface written in the RPC Language and produce the following output files: header/interface module, client and server stub programs containing the RPC system calls needed for the data transfer process and a program to perform data conversion to a format required for transport over the network. Combining the header module, client stub and a user-created client program (calls remote procedure) will create the client program; likewise, combining the header module, server stub and a user-created server program (implements remote procedure functionality) will create the server program.

Figures 3.2, 3.3 on pages 35,36 illustrate the process of creating RPC based applications for a mixed language system consisting of a Cedar-based server and a C-based client. The former is created using the CedarRPCGen utility provided by the Cedar10.1 environment while the latter is using the rpcgen UNIX facility. No special adaptation is required for developing a RPC between applications differing in implementation language and software platform; it involves using the same RPC language-based protocol specification, independently applying the CedarRPCGen/rpcgen facilities with it's associated methodology to generate each side of the RPC connection and then combining.

For more details on using rpcgen and CedarRPCGen, consult the referenced documents (Ref 14-17) and the example discussed on the next few pages.

For those individuals who are interested in developing transport independent RPC-based client/server applications: SunSoft, Inc has made a product announcement for the ONC RPC Application ToolKit 1.0. Running atop SunOS 4.1.x and Solaris 1.0 SPARCsystems, this advanced development platform enables C software developers to create client-server applications that run unmodified across a range of operating systems, hardware platforms and networks.

3.5.1 ⊘ Example: Creating an RPC based application

For more details on rpcgen, RPC protocol specification language, CedarRPCGen and other real-life examples, consult Ref 14-17

Assume that our application consists of two remote procedures
printmessage: remote display of inputted text string
addition: adds input operands (integer and numeric string)

In order to use the rpcgen or CedarRPC facilities, it is necessary to create a remote program interface definition (files should have a .x extension) using the RPC Language (see Ref 14 for complete syntax description). The protocol specification for this particular example is shown below:

```
/* msg.x: RPC protocol specification */

struct ops {
  int num;
  string s<>;
};

typedef struct ops OPERANDS;

program MESSAGEPROG {
  version MESSAGEVERS {
    int PRINTMESSAGE (string) = 1;
    int ADDITION (OPERANDS) = 2;
  } = 1;
} = 0x20000099;
```

Notice that one version (MESSAGEVERS, value 1) of remote program "MESSAGEPROG" has been declared and assigned program number 0x20000099. It contains the procedures "printmessage" and "addition".

Applications based upon C:

In order to utilize the rpcgen utility, C-based client and server modules to invoke and implement the remote procedures respectively must be written.

Here is the server module (msg_server.c) which implements the example remote procedures:

```
#include <stdio.h>
#include <rpc/rpc.h>   /* always needed */
#include "msg.h"       /* generated by rpcgen */

int *printmessage_1(msg)
 char **msg;
{
  static int result;
  FILE *f;

  f = fopen("/dev/console", "w");
  if (f == NULL) {
    result = 0;
    return (&result);
  }
  fprintf(f, "PrintMessage proc: %s\n", *msg);
  fclose(f);
  result = 1;
  return (&result);
} /* printmessage_1 */

int *addition_1(operand)
 OPERANDS *operand;
{
  static int result;
  FILE *f;

  result = atoi(operand->s) + operand->num;

  if ((f=fopen("/dev/console", "w")) != NULL) {
    fprintf(f, "Addition proc: %d\n", result);
    fclose(f);
  }

  return (&result);
} /* addition_1 */
```

Comparing the remote procedure declarations between the C server and the RPC protocol specification, three items should be noted in the former:

1. procedures take pointers to their arguments, rather than arguments themselves. Notice that UNIX type "char *" is equivalent to RPC language type "string"
2. procedures return pointers to their results
3. procedure names formed by converting the protocol specification name to lower-case letters, appending the underscore character and finally the version number.

Applications based upon C:

Here is the client module (msg_client.c) which invokes the remote procedures:

```
#include <stdio.h>
#include <rpc/rpc.h>   /* always needed */
#include "msg.h"       /* generated by rpcgen */

main(argc, argv)
 int argc;
 char *argv[];
{
  CLIENT *cl;
  int *result;
  char *server;
  char *message;
  OPERANDS addition_ops;

  if (argc != 3) {
    fprintf(stderr, "usage: %s host message\n", argv[0]);
    exit(1);
  }

  server = argv[1];
  message = argv[2];

  /* Client is utilizing the TCP transport mechanism */
  cl = clnt_create(server, MESSAGEPROG,
                          MESSAGEVERS, "tcp");
  if (cl == NULL) {
    clnt_pcreateerror(server);
    exit(1);
  }

  /* Invoke remote printmessage procedure */
  result = printmessage_1(&message, cl);
  if (result == NULL) {
    clnt_perror(cl, server);
    exit(1);
  }
  if (*result == 0) {
    fprintf(stderr, "%s: %s couldn't print your message\n",
            argv[0], server);
    exit(1);
  }
  printf("Message delivered to %s!\n", server);

  /* Invoke remote Addition procedure */
  addition_ops.num = 5;
  addition_ops.s = "45";
  printf("Addition arguments: %d %s\n",
          addition_ops.num, addition_ops.s);
  result = addition_1(&addition_ops, cl);
  if (result == NULL) {
    clnt_perror(cl, server);
    exit(1);
  }
  printf("Addition results: %d\n", *result);

  exit(0);
} /* main */
```

Applications based upon C:

Several points should be noted regarding the client module:

1. A client handle, "cl", is created using the RPC library clnt_create(). This handle is then passed as the last argument to the skeleton routines associated with the remote procedures (routine names are identical to those specified in the server module. The difference is that the former is created by rpcgen, directly invoked by the client and expects the client handle as the last argument).
2. The transport mechanism upon which the Sun RPC is based is specified by the last argument to clnt_create(). Two different mechanisms are supported, UDP/IP and TCP/IP. In order for the RPC connection to be successful, the client must specify a mechanism supported by the server.
3. Failures to the RPC mechanism cause the remote procedure to return with the NULL value.

Creating the C-based RPC application consists of performing these steps (shown in boldface type) within the unix shell.

1. **% rpcgen msg.x**

   Runs the rpcgen facility and automatically generates the following files needed to implement RPCs:

msg.h	(include in both client & server modules)
msg_svc.c	(server stub program)
msg_clnt.c	(client stub program)
msg_xdr.c	(conversion of data types between machine/network format)

2. **% cc msg_server.c msg_svc.c msg_xdr.c -o msg_server**

   Creates server program, msg_server, that implements the printmessage and addition RPC procedures for UDP & TCP transports (by default, rpcgen generates server code for both transport mechanisms).

3. **% msg_server**

   Executes the server program.

4. **% cc msg_client.c msg_clnt.c msg_xdr.c -o msg_client**

   Creates client program, msg_client, which uses the TCP/IP transport mechanism to invoke the printmessage and addition Remote Procedure Calls.

5. **% msg_client** <your machine name> <your message>

   Executes the client program, either on another SUN workstation (the executable msg_client must be copied onto the remote machine first) or a separate unix shell window.

Applications based upon Cedar/Mesa:

Utilizing the same RPC protocol specification (msg.x), the next example demonstrates the steps in creating a Cedar/Mesa based RPC application. The easiest method is by using the CedarRPCGen utility; the user must create the Cedar/Mesa client and server module to invoke and implement remote procedures printmessage and addition.

Here is the server module (msgServerMain.mesa) for the RPC application defined by msg.x.

```
   DIRECTORY
     Commander, Convert, IO, msg, Rope,
     SunRPC,SunRPCBinding;

   msgServerMain: CEDAR PROGRAM
   IMPORTS Commander, Convert, IO, msg, SunRPCBinding =
   BEGIN OPEN msginterface: msg;
     out: IO.STREAM;

     PRINTMESSAGE: msginterface.printmessageProc = {
       IO.PutF1[out, "PrintMessage: %g\n", IO.rope[in]];
       res _ 4};

     ADDITION: msginterface.additionProc = {
       res _ in.num + Convert.IntFromRope[in.s]};

     msgServer: Commander.CommandProc = {
       p1: SunRPC.Server _
         msginterface.MakeMESSAGEPROG1Server[
           NIL,PRINTMESSAGE, ADDITION];
       p2: SunRPC.Server _
         msginterface.MakeMESSAGEPROG1Server[
           NIL,PRINTMESSAGE, ADDITION];
       out _ cmd.out;
       IO.PutRope[out, "Start up server\n"];
       [] _ SunRPCBinding.Export[unboundServer: p1,
             transport: TCP, reExport: TRUE];
       [] _ SunRPCBinding.Export[unboundServer: p2,
             transport: UDP, reExport: FALSE]};

     usage: Rope.ROPE= "implements msg.x RPCs";
     Commander.Register["msgServer", msgServer, usage];
 END.
```

Several items regarding the server module should noted:

1. DEFINITION module msg.mesa should be imported. This file is automatically generated by CedarRPCGen and defines types and procedures needed by both server and client modules.
2. Implementation procedures for the RPCs printmessage and addition must be explicitly specified using proc MakeMESSAGEPROG1Server defined in msg.mesa.
3. The transport mechanism, TCP/IP or UDP/IP, supported by the server can be explicitly specified as an argument to the SunRPCBinding.Export proc. If left unspecified, the latter is assumed.

Applications based upon Cedar/Mesa:

Here is the client module (msgClientMain.mesa) for the RPC application defined by msg.x.

```
DIRECTORY
  Commander, IO, msg, Rope, SunRPC,
  SunRPCAuth, SunRPCBinding, ThisMachine;

msgClientMain: CEDAR PROGRAM
IMPORTS Commander, IO, msg, SunRPC, SunRPCAuth,
          SunRPCBinding, ThisMachine =
BEGIN OPEN msginterface: msg;

  msgClient: Commander.CommandProc = {
    thisMachineName: Rope.ROPE _ ThisMachine.Name[];
    h: SunRPC.Handle _ SunRPCBinding.Import[
      hostName: thisMachineName,
      pgm: msginterface.MESSAGEPROG,
      version: msginterface.MESSAGEVERS];
    h1: msginterface.MESSAGEPROG1;
    success, addresult: INT32;
    addops: msginterface.OPERANDS = [num: 45, s: "35" ];

    -- Call message proc
    h1 _ msginterface.MakeMESSAGEPROG1Client[h,
      SunRPCAuth.Initiate[]];
    success _ h1.printmessage[h1, "Hi server\n"];
    IF (success > 0) THEN IO.PutRope[cmd.out, "Sent OK\n"]
    ELSE IO.PutRope[cmd.out, "Errors sending message\n"];

    -- Call addition proc
    addresult _ h1.addition[h1, addops];
    IO.PutF[cmd.out, "Addition results: %g + %g = %g\n",
      IO.int[addops.num], IO.rope[addops.s],
      IO.int[addresult]];

    SunRPC.Destroy[h] };

  -- mainline code
  usage: Rope.ROPE = "Client for msg RPC.";
  Commander.Register["msgClient", msgClient, usage];
END.
```

Several items regarding the client module should be noted:

1. DEFINITION module msg.mesa should be imported. This file is automatically generated by CedarRPCGen and defines types and procedures needed by both server and client modules.
2. The client module identifies the remote program by invoking SunRPCBinding.Import procedure with arguments specifying the server machine name, program and version number. In addition, the client can also specify the transport mechanism; UPD/IP is assumed unless otherwise specified.
3. In order to access the remote procedures, the client must invoke proc MakeMESSAGEPROG1Client defined in msg.mesa and save the returned access pointer. This variable is used to invoke the remote procedures; notice the access pointer is the first argument in the proc call.

Applications based upon Cedar/Mesa:

Create the Cedar/Mesa based RPC application by performing the steps shown in boldface type.

1. cowpoke% **cedarrpcgen msg.x**

   The Cedar10.1 version of the cedarrpcgen program is run from the UNIX command shell. It automatically generates the following Cedar/Mesa files :

msg.mesa	(imported by both client & server modules)
msgServerImpl.mesa	(server stub program)
msgClientImpl.mesa	(client stub program)
msgGetPut.mesa	(interface for data type conversion)
msgGetPutImpl.mesa	(implements type conversion)

   Two file modifications must be made in order for the build process to be successful:
   a. msgServerImpl.mesa = remove reference to the ROPE interface within the IMPORT list.
   b. msgGetPutImpl.mesa = remove reference to the msg interface within the EXPORT list.

2. Create msgServer.config to define the server program

```
msgServer: CONFIGURATION
  IMPORTS Commander,Convert,IO,
             SunRPC, SunRPCBinding
  CONTROL msgServerMain, msgServerImpl =
BEGIN
  msgServerMain;
  msgServerImpl;
  msgGetPutImpl;
END.
```

3. % **MakeDo sun4/msgServer.c2c.o**

   Build the msgServer executable by typing the above command into a Cedar viewer.

4. % **Require Cedar SunRPCBinding SunRPCBinding**
   % **Run msgServer**

   Execute the msgServer program by typing the above commands into a Cedar viewer.

5. Create msgClient.config to define the client program

```
msgClient: CONFIGURATION
  IMPORTS Commander,IO,SunRPC,
             SunRPCAuth, SunRPCBinding,ThisMachine
  CONTROL msgClientMain =
BEGIN
  msgClientMain;
  msgClientImpl;
  msgGetPutImpl;
END.
```

6. % **MakeDo sun4/msgClient.c2c.o**

   Build the msgClient executable by typing the above command into a Cedar viewer.

7. % **Require Cedar SunRPCBinding SunRPCBinding**
   % **Run msgClient**

   Execute the msgClient program by typing the above commands into a different Cedar viewer.

Mixed language RPC applications:

Remote procedure calls provide a means for Cedar/Mesa and C based applications to communicate and interact. As a result of compatibility between applications generated by using the rpcgen and CedarRPCGen utility, creating such a mixed language RPC application is quite simple. It involves following these steps.

a. Use the CedarRPCGen utility as discussed previously to generate the Cedar/Mesa based server and client programs.
b. Use the rpcgen utility as discussed previously to generate the C based server and client programs.
c. Use appropriate server and client programs. The only caveat is that transport mechanism expected by the client must be supported by the server.

An example of a mixed language (Cedar/Mesa and C) application can be provided using server and client components built earlier for the msg.x RPC protocol specification. In this case, we are assuming the presence of a interactive Cedar environment upon which the Cedar/Mesa component resides.

An illustration of the steps involved in creating a RPC application consisting of a Cedar/Mesa server and a C client is provided by figures 3.2 and 3.3 .

However, a variation upon this example demonstrates that the Cedar /Mesa component can be packaged (section 3.4) into a single executable containing the application, Cedar support code and support software for Remote Procedure Calls. This example consists of a Cedar/Mesa server and a C client; no changes were made to the C client and only minor edits were made to ServerMain.mesa so that RPC service for procedures printmessage and addition are provided when the packaged world executable is run. The application code can be retrieved and built by typing the following text string to a UNIX command shell:

% **/home/rayli/RPC/get_RPC_example.csh**

It will take approximately 4 minutes for the C client, Cedar/Mesa server files to be retrieved onto the current directory, compiled and linked. Upon completion, the server and client programs can be invoked by typing the following into a UNIX command shell:

% **sun4/msg2ServerWorldPackage**
Runs Cedar/Mesa server

% **msg_client <server machine> <string>**
Run C client on another machine or another shell tool

CREATING RPC APPLICATION: CEDAR/MESA SERVER

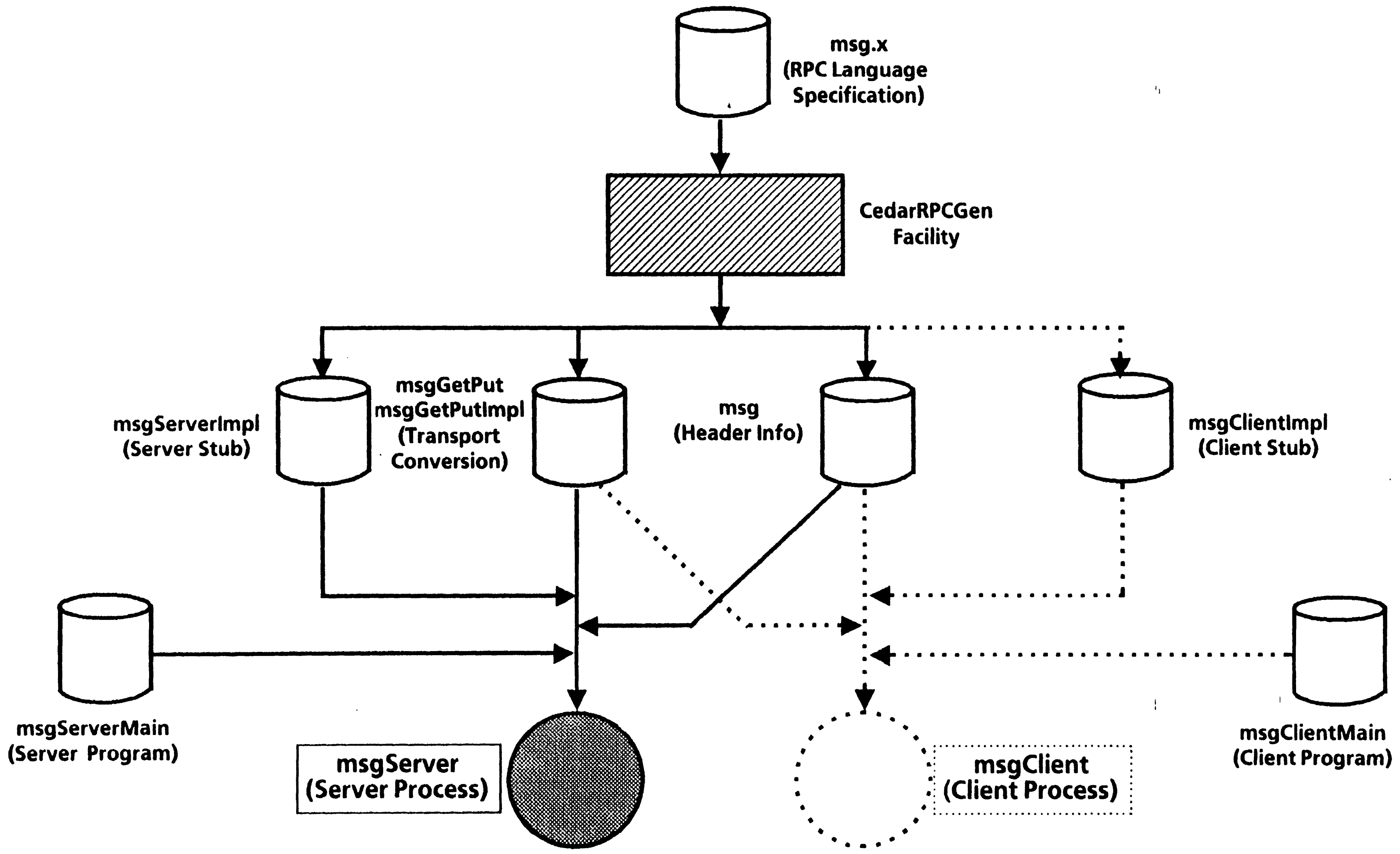


Figure 3.2

CREATING RPC APPLICATION: C CLIENT

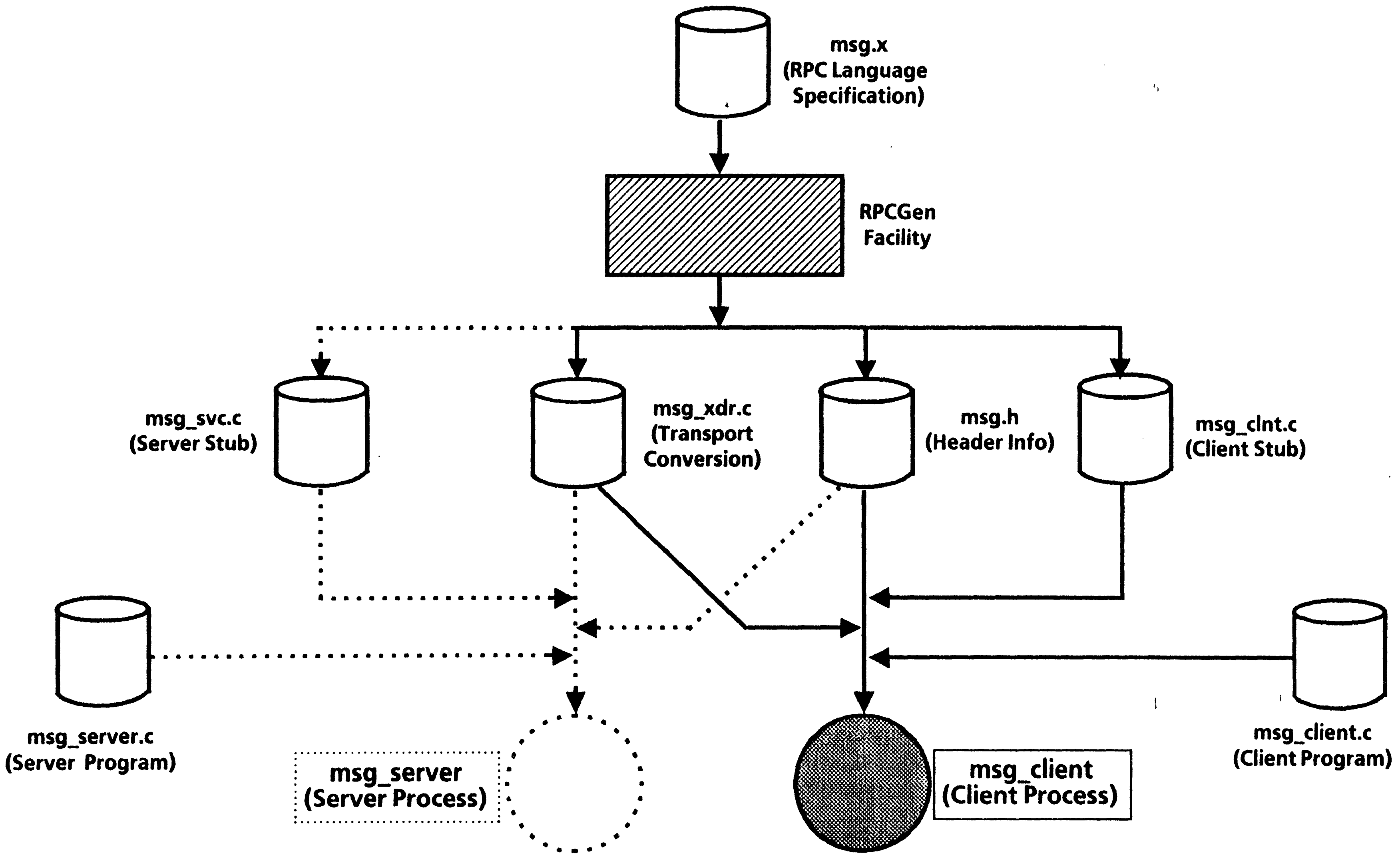


Figure 3.3

3.6 ⊘ Debugging PCR-based Applications

The material in this section is taken from information gathered by John Wenn. It is also based upon SOLARIS 1.0 and does not apply to SOLARIS 2.0 and SunOS 5.0.

Debugging PCR-based applications is supported by the debugger stub program, DebugNub.o, which is automatically loaded into PCR upon start-up and runs as a slave IOP process. DebugNub provides the interface for debugger/debuggee interactions via a reserved UNIX socket. Socket number 4815 is reserved for the first instance of PCR on a workstation with additional instances being assigned socket 4816, 4817, etc; the identity of this reserved socket can be acquired by running "ShowCirioPort" from the Cedar Commander.

There are currently three methods available for debugging PCR-based applications: RemoteDebugTool/dbx, Cirio and a combination of the two. Each will be discussed in the following sections.

RemoteDebugTool / dbx

The combination of RemoteDebugTool (RDT) and dbx can be used to debug modules written in the C programming language. RemoteDebugTool is a UNIX program developed at PARC CSL that attaches to the DebugNub stub and supports the display and manipulation of threads/memory addresses within an executing PCR world. It runs from a UNIX shell on either the same or different SUN workstation running PCR; the debuggee's workstation name and reserved DebugNub socket should be supplied on the command line as follows:

```
   % RemoteDebugTool <machine name>  <socket #>
```

This message below will appear :

```
   Connecting to cowpoke (port 4815) ... Debuggee protocol
   version 7.
   (rdt):
```

A description of all RDT commands is available by typing the "?" character.

Used in conjunction with RemoteDebugTool is the standard UNIX dbx/dbxtool utility to perform source-level debugging. This combination enables the PCR users to debug C-based program modules that were explicitly written in the C language or those translated from Cedar/Mesa by the Mimosa compiler. To invoke dbx on the PCR environment, the C shell script XRDBX must be executed. This script prepares the PCR world for UNIX-level debugging by transforming symbol table information automatically generated by PCR during dynamic loading into an a.out format before invoking an instance of the dbx interpreter. Source level debugging can now occur using the facilities provided by dbx; a description of all dbx commands be acquired by either typing "help" at the dbx prompt (dbx) or by referencing SUN's "Debugging Tools Manual" [Ref 18].

This typescript demonstrates the steps required to debug a PCR-based application using RemoteDebugTool/dbx on SUN cowpoke. The debuggee (Reverse.c2c.o) contains a C-based module reverse_c.c upon which dbx will be utilized. This example uses three different tty windows for the PCR Commander, RemoteDebugTool and dbx [distinguished by the prompts %, (rdt) and (dbx) respectively]. User-issued commands are shown in boldface, system responses in italics and explanations in the smaller font size.

a. In window #1, start PCR world with application loaded

```
cowpoke% X11ViewersWorld
% cd /cowpoke/rayli/test
% run Reverse.c2c.o
% ShowCirioPort
4815
```

Use this socket number as an argument in invoking RemoteDebugTool

```
% ls /tmp/ILsymtab.pid*
ILsymtab.pid927
```

The value 927 represents the process ID of the UNIX process underlying PCR and will be used for initiating the XRDBX script. PCR continually maintains a symbol table file, ILsymtab.pid*, that reflects the name and relocation value of each dynamically loaded module. Should debugging using dbx be desired, a simulated object file representing a statically linked version of the current PCR state is built on the fly.

b. In window #2, start up RemoteDebugTool

```
cowpoke% RemoteDebugTool cowpoke 4815
Connecting to cowpoke (port 4815) ... Debuggee
protocol version 7.
(rdt): uni
(rdt):
```

c. In window #3, prepare to invoke XRDBX by creating a text file XRDBXInner containing the process ID of the UNIX process underlying PCR and the symbol table file XRDBXInner. Once the XRDBX completes, the PCR world remains frozen until the dbx command dbx is issued.

```
cowpoke% cat > XRDBXInner
set core = 927
set model = /tmp/ILsymtab.pid927
cowpoke% XRDBX
Debugging with symbols from file /tmp/ILsymtab.pid927,
source in /pseudo/xrhome/INSTALLED/SRC, and state in
927.
Transforming symtab file ...done.
Reading symbolic information...
Read 43177 symbols
(dbx)
```

d. Perform debugging (eg, setting breakpoints) using dbx on reverse_c.c

```
(dbx) ignore 30 2 11 10
(dbx) file reverse_c.c
(dbx) stop in proc1
(dbx) stop at 250
(dbx) cont
```

e. Run PCR-based application program

f. To quit debug session while keeping the PCR world alive

```
(dbx) detach
(dbx) quit
(rdt) disconnect
(rdt) quit
```

Cirio

Cirio is a software tool created by PARC CSL that runs in PCR and provides debugging support for PCR-based application programs. Cirio currently handles programs written in Cedar/Mesa and a limited subset of the C programming language; users interested in a detailed description of the C language constructs understood by Cirio should contact PARC or their local Cedar Support personnel. The ultimate goal of Cirio is to provide support for an open-ended spectrum of programming languages.

Cirio has two versions. The same-world version runs in the same PCR world as the debuggee and can be created in response to an unforseen error or upon encountering a previously set breakpoint. The remote version provides debug access to a different PCR world which may either on the same or different SUN workstation. In order for a remote PCR world to be debugged, the remote version of Cirio must attach to the DebugNub stub running as a slave IOP within the PCR world. In order to invoke the remote Cirio, the debuggee's machine name and DebugNub port number should be supplied to the Commander viewer:

```
        % CirioRemote  machineName portNumber
```

A detailed description of the capabilities of Cirio is beyond the scope of this document; for more information concerning Cirio debugging, users should consult [Ref 19].

combo of Cirio, RemoteDebugTool/dbx

PCR applications may be debugged using a combination of Cirio and RemoteDebugTool/dbx. Because the DebugNub program provides only one interface port, only one debugger can be active at any time. Using a particular debugger involves detaching the other debugger from the client and attaching itself. Any breakpoints set during the debug session are ignored and the application program runs to completion when the debugger is detached.

The table shown on the following two pages illustrates a comparison of the features offered by the Cirio and RemoteDebugTool/dbx debuggers.

COMPARING RDT/DBX AND CIRIO DEBUGGERS

Debug Feature		RDT/dbx Command	Cirio Command	Cirio Status
General	Specific			
Call Stack	List call stack	where	Summary	Release
	Up/Down call stack	up / down	WalkStack	Release
Variables	Print expression	print	<expr>	Release*
	Print expression always	display	---	---
	Type of expression	whatis	<expr> ?	Release*
	Set variable	assign / set	<var> = <exp>	Release+
	Local variables	dump	ShowFrame	Release+
Breakpoints	Set breakpoint	stop	SetBreak	Release*
	Clear breakpoint	delete / clear	ClearBreak	Release*
	Execute commands at break [1]	when	---	---
	Status of breakpoints	status	ListBreaks	Release
Connect to prog	Start program from debugger	---	---	---
	Debug from fatal error	---	uncaught signal	Release
	Debug running local program	---	set break	Release
	Debug running remote prog	XRDBX	CirioConnectToWorld	Release
Run / Trace	Run program	---	---	---
	Continue execution	cont	Proceed	Release
	Trace execution[1]	trace	---	---
	Stop at next line (into procs)	step	---	Planned
	Stop at next line (ignore procs)		---	Planned
	Call procedure	---	<Proc[a]>	Release@
	Kill current thread	kill	Kill	Release
	Abort current thread	abort	Abort	Release
Files /Directories	Add dir to search path	use	AddDir	Release
	List search path	use	ListDir	Release
	Remove dir from search path	---	RemoveDir	Release
	Change current dir	cd	---	---
	Show source position	<always>[2]	ShowSourcePosition	Release+

COMPARING RDT/DBX AND CIRIO DEBUGGERS

Debug Feature		RDT/dbx Command	Cirio Command	Cirio Status
General	Specific			
Threads	Focus on thread	examine	Detailed	Release
	Set current thread	examine	SetCurrentThread	Release
	Print thread info	display	PrintCurrentThread	Release
	Add thread to list	---	AddThread	Release
	List threads	display	ListAvailableThreads	Release
	Breakpoint stop all threads	<always>	ToggleBreakStopAll	Release
	Freeze thread	freeze	Freeze	Release
Remote Debug	Start remote debug	connect	CirioConnectToWorld	Release
	Stop remote debug	disconnect	CirioDisconnect	Release
	Kill remote client	quit	KillWorld	Release
	Stop remote client	stop	StopRemoteWorld	Release
	Resume remote client	cont	ResumeRemoteWorld	Release
	Resume (1 virtual processor)	uniprocessor	ResumeVP0	Release
Misc commands	Help	help	CirioHelp	Release
	Stop tool	ctrl-C	Stop	Release
	Create menu	menu[2]	---	---
	Add new button	button[2]	---	---
	Alias dbx command	alias	---	---
	Execute file with dbx cmds	source	---	---
	Set language	---	Language	Development
	Automatically set language	<done>	---	Planned
	Keep window state	<done>	---	Planned

Release* = This works in the release version for Cedar code. It works in the Development version for C code

Release+ = Release version works for Cedar code. It works (with restrictions) in Development version for C code. Cirio currently has problems with complex C expressions that include arrays and pointers.

Release@ = This works for Cedar code while local debugging. It does not work for C code in local debugging, and does not work for anything while remote debugging.

Footnote 1 = Dbx has many options on when a breakpoint is executed. They include: at a line number, only if a condition is true, if a variable changes, and at the start of a procedure. The when and trace command have similiar options.

Footnote 2 = This command only works in dbxtool.

3.7 ⊕⊘ Case Study: Integrating EGRET* with WASABI

An example of a mixed language system within the Cedar environment is Wasabi 1.0 (see Figure 3.4). A component of the Production PDL Option system (PPO 1.0), Wasabi interprets and decomposes Postscript PDL level 1 data to the selected output device pixel level. This page information is then passed back to EGRET for subsequent output on a printer or display device.

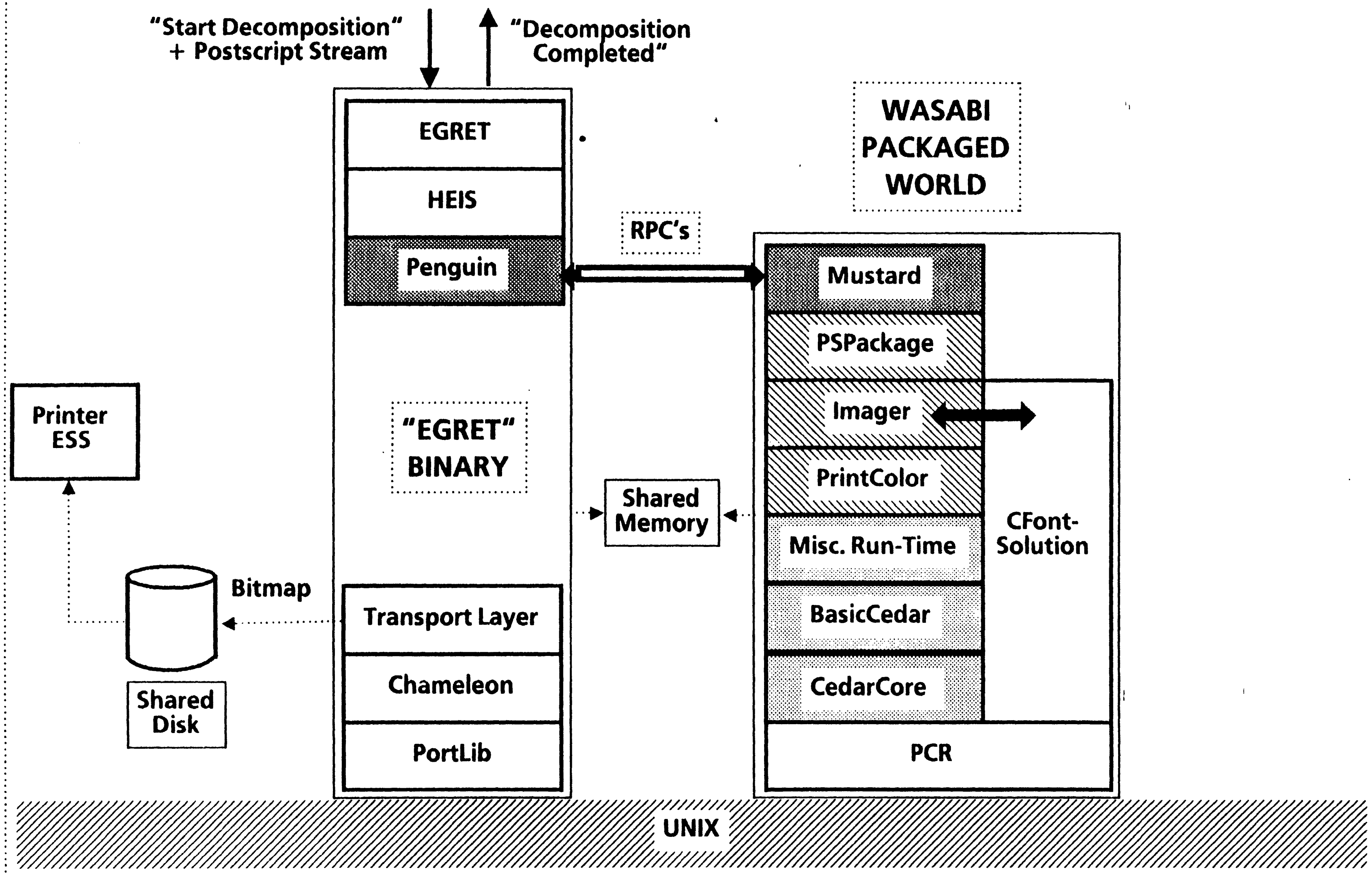
Functional components of Wasabi:

Mustard: Provides an RPC interface to EGRET
PSPackage: Postscript Interpreter
Imager: Common Cedar Imager
CFontSolution: Common Font Rasterizer
PrintColor: Handles gray shades, half-toning for black/white

Wasabi is implemented by a combination of Cedar/Mesa and C code with the majority being Cedar/Mesa based (> 90%). CFontSolution is the only component that contains C code; it utilizes font rasterization services provided by the C-based Font Solution software originally developed by WRC on a standard UNIX platform. This software was ported to the Cedar environment and combined with Cedar/Mesa modules providing the rasterizer with necessary character/font information such as character metrics. This exchange is implemented using the "EXTERNAL NAMES" technique (section 3.2.1) to enable the C-based software to access Cedar/Mesa procedures. All CFont Solution and Cedar/Mesa modules are incorporated into a Cedar/Mesa configuration module (section 3.3) named CFontSolution.

Wasabi is packaged (section 3.4) into a single binary executable containing the Cedar Basement, Miscellaneous Runtime and the application-specific Postscript Decomposer code. This package contains all the facilities required to run Wasabi on SUNOS 4.1.x. Although it occupies a single executable/address space, Wasabi spawns off several UNIX processes (section 2.2) related to the Portable Common Runtime implementation within the Cedar Environment. The exact number of processes is provided by the user along with a number of other PCR operating parameters during package creation.

* Unfortunately both the Host PDL Decompression Daemon and the whole binary is called "EGRET", creating minor confusion for the novice reader. For the sake of clarity, the term EGRET in this document will refer to the whole binary shown on the left hand side of Figure 3.4.



Note: The two binaries are in separate address space with some shared memory

Figure 3.4

Functional components of EGRET:

EGRET: Host PDL Decompression Daemon
HEIS: Host Environment Interface,
(Coordinates decomposition)
Penguin: Provides an RPC interface to Wasabi
Transport Layer: Places compressed page onto shared disk
Chameleon: Provides data compression
PortLib: Portability Library

EGRET is implemented using the C programming language and coordinates the entire process of Postscript PDL decomposition, including driving Wasabi. It runs atop SUNOS 4.1 as a separate process (private address space) from Wasabi. A mechanism is required to pass the following information between these two processes: job control information, printing commands, Postscript PDL data, rasterized frame buffer and detected errors. A combination of Remote Procedure Calls (section 3.5) and shared memory are used to implement the information exchange capability.

Use of Remote Procedure Calls between the EGRET and Wasabi processes enabled both to interact, yet remain separate in the following criterion: UNIX processes, address space, runtime environment. This separation allows each system to be designed, developed, implemented and debugged with a greater degree of independence. It also means that there is no need to port the EGRET software into the Cedar Environment which may not be such a simple task (section 4). The disadvantages of this implementation method are related to complexities introduced when the software is not totally integrated and time delays caused by the additional layers of communication protocol (RPC) introduced.

4.0 ⊕⊘ Porting C to the Cedar environment

There are potential problems involved in porting C code written on foreign software platforms, eg, using standard UNIX libraries (eg, libc.a), to the Cedar environment. These obstacles are related to the fact that Cedar supports multiple-threaded processes while the UNIX programming model allows only one thread per process. This code may be incompatible within a multiple threaded environment and not "thread-safe".

Operating within the context of a thread and not a heavyweight process, user code must be "a good citizen" and not interfere with the operation of the Cedar/PCR environment. The following rules should be obeyed.

a. Do not invoke any system calls which affect the operation or properties of the UNIX heavyweight process since the operating system state is being shared with other threads in the process. For example, setting any limits on maximum system resources consumed by the process, such as the number of file descriptors that may be supported.

b. Refrain from the use of UNIX signals and signal handlers. The signal mechanism is used frequently during the implementation of the PCR. For example, the signal SIGALRM is used by the PCR scheduler to schedule a new thread for execution.

c. Refrain from using the interval timer because PCR is using it for scheduling purposes.

Any source code must also be reentrant or monitored. This refers to a condition whereby multiple processes are concurrently executing within a procedure with the integrity of any global data being preserved. A vehicle for providing synchronized access to shared data is a monitor. This mechanism enforces mutual exclusion by allowing only one process executing within at any time.

4.1 ⊘ Protocol for the porting process

a. Get rid of the UNIX I/O system calls (eg, open, read, write, etc) and replace with a new set of PCR I/O interface calls. The tables shown in the next section lists the "thread-safe" equivalents for the UNIX I/O system calls. [**Warning**: although these procedures should be safe, there are no absolute guarantees].

Stored on the file drawer [Kent:ESCP10:Xerox]/RLi are UNIX csh command files **convert2PCR.csh** and **convert2PCR.data** which takes a C program and generates an equivalent "thread-safe" C program whereby all instances of UNIX I/O system calls (see section A of the Appendix) are replaced by their PCR function equivalents. Users should retrieve these two files and execute the following command within the UNIX shell interpreter:
convert2PCR.csh <Input C filename> <Output C filename>

A second means of replacing the UNIX I/O system calls is to retrieve the C header file stored as **[Kent:ESCP10:Xerox]/RLi/definePCR.h** and include within all C program modules. This header file utilizes the token substitution facilities provided by the C compiler preprocessor.

b. Any routines with global or static variables may be protected by a monitor if there is a possibility that more than one thread is executing here at the same time. A monitors is a collection of procedures, variables and data structures packaged in a manner where global data integrity is guaranteed by insuring only one process can be active in the monitor at any instant. This technique was used by developers of Wasabi during integration of the font processing software package.

Monitors are available to both Cedar/Mesa and C programmers. It is available as a built-in language feature within Cedar/Mesa [Ref 5] and implemented by C-based PCR function calls such as **XR_MonitorEntry** and **XR_MonitorExit** (defined within PCR threads interface **Threads.h** [Ref 23]). Another alternative would be to redefine global variables in a thread-private manner.

c. Be careful about allocating memory using **malloc**. The garbage collector needs to know about any dynamic memory that is being allocated. The preferred routine for memory allocation is **GC_malloc**. The function **GC_malloc_atomic** can be used if the referent object will not contain any pointers; this may be used for storing large bitmaps.

d. Don't use UNIX signals. The PCR code uses several of the signals to perform various actions such as SIGBUS to detect writes to protected pages, SIGALRM to do time slicing amongst threads.

Appendix A. ⊘ UNIX system call replacements

This table illustrates the equivalent PCR functional replacements (C and Cedar/Mesa version) for the UNIX system calls (information taken from [Ref 21- 23]. Refer to these documents for more details). Special mention for those items marked by a numerical superscript are at the end of the table.

UNIX C	PCR C	PCR Cedar/Mesa
accept	XR_Accept	UnixSysCalls.Accept
access	XR_Access	
bind	XR_Bind	UnixSysCalls.Bind
chmod	XR_ChMod	UnixSysCalls.ChMod
close	XR_Close	UnixSysCalls.Close
connect	XR_Connect	UnixSysCalls.Connect
errno[1]	XR_GetErrno	UnixErrno.GetErrno
	XR_SetErrno	UnixErrno.SetErrno
fchmod	XR_FChMod	UnixSysCalls.FChMod
fcntl[2]	XR_FCntl	
fork	XR_Fork	UnixSysCallExtensions.Spawn
		UnixSysCallExtensions.CDSpawn
fprintf[4]	XR_FPrintF	
fstat	XR_FStat	UnixSysCalls.FStat
fstatfs	XR_FStatFS	UnixSysCalls.FStatFS
fsync	XR_FSync	UnixSysCalls.FSync
ftruncate	XR_FTruncate	UnixSysCalls.FTruncate
getdents	XR_GetDEnts	UnixSysCalls.GetDEnts
getdomainname	XR_GetDomainName	UnixSysCalls.GetDomainName
getdtablesize	XR_GetDTableSize	UnixSysCalls.GetDTableSize
getegid	XR_GetEGID	UnixSysCalls.GetEGID
geteuid	XR_GetEUID	UnixSysCalls.GetEUID
getgid	XR_GetGID	UnixSysCalls.GetGID
getgroups	XR_GetGroups	UnixSysCalls.GetGroups
gethostid	XR_GetHostID	UnixSysCalls.GetHostID
gethostname	XR_GetHostName	UnixSysCalls.GetHostName
getmsg	XR_GetMsg	UnixSysCalls.GetMsg
getpagesize	XR_GetPageSize	UnixSysCalls.GetPageSize
getpeername	XR_GetPeerName	UnixSysCalls.GetPeerName
getpgrp	XR_GetPGrp	UnixSysCalls.GetPGrp
getpid	XR_GetPID	UnixSysCalls.GetPID
getppid	XR_GetPPID	UnixSysCalls.GetPPID
getrusage	XR_GetRUsage	UnixSysCallsExtras.GetRUsage
getsockname	XR_GetSockName	UnixSysCalls.GetSockName
getsockopt	XR_GetSockOpt	UnixSysCalls.GetSockOpt
gettimeofday	XR_GetTimeOfDay	UnixSysCalls.GetTimeOfDay
getuid	XR_GetUID	UnixSysCalls.GetUID
ioctl[2]	XR_IOCtl	UnixSysCalls.IOCtl
kill	XR_Kill	UnixSysCalls.Kill
killpg	XR_KillPG	UnixSysCalls.KillPG
link	XR_Link	UnixSysCalls.Link
listen	XR_Listen	UnixSysCalls.Listen

UNIX C	PCR C	PCR Cedar/Mesa
lseek	XR_LSeek	UnixSysCalls.LSeek
lstat	XR_LStat	UnixSysCalls.LStat
malloc	GC_malloc	
mincore	XR_MInCore	UnixSysCalls.MInCore
mkdir	XR_MkDir	UnixSysCalls.MkDir
mknod	XR_MkNod	UnixSysCalls.MkNod
mmap	XR_MMap	
mprotect	XR_MProtect	
msync	XR_MSync	
munmap	XR_MUnmap	
open	XR_Open	UnixSysCalls.Open
poll[3]	XR_Poll	UnixSysCalls.Poll
printf[4]	XR_PrintF	
profil	XR_Profil	UnixSysCalls.Profil
putmsg	XR_PutMsg	UnixSysCalls.PutMsg
read	XR_Read	UnixSysCalls.Read
readlink	XR_ReadLink	UnixSysCalls.ReadLink
readv	XR_ReadV	UnixSysCalls.ReadV
recv	XR_Recv	UnixSysCalls.Recv
recvfrom	XR_RecvFrom	UnixSysCalls.RecvFrom
rename	XR_Rename	UnixSysCalls.Rename
rmdir	XR_RmDir	UnixSysCalls.RmDir
send	XR_Send	UnixSysCalls.Send
sendto	XR_SendTo	UnixSysCalls.SendTo
setsockopt	XR_SetSockOpt	UnixSysCalls.SetSockOpt
shmat	XR_ShmAt	
shmctl	XR_ShmCtl	
shmdt	XR_ShmDt	
shmget	XR_ShmGet	
shutdown	XR_Shutdown	UnixSysCalls.Shutdown
socket	XR_Socket	UnixSysCalls.Socket
sprintf[4]	XR_SPrintF	
stat	XR_Stat	UnixSysCalls.Stat
statfs	XR_StatFS	UnixSysCalls.StatFS
symlink	XR_SymLink	UnixSysCalls.SymLink
sync	XR_Sync	UnixSysCalls.Sync
truncate	XR_Truncate	UnixSysCalls.Truncate
unlink	XR_Unlink	UnixSysCalls.Unlink
utimes	XR_UTimes	UnixSysCalls.UTimes
write	XR_Write	UnixSysCalls.Write
writev	XR_WriteV	UnixSysCalls.WriteV

Notes:

1 Retrieving the error code incurred in a UNIX system call involves invoking a function rather than accessing external variable "errno"

2 Extremely dangerous to use. Only some uses will work, others will thwart PCR implementation. Try to avoid use if possible. In all cases, do not manipulate the "no delay" or "no blocking" flags. PCR uses these flags internally as part of it's I/O system

3 Only a restricted form is allowed. Only one file descriptors can be examined and it must be valid

4 A maximum of eight print format variables allowed

Appendix B. ⊘ Unimplemented UNIX system calls

Some UNIX system calls are not valid or unimplemented within the Cedar environment. A list of the unimplemented UNIX system calls are shown below along with the indexed reason for this exclusion. For a more detailed explanation, refer to UnixSysCallsDoc.tioga [Ref 21].

Function	Reason	Function	Reason
acct	2	select	1 Use poll equiv
adjtime	2	semctl	5
asyncdaemon	4	semget	5
audit	2	semop	5
auditon	2	setauid	2
auditsvc	2	sbrk	4
brk	4	setgroups	3
chdir	3	sethostname	2
chown	2	setitimer	4
chroot	2	setpgrp	3
creat	1 Use open equiv	setregid	3
dup	4	setreuid	3
dup2	4	setpriority	3
execve	4	setrlimit	3
exit	4	settimeofday	2
fchown	2	setuseraudit	2
flock	4	shmctl	4
getauid	2	shmget	4
getdirentries	1 Use getdent equiv	shmop	4
getitimer	4	sigblock	3
getpriority	3	sigpause	3
getrlimit	3	sigsetmask	3
mmap	4	sigstack	3
mount	2	segvec	3
mprotect	4	socketpair	4
msgctl	5	swapon	2
msgget	5	syscall	4
msgop	5	umask	3
munmap	4	uname	1 Use gethostname equiv
nfssvc	4	unmount	2
pipe	5	vadvise	3
ptrace	3	vhangup	No meaning in PCR
quotactl	2	wait	4
reboot	2	wait8	4
recvmsg	1 Use recv, recvfrom equiv	wait4	4
sendmsg	1 Use send, sendto equiv		

Reasons for exclusion from Cedar environment

1. The function is obsolete or can be emulated with other calls
2. Function is available only to the super user or affects only the global Unix state
3. Affects the properties of a single UNIX heavy-weight process
4. It's use interferes with the PCR kernel implementation.
5. Within PCR tasks, use of shared memory, monitors and condition variables are recommended.
   For communicating with other UNIX processes outside PCR, use FIFO files (aka pipes).

Appendix C: Selected Reference Materials

An Introduction to the Portable Cedar Basement Documentation

Mark Weiser

Abstract: This document is an overview of the organization of a set of documents which together describe the Portable Cedar Basement. You should certainly read (or at least browse) this first before trying to tackle the others.

Created by: Mark Weiser

Maintained by: Mark Weiser <weiser.pa>

Keywords: PCR, Cedar, PCedar, documentation

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

For Internal Use Only - draft

Introduction

This document is an overview of the organization of a set of documents which together describe the Portable Cedar Basement. You should certainly read (or at least browse) this first before trying to tackle the others.

DEFINITION: *The Portable Cedar Basement is that code, provided by PARC to SSU, needed for runtime support of BWS and Viewpoint code running on Sun platforms.*

As a consequence of this definition, the Mimosa compiler and friends are not part of the Portable Cedar Basement because they are tools for doing the BWS and VP port, but are not actually used for runtime support in applications. Similarly, documentation about PViewers on Suns, or the Cedar environment on Suns generally, is not part of the Portable Cedar Basement.

The Portable Cedar Basement consists of two parts:

1. PCR
2. Cedar/Mesa Support

Part 1, PCR, contains no Cedar or Mesa specific code, and is written completely in C. PCR should never be used directly by Cedar or Mesa programmers. Mesa interfaces exist in Cedar/Mesa Support for accessing all PCR functionality. However, the PCR documentation is sometimes the best source of information about what is really going on.

Part 2, Cedar Support, itself contains three parts:

a. CedarPreBasics

b. CedarCore

c. BasicCedar

Part 2.a, CedarPreBasics, is the first layer of direct Cedar support. CedarPreBasics itself is written entirely in C. CedarCore is almost all Cedar/Mesa, with two small C-language modules. BasicCedar is all Cedar/Mesa. For much more information about these layers and what is in them, see PCedarOverview.tioga.

The next section gives an outline of the complete set of documents, and a brief description of each. All the documentation is at least indirectly referenced in [PCedar1.2]<Top>PCedarBasementDoc.df. A bringover of PCedarBasementDoc.df should get your hands on everything referenced here to whatever depth.

Outline

I. **Introduction** - *all controlled by [PCedar1.2]<Top>PCedarBasementDoc.df*

  A. **PCedarBasementIntroduction.tioga** - This document.

  B. **PCedarOverview.tioga** - An overview of all the code and interfaces in the basement, and pointers to more detailed documentation and status.

II. **PCR** - *all controlled by [PCedar1.2]<Top>PCRDoc.df*

  A. **The Portable Common Runtime Approach to Interoperability** - technical paper about PCR, as submitted to SOSP.

  B. **Detailed PCR interfaces**

    1. **ThreadsInterfaceDoc.tioga**

    2. **GCInterfaceDoc.tioga**

3. **LoadingInterfaceDoc.tioga**
4. **CommandloopInterfaceDoc.tioga**
5. **CleanUpNeededDoc.tioga** - where we admit our mistakes but pledge to do better.

C. **PCRDoc.tioga** - how to actually run PCR

D. **PCRSymtabUtilitiesDoc.tioga** - ldhide, transform←symtab, etc.

III. Cedar Support

A. **ExperiencesCreatingAPortableCedar.tioga** - technical paper about Cedar support, to appear in SIGPLAN '89 conference. *Controlled by [PCedar1.2]<Top>PCedarDoc.df*

B. **CedarPreBasics** - *all controlled by [PCedar1.2]<Top>CedarPreBasics-Source.df*

1. **CedarPreBasicsInterfaceDoc.tioga** - description of the various CedarPreBasics routines, except for installation support.
2. **InstallationNotes.tioga, InstallationSupport.mesa** - documentation of the installation support part of CedarPreBasics.

C. **CedarCore** - *documentation accessible via [PCedar1.2]<Top>CedarCoreDocumentationWorld.df*

1. **RuntimeSupport** -- mesa and tioga
2. **RopePackage** -- mesa only
3. **SafeStoragePackage** -- mesa only
4. **RegisterRefLiteralImpl** -- mesa and tioga
5. **VMPackage** -- mesa and tioga
6. **RealPackage** -- mesa and tioga
7. **RasterOpPackage** -- mesa and tioga
8. **Faces** -- mesa and tioga
9. **Communication**-- mesa and tioga

D. **BasicCedar** - *documentation accessible via [PCedar1.2]<Top>BasicCedarDocumentationWorld.df*

1. **Debugger** -- mesa only
2. **BootPackages** -- mesa only (but BasicPackages refers to, out of date)
3. **BasicTimePackage** -- mesa only
4. **IOPackage** -- mesa and tioga
5. **Greet** -- mesa only
6. **BasicPackages** -- mesa and tioga (tioga out of date)
7. **ProcessPropsImpl** -- mesa only
8. **UnixSysPackage** -- mesa and tioga
9. **UXStringsPackage** -- mesa and tioga
10. **UXIOImpl** -- mesa only

An Overview of the Portable Cedar Basement

Carl Hauser, Mark Weiser

Abstract: This is high-level introduction to the low-level concepts and facilities of the Portable Cedar (PCedar) system. It superficially describes the major components of the system referring the reader to more detailed documentation of the individual components. More importantly, it describes the relationships between these components. Therefore, this document is a good starting place for people wanting to understand more about PCedar as a whole.

Created by: Carl Hauser

Maintained by: Carl Hauser <chauser.pa>

Keywords: Cedar, CedarPort, CedarTuning, mesa, portableCedar, Sun

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

For Internal Use Only

1. Introduction

Portable Cedar (PCedar) is a reimplementation of the Cedar system in use at the Computer Science Laboratory of the Xerox Palo Alto Research Center. PCedar is intended to make the unique collection of features of Cedar available to a much wider audience by allowing the system to run on industry-standard hardware. In this introduction, I will briefly introduce the low levels (basement) of the PCedar system, trying to show how they fit together. Later sections will deal with each of these in some detail.

PCedar is implemented on top of PARC's so-called Portable Common Runtime (PCR). PCR provides garbage collection, lightweight threads, and dynamic program loading in a language-independent fashion. PCR also provides a simple command interpreter used to control dynamic loading of other parts of the system. PCedar uses these facilites of the PCR to implement the Cedar language constructs surrounding the Cedar notion of PROCESS and various facilities involving garbage-collected, dynamically-typed, allocable storage. The PCR is intended to support other languages as well. We intend to promote its use for languages such as C, Modula-3, LISP, and C++. PCR is written mostly in C, with a smattering of machine-dependent and operating-system-dependent assembly code. It is maintained with makefiles on a Unix system.

In order to resolve references through interfaces and to support the particular runtime type encoding used by the Cedar compiler, more support is needed than provided by PCR. The CedarPreBasics layer contains this support. CedarPreBasics registers new commands with PCR's command interpreter to allow Cedar/Mesa programs to be installed and run. CedarPreBasics is written exclusively in C. CedarPreBasics and the remaining components of PCedar are maintained using the DF software and other tools in the Cedar and PCedar environments. CedarPreBasics is documented in CedarPreBasicsInterfaceDoc.tioga, InstallationNotes.tioga, and InstallationSupport.mesa.

Once CedarPreBasics is available, Cedar/Mesa programs can be run with intermodule, type checked reference resolution performed by CedarPreBasics. The next layer to be loaded is CedarCore which completes the runtime support for the language (amongst other things). CedarCore contains the code supporting Cedar/Mesa SIGNALs together with interfaces and implementations connecting the Cedar language view of PROCESSes to the PCR implementation. CedarCore also provides interfaces for machine-independent, commonly used, low level arithmetic, logical, and storage overlay operations (Basics, PBasics, PBasics16, Checksum). CedarCore also contains the implementations for the Cedar language primitives ROPE and ATOM. Once CedarCore has been run, programs using the full Cedar/Mesa language may be run. CedarCore is written mostly in Cedar with a bit of C (which (c)(sh)ould now probably be removed).

Above CedarCore lies BasicCedar. We have gathered many commonly used facilities into this package so that they are available to all Cedar programmers. Various hash tables (RefTab and friends), access to the operating system (UnixSysCalls), time (BasicTime), Cedar's IO streams (IO), and some debugging support (Debugger) are examples of the facilities found here. The distinction you should draw between CedarCore and BasicCedar is that CedarCore supports the Cedar language while BasicCedar embodies low-level aspects of the Cedar programming environment: those facilities deemed so important that they should be available to every Cedar program. Although this distinction has not been followed exactly in dividing packages between CedarCore and BasicCedar it probably should have been. BasicCedar is written entirely in Cedar.

We have now arrived at the point where the system stops being so precisely layered. Futhermore, it begins to look and feel a lot like Cedar on D-Machines. Above the BasicCedar level many of the programs for PCedar share much of their source code with their D-Machine counterparts (DCedar). The interested reader can gain a fair appreciation of PCedar from this level up by reading [Swinehart et al. *A Structural View of the Cedar Programming Environment*, Xerox PARC Report CSL-86-1].

2. PCR

The components of PCR are Threads, GC, Loading, and Commandloop.

2.1 Threads

The components of threads are described in a number of C-language ".h" header files. There are basically two parts to the threads world: one for threads themselves and related activities (monitors, condition variables, exception handling, etc.), and one for I/O. Threads themselves are described by the header files: BasicTypes.h, Errno.h, Threads.h, and ThreadsBackdoor.h. I/O is described by the interfaces Cour.h, SPPEmu.h, ThreadsMsg.h, ThreadsSlaveIOP.h, UIO.h, ThreadsFrameBuffer.h, courier.h, and courmux.h.

None of the C interfaces are appropriate for Cedar or Mesa programmers, who should instead use UnixSysCalls for lowest-level I/O operations (and preferably use something higher like StreamIO instead of lowest level I/O), and the Process interface or Mesa language mechanisms for threads interactions.

Full documentation for PCR threads is in ThreadsInterfaceDoc.tioga, which has the following sections:

Time -- milliseconds to/from ticks, timeouts.
Monitors -- initialize, enter/leave monitors.
Condition Variables (CV's) -- abort, wait, timeout on conditions.
Context Save/Restore -- setjmp/longjmp replacements for internal use.
Signal/Error Interface -- general mechanisms for integrating threads with Unix signals.
Thread properties -- attach and query general client thread-specific data.
Thread Priorities -- change priorities.
Thread Create/Destroy runtime support -- fork, join, detach, abort, etc.
Doing low level output from threads -- very raw message output. Deprecated.
Errno -- using Unix errno in threads.

2.2 Garbage Collection

The garbage collector provides language-independent automatic storage management for all clients of PCR and above. PCR provides only garbage collected storage--there is no other kind of storage. Mesa and Cedar programs should not access the PCR storage management interfaces directly, but use the Mesa NEW construct, or other Mesa interfaces.

Full documentation the storage management interface is provided by GCInterfaceDoc.tioga, which has the following sections:

Externally Visible Variables -- XR←gcIgnoreDataAndBSS and XR←gcVersion.
Informational Routines -- look at global storage statistics
Routines to Control Behavior -- control collector behavior, include progress printing.
Routines to cause behavior -- the allocation routines.

Unix interface replacements -- alternative allocation interfaces.

To be called by initializing world -- not for client use.

Finalization -- still in flux.

Things used by the GC world -- INTERNAL USE ONLY.

2.3 Dynamic loading

The dynamic loader provides a mechanism for loading C and Cedar code into the PCR system. There are basically two calls: one to load a file (**load←file**) and one to interrogate symbols in the file (**get←sym←val**). There are also lots of variants of these, and various other interrogatory routines (e.g. **XR←GetNumLoadedProcedureSymbols()**). The interface to the dynamic loader is probably the least clean of any in PCR.

Full documentation is in LoadingInterfaceDoc.tioga.

2.4 Commandloop

These routines implement the very basic PCR user interface, which is just a simple command interpreter (read-eval-print loop). These routines are NOT intended for direct use by Cedar or Mesa programmers. There are Cedar/Mesa interfaces which provide equivalent functionality in cleaner ways. These routines provide access to the PCR Unix command line, parsing of arguments, etc.

Full documentation is in CommandloopInterfaceDoc.tioga.

3. CedarPreBasics

CedarPreBasics contains a number of interfaces for raw support of the Cedar language. **CedarPreBasics** is written in C. By including **CedarPreBasics** into PCR one can then run a subset of Cedar (no signals, no ref-literals, no ropes, no processes). All the routines in **CedarPreBasics** are for internal use only. No routines in **CedarPreBasics** are for direct calling by Cedar/Mesa or C applications. **CedarPreBasics** contains support for installation and management of Cedar interfaces and start traps, floating-point arithmetic, bit and byte moves, REF management, and several other miscellaneous things. **CedarPreBasics** also registers some additional commands with the PCR command interpreter.

Full documentation is in CedarPreBasicsInterfaceDoc.tioga, which has the following sections:

InstallationSupport -- start traps, interfaces, cedar symbols

CompilerSingleReal -- floating-point number support

Cmds -- register some additional commands, no procedural interface

Basics -- some basic routines required by the code generated by the Mimosa backend

Safestorage -- -literal and storage management interfaces from Cedar to PCR

Cedarextra -- additional routines required by the code generated by the Mimosa backend

4. CedarCore

The components of CedarCore are (February 1. 1989) RuntimeSupport. RopePackage. SafeStoragePackage. RegisterRefLiteralImpl. VMPackage. RealPackage. RasterOpPackage. Faces. and Communication.

4.1 RuntimeSupport

The components of RuntimeSupport are RuntimeErrorImpl. SignalsImpl. ChecksumImpl. UnboundImpl. ProcessImpl. and BasicsImpl. In addition to these Cedar implementation modules. RuntimeSupport also contains two C modules: SignalSupport and ProcessSupport.

RuntimeErrorImpl

This module exports defined signal and error values to the RuntimeError interface. Things here and in SignalsImpl and SignalSupport are carefully arranged so that. for example. RuntimeError.Aborted is identical to the language builtin signal ABORTed and similarly for other language builtins. RuntimeErrorImpl also defines other signals and errors that could be raised as a result of programs doing illegal things in the language (NarrowRefFault. for example. is defined here).

SignalsImpl and SignalSupport

These modules support the Cedar/Mesa SIGNAL and ERROR language feature. There are two aspects of this feature. First. each process needs a signal handling environment maintained for it as it enters and leaves scopes protected by catch phrases. To do this. the compiler generates calls to XR←PushHandler() and XR←PopHandler(). These are implemented in SignalSupport. They maintain a stack of handlers. The second aspect of signal handling is actually raising and catching signals. The Mimosa-generated code for raising a signal is a call on XR←RaiseError() or XR←RaiseSignal(). These are implemented in SignalSupport. but the code there simply calls procedures in SignalsImpl. SignalsImpl.SignalHandler does the real work.

The initialization of SignalsImpl connects the ERROR and SIGNAL values declared in RuntimeErrorImpl and the procedures declared in SignalsImpl into the C external name space by passing them to a procedure in SignalSupport. SignalSupport assigns these values to C external variables. This is a hand-crafted import into C from Mesa. Better ways are available to do this and it deserves a second look.

The design of the PCedar signal handling machinery is intended to provide sufficiently similar semantics to the PrincOps design that programs using SIGNAL and ERROR in the usual ways will not be affected.

The PCedar design is actually somewhat more powerful than the PrincOps design and interacts better with INLINE procedures. The conservative approach to programming with the new design is to continue using common DCedar signal handling idioms. This will also be necessary to maintain source code compatibility between DCedar and PCedar.

As noted above. PCedar uses dynamic handler establishment which is expected to be somewhat slower. relative to procedure call. than the (static) DCedar mechanism.

ChecksumImpl

ChecksumImpl provides INLINE and callable procedures for computing 16 bit checksums of sequences of 16 bit words. The computed checksum should agree with what's computed in

DCedar for the same sequence. This is important because these checksums are passed over the network.

The declaration of Checksum.ComputeChecksum had to change from DCedar because PCedar POINTER variables have 4-byte alignment while DCedar POINTER variables have 2-byte alignment. In order to be able to compute checksums starting at positions with 2-byte alignment, another parameter is required giving the offset from a 4-byte aligned target.

Do we need a standard 32 bit checksum program for internal/PCedar only use?

UnboundImpl

This module installs code used by the installation support component of CedarPreBasics to raise RuntimeError.UnboundProcedureFault. Since CedarPreBasics is written in C it is hard to do signalling there. This is another instance of Mesa-C intercalling that needs to be revisited.

ProcessImpl and ProcessSupport

Together, these provide the needed connection between Cedar/Mesa and the threads implemented in PCR. For the most part, operations on threads, monitors and condition variables in Mesa programs can be translated directly into PCR calls. The two exceptions are: WAIT on a condition variable, where the PCR primitive returns a failure indication if the thread is aborted while waiting, but the Cedar/Mesa specification requires that the ABORTed error be raised; and JOIN with another thread, where the PCR primitive returns a failure indication if the joined process is invalid, but the language specification requires the InvalidProcess error be raised. ProcessSupport provides the XR←Join() and and XR←Wait() procedures to be called from Mimosa-generated code and implements them with appropriate calls on the PCR and signal machinery.

ProcessImpl implements the Cedar/Mesa Process interface by making calls on the PCR.

JOIN and Process.Detach are now implemented safely because the runtime uses enough bits for process IDs that they need never be reused. However, the language still treats them as unsafe. For programs to be portable between the DCedar and PCedar worlds they must continue to treat all uses of JOIN and Process.Detach as UNSAFE.

BasicsImpl (and the Basics, PBasics and PBasics16 interfaces).

BasicsImpl implements the FillBytes and FillWords procedures of the Basics interface. Most of the procedures in the Basics interface and all of those in PBasics and PBasics16 are INLINE. These interfaces define low level types and operations. The philosophy is as follows: Basics is an interface describing types and operations in the "natural" word length of the machine. For PCedar it is mostly 32-bit oriented, whereas in DCedar it is 16-bit oriented. PBasics is explicitly 32-bit oriented in both worlds while PBasics16 is 16-bit oriented in both worlds.

Clients are encouraged to write portable code using the PBasics and PBasics16 interfaces.

PBasics and PBasics16 incorporate operations and descriptions that reside in DCedar's PrincOps and PrincOpsUtils interfaces. Programs using PrincOps or PrincOpsUtils must change for PCedar. If PBasics and PBasics16 don't have enough functionality to allow porting they need to be extended or you need to look elsewhere.

4.2 RealPackage

Various operations on Cedar/Mesa REAL numbers. This is not language support per se. Hence, it is a candidate for moving to BasicCedar.

4.3 RopePackage

This supports the Cedar Rope.ROPE type and its friends.

4.5 SafeStoragePackage

Most importantly, SafeStoragePackage contains AtomImpl supporting the Cedar ATOM type. It also has implementations for the List, RefQueue, SafeStorage, and UnsafeStorage interfaces. This particular packaging occurs for historical reasons and should be revisited.

SafeStorageImpl has procedures for creating objects of arbitrary type and for examining the type of objects. Both are used in AtomImpl. There's a lot of cruft here including procedures that are unimplemented because their utility in PCedar is questionable while the code was stolen from Cedar.

4.4 RegisterRefLiteralImpl *- see also SafeStorage interface in CedarPreBasics*

Further support for Rope.ROPE, Rope.Text, REF TEXT and ATOM. The problem that this package solves is that objects of these types must be in allocated storage, prefixed by correct type codes. The compiler, of course, has no way to produce such things. Instead, it produces a STRING value and a call to XR←GetRefLiteral() with that string and a type as argument. XR←GetRefLiteral() is itself part of SafeStorage in CedarPreBasics. When RegisterRefLiteralImpl is STARTed it registers itself with SafeStorage so that when XR←GetRefLiteral() is called, the call is passed along to RegisterRefLiteralImpl.Create.

(Note that the compiler distinguishes the type of quoted character strings by the context in which they appear. Thus "abc" may sometimes be a Rope.ROPE, sometimes a Rope.Text, sometimes a REF TEXT and sometimes a STRING. This package doesn't have to worry about it--the compiler already chose an appropriate type.)

There are two important consequences of this design: first, nothing containing a Rope.ROPE, Rope.Text, REF TEXT or ATOM literal may be *installed* before RegisterRefLiteralImpl is STARTed. Given the way Cinder generates code and the current expressive power of the configuration language the practical consequence of this is that **nothing in CedarCore may contain ATOM or Rope.ROPE literals.** The second consequence is that a version mismatch on the Rope interface will often manifest itself as an occurrence of the error RegisterRefLiteral.UnknownType instead of a runtime interface version mismatch.

4.6 VMPackage

A mere shadow of its Cedar counterpart, VMPackage for PCedar contains only routines for acquiring large chunks of storage and doesn't worry about its being *virtual* storage. CountedVM is also also available for getting help from the garbage collector in tracking large chunks of memory.

Beware: finalization isn't implemented, so dropping CountedVM objects on the floor constitutes a storage leak. This needs attention when finalization *is* implemented.

4.7 RasterOpPackage

RasterOp is a replacement for the BITBLT functionality, but written in PCedar. See RasterOpDoc for details. This probably should move up to BasicCedar.

4.8 Faces

ProcessorFaceImpl.mesa and FacesSupport.c make up this package. ProcessorFace provides access to the processor id and type. FacesSupport.c uses the C preprocessor and its conditional compilation facility to provide a runtime distinction between sparc and mc68020 processors. This also should probably move up to BasicCedar.

4.9 Communication

XNSImpl provides access to information about the local host's XNS host number and its characteristics. This should also probably move up to BasicCedar, as well. It depends on ProcessorFace, so if Faces moves, Communication must also.

Some fine tuning of the system organization is indicated: PCedar supports DARPA Internet protocols at least as well as XNS and no mention of them is made in these low levels. What have we done?

5. BasicCedar

Components of BasicCedar are Debugger, BootPackages, BasicTimePackage, IOPackage, Greet, BasicPackages, ProcessPropsImpl, UnixSysPackage, UXStringsPackage, UXIOImpl.

5.1 Debugger

This program is called when otherwise-uncaught signals occur. It has a fixed list of well-known signals whose identity it can print on the system error stream (using XR←DebugPutChar). After printing the fact that an error occurred, the debugger invokes the PCR debugger, stopping the PCR world (much like the "swatted" state (815 or 915) on a DMachine). This state can then be examined using dbx.

It should be possible to continue using the PCR in which an uncaught error occurred, but the thread in which it occurred will be frozen unless you thaw it using the PCR debugger. You do this in response to the "(stopped):" prompt. The command is "thaw <threadnum>" where <threadnum> is the number with the "*" next to it in the list of threads. This will cause the ABORTED error to be raised in the thread. Programs like the Cedar command tool deal gracefully with ABORTED, even though they don't like other signals and errors.

5.2 BootPackages

A collection of useful features: CardTabImpl, RefTabImpl, SymTabImpl, RandomImpl, SystemSiteImpl.

CardTabImpl, RefTabImpl, SymTabImpl: hash tables associating CARD, REF and ROPE keys with REF values, respectively. The hash tables adjust their size as they are loaded.

RandomImpl: a random number generator.

SystemSite: supplies information about the default GV and XNS registries/domains at the location the system is being run. These currently default to "pa" and "PARC:Xerox". We need to invent a way for these to be initialized properly. I think in Cedar it is controlled by the Machine.profile.

5.3 BasicTimePackage

Provides both time-of-day and fine grain timer services in a host-independent fashion. The principle module here is BasicTimeImpl which relies on the HostTime. The implementation of HostTime is host dependent. For Unix, we provide HostTimeUnixImpl as part of this package.

5.4 IOPackage

IOPackage provides the Cedar abstraction known as the IO.STREAM. It also has programs for formatting various Cedar objects onto streams and for parsing text on streams into Cedar values. There is also a collection of implemented streams: an empty input stream, a bit-bucket output stream, pipes, and ways to make streams from ROPEs and v.v.

5.5 Greet

A silly little program that announces itself whenever BasicCedar is started. Useful for identifying the running version of BasicCedar, provided Greet is kept up to date. Also exports the SystemVersion interface.

5.6 BasicPackages

Another collection of useful packages: CommanderImpl, PriorityQueueImpl, RopeFileImpl, RopeListImpl, RedBlackTreeImpl, ScaledImpl, PropImpl. These have more dependencies than BootPackages, so they are separated to provide a reasonable build order.

CommanderImpl: a registration mechanism for commands (as used in CommandTool) together with procedures for executing commands in particular property list environments.

PriorityQueueImpl: just what you'd expect.

RopeFileImpl: makes a buffered Rope.ROPE from the contents of a file. Fragments of the file are read as needed when references occur within the ROPE.

RopeListImpl: utilities for dealing with lists of ROPEs. (Wouldn't you just love a polymorphic language?)

RedBlackTreeImpl: a package for maintaining mappings from REFs to REFs where the client provides an ordering for the key REFs. Contrast this with RefTab which has similar function but cannot do ordered lookups or enumerations.

ScaledImpl: fixed point arithmetic with 16 bits before and after the binary point.

PropImpl: REF to REF association lists.

5.7 ProcessPropsImpl

Every process has an a-list (association list). ProcessProps gives access to it in a structured way.

5.8 UnixSysPackage

The UnixSysCalls interface is the only approved way of using Unix facilities from PCedar. It defines procedures for each of the Unix system calls that has been implemented. A few more system calls deserve implementation. and the remainder are either very difficult or nonsensical in the multithreaded world. There are also interfaces controlled by the same DF file (UnixSys.df) describing many of the data structures used across the system call interface, and an interface called UnixSysCallsExtensions with procedures for some of the system-call-like extensions provided by PCR.

5.9 UXStringsPackage

One problem in using the UnixSysCalls interface is that Unix system calls expect their character string parameters to be null-terminated with no preceding length fields. whereas all the Cedar/Mesa string objects are unterminated with preceding length fields. UXStrings is a way to back and forth between these types.

UXStringsPackage also provides UXProcs for going back and forth between C procedure values (the address of executable code) and Mesa procedure values (the address of a procedure descriptor).

5.10 UXIOImpl

UXIO has procedures for creating Cedar IO.STREAMs on Unix file system objects: named files and standard files.

6. Conclusion

This ends our brief tour of the PCedar Basement.

Appendix A - Component Catalog

[PCedar1.2] below is a Pseudoserver standing for [nadreck-nfs]<pixel1>pcedar1.2>

2. PCR

2.1 Threads

Code Type: C

Location: palain:/jaune/xrhome/INSTALLED/threads

Will PCR run without this component: No.

Object code size: 80344+5592 bytes

Restrictions on use: 1. backing file must be on local machine. 2. Requires VP=1 for packaged use.

Features not yet implemented: none.

Special tools/switches/etc. for building: none

Is the structure/location of component/element expected to:

Be removed or relocated in architecture: no.

Be rewritten or written: A rewrite for better debugging support throughout PCR is now underway (4/23/89). A rewrite for greater portability will be done someday soon.

Documentation:

[pcedar1.2]<Documentation>ThreadsInterfaceDoc.tioga

(Controlled by [pcedar1.2]<top>PCRDoc.df).

Test code:

Status: test code is in palain:/jaune/xrhome/INSTALLED/threads

2.2 Garbage Collection/Storage Management

Code Type: C

Location: palain:/jaune/xrhome/INSTALLED/gc

Will PCR run without this component: No. This is the only legal method of obtaining non-stack storage, and PCR needs this. There is an interface for turning off the garbage collection part of PCR Storage Management, at the peril of running out of storage. PCR Storage Management has no explicit free.

Object code size: 16k code, 1.4k data, on a SPARC (bytes)

Restrictions on use: 1. Current collector is stop-the-world, which can affect interactive or real time programs. 2. Pointers must not be hidden from the collectors view: e.g. by XOR them with a constant. 3. Only pointers to the beginning of objects count for holding onto them. 4. GC performance is greatly improved if large pointer-free blocks are allocated through an alternative interface which inform the collector that they have no pointers.

Features not yet implemented: 0. Finalization. 1. Parallel collection, to get around stop-the-world. 2. Generational collection, to do most collections faster. 3. Counting pointers to the middle. 4. Using Cedar/Mesa type information to locate pointers faster and more accurately.

Special tools/switches/etc. for building: -DPRINTSTATS - gather statistics after each collection.

-DPRINTTIMES - gather timing information

-DNTFY←KLUDGE - to run with SunView windows

-DFINALIZE - to include the finalization code

Is the structure/location of component/element expected to:

Bug fixes: The basic collector has been extremely solid for 9 months--no bugs. The proposed new features required a complete rewrite--there will be bugs then.

Be removed or relocated in architecture: no.

Be rewritten or written: Yes, all of the proposed changes except finalization require more-or-less a rewrite from scratch.

Documentation:

[pcedar1.2]<Documentation>GCInterfaceDoc.tioga

(Controlled by [pcedar1.2]<top>PCRDoc.df).

Test code:

Status: There is test code in the gc subdirectory of the PCR/INSTALLED world. To find out how to run the test code, read the *makefile* there. This test code has been built up during GC development work, is used on new releases, and has found many bugs before final release.

2.3 Dynamic loading

Code Type: C

Location: palain:/jaune/xrhome/INSTALLED/loading

Will PCR run without this component: yes, if no dynamic loading is done and no DBX debugging is done.

Object code size: 12408 + 1056 bytes

Restrictions on use: none

Features not yet implemented: Dynamic library searching, support for COFF files.

Special tools/switches/etc. for building: none

Is the structure/location of component/element expected to:

Change; bug fixes: Dynamic library searching will be added soon.

Be removed or relocated in architecture: no

Be rewritten or written: no

Documentation:

Type: [pcedar1.2]<Documentation>LoadingInterfaceDoc.tioga

Test code:

Status: written, in palain:/jaune/xrhome/INSTALLED/loading.

3. CedarPreBasics

Code Type: C

Location: [PCedar1.2]<Top>CedarPreBasics.df, [PCedar1.2]<Top>CedarPreBasicsExtras.df

Will PCR run without this component: yes

Object code size: 114848 + 4760 bytes

Restrictions on use: none

Features not yet implemented: unload. Debugging-related queries.

Special tools/switches/etc. for building: makedo; switches are supplied by files included in the dfs.

Is the structure/location of component/element expected to:

Change: bug fixes: Loadstate-related aspects will have to be enhanced to support debugging and unload.

Be removed or relocated in architecture: no

Be rewritten or written: see change above

Documentation:

Type: Installation support is described, roughly, in [PCedar1.2]<Documentation>InstallationSupport.mesa and InstallationNotes.tioga. These documents are very old and need updating to conform to the later designs that were actually implemented.

Test code:

Status: never to be written by CSL

4. CedarCore

Code Type: Cedar

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change: bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec, Test Spec/Plan, plain text, etc.◀

Test code:

Status: ▶written, to be written, never to be written◀

4.1 RuntimeSupport

Code Type: Mostly Cedar, some C

Location: [PCedar1.2]<Top>RuntimeSupport-Suite.df

Will PCR run without this component: No.

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: none

Is the structure/location of component/element expected to:

Change: bug fixes: a new implementation of the SIGNAL machinery is planned within the next couple of weeks (from March 29, 1989).

Be removed or relocated in architecture: no

Be rewritten or written: see changes above.

Documentation:

Type: ▶Func Spec, Test Spec/Plan, plain text, etc.◀

Test code:

Status: never to be written by CSL (although DSBU test suites do considerable checking of this component). You might want to fill this in with what you know.

4.2 RealPackage

Code Type: Mesa

Location: [PCedar1.2]<Top>Real-Suite.df

Will PCR run without this component: no

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: none

Is the structure/location of component/element expected to:

Change; bug fixes: no

Be removed or relocated in architecture: no

Be rewritten or written: no

Documentation:

Type: [PCedar1.2]<Documentation>FloatingPointDoc.tioga. This corresponds to the real arithmetic used in PrincOps Cedar; it needs updating in consultation with R. Atkinson to correspond to the PCedar implementation.

Test code:

Status: David Goldberg and Brian Lyles have done some accuracy testing for the mathematical functions in RealPackage. Contact them for details.

4.3 RopePackage

Code Type: ▶C, Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change: bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text, etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

4.5 SafeStoragePackage

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text, etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

4.4 RegisterRefLiteralImpl

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change: bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

4.6 VMPackage

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

4.7 RasterOpPackage

Code Type: ▶C. Cedar◀

Location: ▸location◂

Will PCR run without this component: ▸?◂

Object code size: ▸bytes◂ bytes

Restrictions on use: ▸details◂

Features not yet implemented: ▸details◂

Special tools/switches/etc. for building: ▸details◂

Is the structure/location of component/element expected to:

Change: bug fixes: ▸details◂

Be removed or relocated in architecture: ▸details◂

Be rewritten or written: ▸details◂

Documentation:

Type: ▸Func Spec. Test Spec/Plan. plain text, etc.◂

Test code:

Status: ▸written. to be written. never to be written◂

4.8 Faces

Code Type: ▸C. Cedar◂

Location: ▸location◂

Will PCR run without this component: ▸?◂

Object code size: ▸bytes◂ bytes

Restrictions on use: ▸details◂

Features not yet implemented: ▸details◂

Special tools/switches/etc. for building: ▸details◂

Is the structure/location of component/element expected to:

Change: bug fixes: ▸details◂

Be removed or relocated in architecture: ▸details◂

Be rewritten or written: ▸details◂

Documentation:

Type: ▸Func Spec. Test Spec/Plan. plain text, etc.◂

Test code:

Status: ▸written. to be written. never to be written◂

4.9 Communication

Code Type: ▸C. Cedar◂

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

5. BasicCedar

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

5.1 Debugger

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change: bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

5.2 BootPackages

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec, Test Spec/Plan, plain text, etc.◀

Test code:

Status: ▶written, to be written, never to be written◀

5.3 BasicTimePackage

Code Type: ▶C, Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec, Test Spec/Plan, plain text, etc.◀

Test code:

Status: ▶written, to be written, never to be written◀

5.4 IOPackage

Code Type: ▶C, Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change: bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text, etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

5.5 Greet

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change; bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text, etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

5.6 BasicPackages

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change: bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

5.7 ProcessPropsImpl

Code Type: ▶C. Cedar◀

Location: ▶location◀

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: ▶details◀

Is the structure/location of component/element expected to:

Change: bug fixes: ▶details◀

Be removed or relocated in architecture: ▶details◀

Be rewritten or written: ▶details◀

Documentation:

Type: ▶Func Spec. Test Spec/Plan. plain text. etc.◀

Test code:

Status: ▶written. to be written. never to be written◀

5.8 UnixSysPackage

Code Type: Mesa

Location: [PCedar1.2]<Top>UnixSys-Suite.df

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: see UnixSysCalls.mesa

Special tools/switches/etc. for building: none

Is the structure/location of component/element expected to:

Change: bug fixes: no plans. Implementation bugs can be fixed. Problems with the interface will have to be addressed with Extras interfaces.

Be removed or relocated in architecture: plans call for moving Cedar to a less SunOS-specific syscalls interface sometime in the future (not PCedar1.2)

Be rewritten or written: not in PCedar1.2

Documentation:

Type: See the interface modules in the package.

Test code:

Status: never to be written by CSL

5.9 UXStringsPackage

Code Type: Cedar

Location: [PCedar1.2]<Top>UXStrings-Suite.df

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: none

Special tools/switches/etc. for building: none

Is the structure/location of component/element expected to:

Change: bug fixes: no

Be removed or relocated in architecture: no

Be rewritten or written: no

Documentation:

Type: see interfaces in package

Test code:

Status: never to be written by CSL

5.10 UXIOImpl

Code Type: Cedar

Location: part of the IO package controlled by [PCedar1.2]<Top>IO-Suite.df

Will PCR run without this component: ▶?◀

Object code size: ▶bytes◀ bytes

Restrictions on use: ▶details◀

Features not yet implemented: ▶details◀

Special tools/switches/etc. for building: none

Is the structure/location of component/element expected to:

Change: bug fixes: no

Be removed or relocated in architecture: yes; for CSL use, UXIO will be supplanted by PFS sometime during the life of PCedar1.2. UXIO can remain available (with little attention from CSL) indefinitely.

Be rewritten or written: no

Documentation:

Type: see UXIO.mesa

Test code:

Status: never to be written by CSL

The Portable Common Runtime Approach to Interoperability

Mark Weiser, Alan Demers, Carl Hauser

▶CSL-89-xx◀ ▶Month 1989◀ [▶P89-xxxxx◀]

Abstract: Operating system abstractions do not always reach high enough for direct use by a language or applications designer. The gap is filled by language-specific runtime environments, which become more complex for richer languages (CommonLisp needs more than C++ needs more than C). But language-specific environments discourage integrated multi-lingual environments, and also make porting hard (for instance, because of operating system dependencies). To help solve this, we have built the Portable Common Runtime (PCR), a language-independent and operating-system-independent base for modern languages. PCR offers four interrelated facilities: threads (light-weight processes), low-level I/O (including network sockets), storage management (including universal garbage collection), and symbol table management (including static and dynamic linking and loading). These are the four that languages and applications must share if they are going to tightly interoperate: threads so they can each multi-process while respecting each other's critical sections, I/O so they can share low-level device handles, storage management so they can pass pointers in the presence of garbage collection, and symbol table management so they can intercall and interload. PCR is "common" because these facilities can be shared among different languages, usually without recompiling. So far we have implemented C, Cedar, Scheme, and CommonLisp intercalling, and can use pre-existing C and CommonLisp (Kyoto) binaries. PCR is "portable" because it uses only a small set of operating system features. So far it has been run on SunOS Unix™ (version 4.0), Mach, and on a bare homebuilt machine with PCR itself serving as the main operating system. PCR is about 20,000 lines of C code, and about 200 lines of assembler. It is in everyday use by about ten developers at Xerox PARC (as of March 1989), and its source code is available for use by other researchers and developers.

Submitted to SOSP.

CR Categories and Subject Descriptors: ▶Class No◀ [▶**Major Classification**◀]: ▶Classification Topic◀ – ▶Descriptors, Descriptors, ... Computing Reviews categories from

January 1982 CACM go here◀;

Additional Keywords and Phrases: ▶keywords, keywords◀

Introduction

The Problem - interoperating languages

Although there are many facets to interoperability, one remains largely unassailed: tightly coupled interoperating languages. By tightly coupled, we mean to imply that an application as real-time or sophisticated as a device driver or a database management system might have different parts written in different languages. The parts could share data structures, memory, and threads of control. We also prefer to interoperate without giving one language a primary role. We believe the choice of a language should be determined by the semantic model needed, not by the degree of support from the operating environment.

The Portable Common Runtime solution

Ordinarily the level of abstraction below the language designer's is that of the operating system. We are not proposing a new operating system, but a run-time layer. We take this approach because we are interested in interoperating with existing languages and operating systems, not in making a clean break. There are many clean-break operating systems--interoperating is the greater intellectual challenge. A second reason for taking a runtime approach is that our abstractions are not alternatives to, but are built on top of, typical operating system abstractions such as virtual memory, communications, and file system. The next generation of operating systems will build abstractions such as garbage collection into their kernels. We are exploring this in practice now with the Portable Common Runtime (PCR).

PCR differs from other runtimes both in the sophistication of some of its features, and the paucity of others. Compared with the Unix standard library, for instance, it offers the new features of threads and garbage collection and dynamic loading, but does not offer string functions or sophisticated printing or input scanning. Our choice is to focus deliberately on those features which languages must share to tightly interoperate, while avoiding other features in a runtime library that are not so important to interoperation. We assume that features we do not implement can continue to be done on a language-dependent basis without seriously reducing interoperation. For instance, Cedar strings contain a length, C strings are null-terminated. One can write in either language routines which convert one representation to the other, so there is no fundamental interoperability issue. This is not true for garbage collection, say, or process model: if two programs do not share a single underlying abstraction, they must live in separate worlds.

PCR fails to solve the whole problem of language and application interoperation in at least two ways. First, it does nothing for data representation. For some kinds of interoperation, such as between spreadsheets and graphing programs, this is the key issue. Other attacks on interoperation, such as remote procedure call and Presentation Manager [Apik and Diehl 1988], do impose a standard method of data exchange. Second, PCR says nothing about a user interface. Again, other approaches, like *Open Look* and *Motif*, address this.

For additional alternatives to our approach, see the penultimate section of this paper on *Related Work*.

PCR Design Principles

The PCR design was constrained by the following principles:

1. live above the operating system.
2. let dumb applications stay dumb.
3. permit the use of existing compilers, libraries, and binaries.
4. let sophisticated applications be written.

Living above the operating system meant, in the first place, avoiding changes to operating system kernels, and the second place, not duplicating operating system functions. Therefore, for instance, PCR on Mach [Accetta et al 1986] maps PCR threads into Mach threads. However, PCR does require from its base certain functions not always available from every operating system. We require the ability to protect pages of memory, and to catch and restart from protection failures. We require that the operating system provide a file system. These features are available more and more, and so we traded off loss of portability to older O.S.'s for much greater functionality. We have not been averse to kernel changes to improve efficiency (for instance, altering SunOS 4.0 to have user-controlled virtual dirty bits), but we do not require them.

Letting dumb applications stay dumb means that, as we added potentially interfering new features, older programming styles can mostly remain intact. For instance, although we garbage collect C code, we do not require that C programmers replace their 'malloc' and 'free' calls. PCR simply ignores the 'free's, and invisibly collects 'malloc'ed space. Binary files that can be dynamically loaded in PCR can also be statically linked using the vanilla Unix 'ld' command. PCR is not perfect in this respect, as the details in following sections make clear, but it achieves a useful compromise between backward compatibility and new function, as the next paragraph indicates.

By permitting the use of existing compilers, libraries, and binaries, we help to enforce on ourselves our rule of letting dumb applications stay dumb. We co-exist with a machine's native stack and calling conventions, so compiler back-ends do not have to change, and we accept standard relocatable object file format, so precompiled code continues to work. For example, the complete SunView window system library operates unrecompiled, dynamically loaded and garbage collected in PCR. One thing that cannot work is dynamically load binaries from which relocation information has been lost.

By saying we want to permit sophisticated applications, we mean applications most naturally expressed using knowledge of PCR details. For instance, an application managing many concurrent activities will use the threads facilities. The language which has stretched our interfaces farthest has been CommonLisp, because it already has notions of dynamic loading and of garbage collection. We had to make sure we offered facilities on top of which CommonLisp language implementors could work. For example, an implementation using tagged pointers must be able to co-exist with our collector. In general, our solutions were of two types: make the interfaces more general, and provide for upcalls [Clark 1985] when there was no other way.

[Artwork node; type 'Artwork on' to command tool]

Design and Implementation of PCR

Threads

The PCR threads interface offers the usual semantics of monitors, monitor locks, condition variables, fork/join, aborting, etc. [Hoare 1974, Brinch-Hansen 1975]. As indicated above, we have worked to make the interface general enough to be used cooperatively by many different languages. PCR threads meet the runtime requirements of languages such as Cedar/Mesa [Swinehart et al 1986], Modula-2+, Modula-3, CommonLisp, and ARGUS [Liskov et al 1987]; and can easily simulate other threads packages such as Cooper's C-Threads for Mach, Sun's lwp [Sun 1988a], Bershad's [Bershad et al 1988], Doeppner's at Brown University, etc. The following overview highlights noteworthy features of our implementation.

Threads implementations fall into two categories: inside or outside the OS kernel. Implementations inside the kernel, such as Mach [Accetta et al 1986] or V [Cheriton and Zwaenepoel 1983], have explicit knowledge of multiple threads per address space, and the OS scheduler treats such threads separately.

Implementations outside the kernel generally use coroutines in a single heavyweight process. Coroutine implementations can be faster at thread switching, because they avoid any overhead associated with entering and leaving the kernel (similar to the speedup achieved by *{Synthesis}* [Pu et al 1988], although via a different method). However, their reliance on only one heavyweight process introduces a serious problem: if that process ever blocks, *all* threads are blocked. Techniques for avoiding blocking--use of the Unix non-blocking I/O primitives, for example--can alleviate this problem, but they cannot entirely eliminate it, since some kinds of blocking (e.g. page faults) cannot be predicted or even detected outside the kernel. Our implementation avoids both of these problems.

Our approach to running PCR threads in an operating system like Unix, which has no notion of a lightweight process, is to have a small number of heavyweight processes act as a pool of "virtual processors" ("VPs") to execute the many threads. All VPs execute code and data out of a common shared address space. Each VP is treated by the PCR scheduler exactly like a cpu in a shared-memory multiprocessor system.

In the normal case of a thread blocking predictably (e.g. by waiting on a monitor lock or condition) or being preempted at a timeslice, scheduling a new thread under this scheme is essentially a coroutine jump within a single VP. Non-blocking I/O and other techniques are used to make most instances of thread blocking predictable, and thus avoid most instances of VP blocking. Occasionally, however, a thread blocks unpredictably, say for a page fault or file system I/O. In that case the VP running the thread blocks; but the remaining VP's are still available for heavyweight process scheduling by the OS, and continue to run other threads. On a uniprocessor, assuming the number of available VP's exceeds the number of unpredictably blocked threads, the net effect is just to trade a heavyweight process switch (between VP's) for a lightweight switch (between threads in a single VP); some threads continue to make progress at all times. This design has an advantage on true multiprocessors as well: Since the operating system kernel for a multiprocessor can be expected automatically to schedule multiple ready-to-run processes on multiple processors, PCR should get true multiprocessing (depending only on a reasonable base kernel implementation) with no change to the implementation.

The PCR implementation relies on a relatively small number of underlying kernel features, chief of which is the ability to share memory among heavyweight processes. Since this feature exists in OS/2, the Unix SVID, Mach, SunOS, Berkeley Unix, and many other modern operating systems, we anticipate no serious portability problem. Other OS features required by PCR are the ability for heavyweight processes to interrupt one another and to catch interrupts, and the ability to define a medium-grained interval timer (our scheduler wakes up ten times a second for time-slicing). Our implementation runs better if it can also write-protect pages (used for stack red-zoning and parallel garbage collection), catch and restart from protection violations, and remap pages to different addresses. However, PCR can run in pure real memory if necessary, as illustrated by the implementation on our home-brew processor board.

Debugging of threads is currently a bit ugly, and we are working to improve it. At present, there are a few interactive commands by which one can stop all VP's, run on a single VP, freeze or thaw individual threads, or 'examine' an individual thread. Examining works like this: before examining, a normal debugger (say Unix dbx) is pointed at a prespecified VP, and a breakpoint is set at a well-known location ("XR←ExamineMe"). When the examine command is given for a thread, the thread is scheduled on that VP, and forced to execute through the breakpoint location. The specified VP hits the breakpoint with the desired thread's stack appearing as the main process stack, so the debugger is happy.

I/O

The I/O interface currently provided by PCR is a nearly-exact emulation of the Unix I/O system calls. This is certainly the least portable aspect of the PCR design, and we plan eventually to replace it. However, developing the ultimate general-purpose, powerful and fully portable I/O interface will involve substantial research and effort; the current design was simple to produce (we copied it) and has enabled us to write PCR-based applications and validate some implementation techniques.

One limitation of Unix (and some other systems as well) is particularly troublesome when combined with the implementation of threads described above: the maximum number of open files that a single heavyweight process can hold is much less than the total number of open files supported by the system. In "normal" use of Unix, with each heavyweight process running a single application, the open file limit is large enough to be uninteresting. But we want to implement network servers and other large systems using PCR; it is important that the per-heavyweight-process resource limitations of Unix not translate into system-wide resource limitations for PCR.

To deal with this problem, our implementation on top of Unix uses additional heavyweight processes as "I/O processors"("IOP's"), essentially to serve as caretakers for file descriptors. It works as follows:

A file is opened by allocating a file descriptor slot in one of the IOP's and sending a message to that IOP asking it to open the file. While the file remains open, its "real" descriptor remains in the IOP; the descriptor slots of the VP's are treated as an LRU cache of copies. To perform I/O on a descriptor, a thread first ensures that a copy of that descriptor exists in the VP's descriptor cache. If necessary, the least recently used descriptor in the cache is replaced by a copy of the desired descriptor, which is transferred from the corresponding IOP using Unix-domain IPC (Berkeley Unix) or stream operations (Unix System V). Currently, all VP's maintain identical file descriptor caches, though this constraint could be relaxed at the cost of some complexity in the implementation. The thread then attempts a non-blocking I/O operation on the descriptor. If the operation fails because it would block, the thread sends a message to the IOP asking to be notified when the descriptor becomes ready. It then waits on a condition variable, allowing the VP to schedule a different thread without blocking. Eventually the descriptor becomes ready and the IOP notifies the waiting thread, which wakes up and retries the I/O operation. This scheme works well under the obvious condition that the "working set" of descriptors fits in the VP's descriptor cache.

User code sees none of this, of course. We impose a layer of indirection in the file descriptors, and mimic all the usual Unix I/O system call layer (read, write, open, ...) by our own compatible calls. We do the same for the Berkeley Unix socket-oriented calls and the System V stream-oriented calls.

This I/O design will enable us to implement very large systems using PCR, at the cost of occasionally having to fault copies of descriptors into the VP descriptor caches.

Storage Management

In order to work for languages which cannot guarantee pointer locations, the Portable Common Runtime uses a conservative collection scheme as implemented by Boehm [Boehm and Weiser 1988]. There are actually two storage allocation systems which have been implemented for PCR. The first is a direct adaptation of Boehm's Russell collector, with additions for typed objects and finalization. The second is a new implementation which is real-time, parallel, generational but noncopying, and handles pointers to the interior of objects. Because of its unique features, this second implementation is described in more detail in a separate paper [Weiser in preparation]. Here we focus on the highlights common to both collectors, and in particular on the mechanisms common to both for finalizing objects in a conservative world, and for allowing application defined pointer definition.

Garbage collectors can be either reference counting or mark-and-sweep. Reference counting collectors require overhead on each pointer manipulation, mark-and-sweep collectors require lots of work for each collection. Conservative collectors are a new type of mark-and-sweep collector. They have only uncertain knowledge about where pointers actually occur [Bartlett 1988], but are careful to err on the conservative side of assuming something is a pointer or not. PCR collectors are all at least potentially conservative so they can work with unsafe pointer languages like C.

As Bartlett and Boehm have shown, conservative or partially conservative collectors have been shown to work for many languages. For PCR they have been extended in two ways: finalization, and extendable pointer representations.

Finalization is the method by which an application can request that it get a last chance to look at an object before it is freed. The application can abort the free at that point, or let it continue. In PCR, finalization works as follows: finalization may be requested for any object by passing it to the PCR routine 'RegisterForFinalization'. RegisterForFinalization returns a handle to the object. The handle may be turned into a true pointer to the object at any time, but does not count as a pointer for purposes of collection (i.e. an object with a handle can still be finalized). The pointer to the object is remembered inside the collector in a place where it will not ordinarily be used to mark the object.

During each collection, after the mark phase but before sweeping, the PCR collector executes the algorithm below:

```
1. Foreach unmarked finalizable object o :
2.         foreach pointer p in o:
3.                 mark p↑, and mark all p↑'s descendants
4. Foreach finalizable object o still unmarked:
5.         place o on a finalization queue
```

Note that because most objects have already been marked, the marking through of step 3 will usually terminate quickly.

This algorithm has the difficulty of never finalizing circular lists. The circularity of such lists must be broken by using a RegisterForFinalization handle for one of the links, rather than an actual pointer. This handle doesn't count as a reference, so finalization still occurs.

An alternative implementation, that was formerly used in Cedar, uses a dangerous technique of 'package ref counts', which means lying to the collector about how many references actually exist, and having no way to tell a known from an unknown reference. Our method, using explicit handles which can be turned into a pointer, is safer and less error prone.

Finalization is tricky however it is done, but it is not frequently programmed directly. For instance, in the two million lines of Cedar code in use at PARC, there are only twelves calls that register objects for finalization. A bit of care in proper programming practice, therefore, is ok. However, doing without finalization is not possible: these twelve modules include stream and network I/O, so indirectly almost everyone uses finalization.

The PCR collector is conservative and so works even for languages that permit any word in memory to contain a pointer (such as C). However, for some languages (such as Cedar and Lisp) it is possible to tell exactly when a bit pattern is a pointer. To improve performance for such languages, the PCR design has the notion of a pointer-finding upcall. We have tried this in conjunction with third party CommonLisp implementations that use tagged pointers. The pointer-finding upcall works like this:

All objects in PCR are typed by the kind of pointer-finding upcall needed to deduce their pointers. This upcall-type is intended to be based on language family type: one for tagged pointers, one for pointers dependent upon type of language-dependent data structure, another for conservative pointer-finding, etc. Language families register their pointer-finding upcall, and a root-finding upcall, with the collector at runtime, and receive a language-family type in return. All allocations in that family must then be made with that language family type as a parameter. During the mark phase of a collection, PCR does the following: it calls all root-finding callbacks to find any language dependent roots. It also marks the global roots known to it, such as the threads stacks and the registers. Finally, it marks through all the objects, upcalling as necessary to find each pointer.

The cost of the upcall is not large: measured at 2 microseconds per object on a 20 Mhz 68020 (sun-3/60), and between 300 and 500 nanoseconds per object on a 16 Mhz SPARC (sun-4/260). This is roughly comparable to the cost of conservatively examining a word in the object to see if it is a pointer, which requires at least a range check to see if it could be a value in the heap. Thus the upcall wins if it can reject non-pointers better than the conservative check, with at least a constant win of one pointer check. For instance, even supposing an absurdly high rate of pointers in objects of 25%, an upcall which positively identified such pointers in objects (by using an object type field, say), and spent an average of three conservative checktimes per pointer per object doing so, would on average be a win for all objects of size greater than 2 words.

Our most widely used PCR implementation today (March 1989) uses a collector based on Boehm's [Boehm and Weiser 1988], modified to keep a type word before the first word of the object. Its collection speed is about a half second per megabyte of active object space (assuming no paging) on a Sun-4/260 (16 Mhz SPARC).

Symbol Table Management

Finally we come to a part of the PCR that does dynamic linking and loading, stack walking, and helps out debuggers. This part of the PCR consists of three components. The first is responsible for external object file format. The second is responsible for internal management, and symbol resolution. The third does stack walking.

Component one, external object file format, reads object files into internal form where the object code can then be relocated and undefined names resolved against the rest of the PCR load state. In addition, this component is responsible for maintaining a simulated object file whose appearance is that of a normal object file representing the state of PCR if it had been statically linked. This simulated object file can be used for debugging the dynamic state of PCR at anytime using unmodified native operating system debuggers, for instance DBX.

(For performance reasons the simulated object file is not actually rebuilt at each dynamic load. Instead, a record is kept in the file of the name and relocation value of each dynamically loaded module, and then the simulated object file is built on the fly should debugging be necessary.)

Component two, the internal symbol manager, is responsible for relocating loaded object modules and resolving undefined symbol references. It will also dynamically search libraries for symbols not resolved in the current PCR load state, calling back to the external object module for interpreting libraries. This module is architecture dependent, because instruction and data formats differ among architectures.

The internal symbol manager can also be used for dynamically asking for the value of a previous symbol, and thus finding previously loaded modules by hand. If multiple modules define the same externally visible symbols, the most recently loaded module is used.

Component three implements a version of the Unix setjmp/longjmp by which return can be made to an arbitrary (cooler) point on the stack. We use our own setjmp/longjmp, because we require that the registers be returned to their values at the time the call frame was last left. Some Unix longjmp's do this (like Berkeley's Vax implementation, and Sun SPARC), some do not (like Sun 68020). By ensuring that setjmp/longjmp restore registers to their most recent values, we can use setjmp/longjmp for signal handling without inhibiting optimization. Full register-restoring longjmp may be impossible to implement on some machines, depending upon the optimization strategies. For instance, the late-register-binding strategies of the compilers and loaders for the DECWRL Titan would make such a longjmp very difficult to write.

Each dynamically loaded module is checked for two special names: 'XR←install' and 'XR←run'. If present, these are called in that order. XR←install is there for any language-dependent symbol binding routines (Cedar and Lisp use it, for instance). XR←run is the entry point to actually start executing the loaded code.

The dynamic loading code is fully compatible with existing Unix programs and libraries. Anything which can be dynamically loaded can also be statically bound into an instance of PCR. This enables us to debug PCR-based applications using dynamic loading, and then, using those same modules, easily construct a single executable program indistinguishable from any other executable binary on the machine. We use this, for instance, to make some of our common tools, like the Cedar compiler and Postscript and Interpress decomposers, look like ordinary Unix programs.

Although we have done no optimization (symbol table searches are linear, for instance!) dynamic loading is quick, faster than Sun Unix ld. The reason seems to be that ld has many general cases to handle, and must also build an output file, while we simply load and relocate in place.

Related Work

Our work builds on previous research in light-weight processes, garbage collection, library management, etc., and reference to these are in the main body of the text. In this section we collect the discussion about alternative approaches to language interoperability.

One current approach to language interoperability, exemplified in Mercury [Liskov et al 1988] and HRPC [Bershad et al 1987], uses client-server models of interoperation where remote procedure call connects, and insulates, applications in different languages. The problem here is the lack of tight coupling. Remote procedure call, even when local but across address spaces, is usually much more expensive than calls within the same address space. When the language partitioning and the client/server partitioning match, RPC does well. When they do not match, they force the application writer to introduce artificial distinctions.

Another approach to language interoperability uses a common base language to which other languages must conform. The foreign function call interfaces in Common Lisp [Sun 1988b, Franz 1988], are examples of this approach. The problem here is that the privileged language enjoys easier debugging, better access to services, and more attention from developers. The choice of language in which to write an application becomes distorted by issues beyond appropriate language semantics, and the languages interoperate assymmetrically.

A third approach to language interoperability is to standardize on a common intermediate form. This is a variation on the privileged language approach, permitting different languages to interoperate as long as they use a common compiler back-end. In spite of several attempts in this direction [e.g. Tanenbaum et al 1983], the restrictions on language designers and implementors have proven too severe for wide adoption. We hope our more modest approach (agreement on important parts of the run-time environment), by analogy with the success of common operating systems, will prove better in practice.

A fourth "approach" is to say that language interoperability is bunk. Either there is one true language, or what really matters is not language but environment. Proponents of these views either focus efforts on inventing new languages to solve all their problems [U.S. DOD 1983] or in developing the single language environment as in Smalltalk [Goldberg and Robson 1983], Interlisp [Xerox 1985], Cedar [Swinehart et al 1986], or the new DARPA environments proposal [Gabriel 1989]. We think different problems are attacked better in different languages, and that software engineers and computer scientists should not be restricted to a single semantic arrow in their quivers.

Experience and Conclusions

The Portable Common Runtime is in daily use by about twenty researchers at PARC. We are running about 500,000 lines of Cedar code on top of PCR as of March 1989, with more ported everyday [Atkinson et al 1989]. PCR is the lowest-level foundation of future work in PARC's Computer Science and Electronic Documents Labs. PCR is about 20,000 lines of C, and about 200 lines of assembler.

Several of our uses push hard on the PCR facilities. For instance, we have an X window client which creates several threads per window. We also have an Interpress printer driver which reuses lots of free storage, and so stretches the collection facilities. To bring up a full Cedar world on our Sun workstations, more than 60 large modules (totalling over 5 megabytes) must be dynamically loaded.

We routinely use PCR to intercall between C and Cedar, and intercalling with Kyoto Common Lisp receives a small amount of use. A small part of the Kyoto runtime was changed to use the PCR collector and dynamic loader: otherwise it is unchanged. We have tested the dynamic loading and automatic garbage collection of large pre-existing SunView applications, merely relinked to be relocatable instead of executable, and they run fine.

Most of our use of PCR is under SunOS 4.0, on SPARC-based processors. We also have a small amount of 68020 and Mach use. The use which shows PCR's portability best is on CSL's own SPARC-based computer, which has no operating system at all but talks through a shared memory connection to another processor running the Cedar operating system. Thus this PCR has nothing Unix-like nearby to rely upon. Bringing up this PCR from our original SunOS SPARC-based version took less than a month.

The performance of PCR is difficult to quantize: relative to what? Our main use is for Cedar, and the standard Cedar machine of the past was a Dorado, about a 4 MIPS machine. Therefore, comparing PCR on an 8 MIPS Sun ought to show things running twice as fast. On a few measures this works out: for instance, storage allocation times. On a few measures PCR is worse, such as thread switch time, largely because of overhead in saving register windows on the SPARC. For another comparison, Gabriel's lisp benchmarks on Kyoto Commonlisp have about the same performance with or without PCR. Generally we do not see PCR itself as a performance bottleneck for our applications.

For the future, we hope to see PCR in wider use, both inside and outside PARC and Xerox. The source code and documentation for PCR is available from the Computer Science Laboratory at PARC for a copying charge, no license required. We hope to interest other portable language efforts, such as C++, Objective C, and Modula-3, in using PCR as their base. And finally, we hope to see at least the facilities offered by PCR--threads, language-independent garbage collection, and user-controlled dynamic linking and loading--available in all future operating systems.

References

(Accetta et al 1986)

Accetta, J. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F. Tevanian, A., and Young, M. W., "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of Summer Usenix*, July, 1986.

(Apik and Diehl 1988)

Apik, S., and Diehl, S., "Presentation Manager and LAN Manager", *BYTE*, Vol. 13(10), October 1988, pp. 157-159.

(Atkinson et al 1989)

Atkinson, R., Demers, A., Hauser, C., Jacobi, C., Kessler, P. and Weiser, M., "Experiences Creating a Portable Cedar", to appear in the 1989 ACM SIGPLAN Conference on Programming Language

Design and Implementation. June 1989.

(Bartlett 1988)

Bartlett. J. F.. "Compacting Garbage Collection with Ambiguous Roots". DEC Western Research Lab Research Report 88/2. February 1988.

(Bershad et al 1987)

Bershad. B. N.. Ching. D. T.. Lazowska. E. D.. Sanislo. J. and Schwartz. M.. "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems". *IEEE Transactions on Software Engineering* **SE-13**. *8*. August. 1987. pp. 880-894.

(Bershad et al 1988)

Bershad. B. N.. Lazowska. E. D.. Levy. H. M.. Wagner. D. B.. "An Open Environment for Building Parallel Programming Systems". *Proceedings ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications. Languages and Systems. SIGPLAN Notices.* Vol. 23(9). September 1988. pp. 1-9.

(Boehm and Weiser 1988)

Boehm. H-J.. and Weiser. M.. "Garbage Collection in an Uncooperative Environment". *Software-Practice and Experience.* Vol. 18(9). September 1988. pp. 807-820.

(Brinch-Hansen 1975)

Brinch-Hansen. P.. "The Programming Language Concurrent Pascal". *IEEE Transactions on Software Engineering* **SE-1**. *2*. June. 1975. pp. 199-207.

(Cheriton and Zwaenepoel 1983)

Cheriton. D. and Zwaenepoel. W. "The Distributed V Kernel and its Performance for Diskless Workstations". *Proceedings of the Ninth ACM Symposium on Operating Systems Principles.* Bretton Woods. NH. October. 1983. pp. 128-140.

(Clark 1985)

Clark. D. "The Structuring of Systems Using Upcalls". *Proceedings of the Tenth ACM Symposium on Operating Systems Principles.* Orcas Island. WA. December. 1985. pp. 171-180.

(Franz 1988)

Franz Inc.. "Foreign Functions". *Allegro Common Lisp User Guide. Release 2.2.* Section 10. January 1988.

(Gabriel 1989)

Gabriel. D.. Ed. "Draft Report on Requirements for a Common Prototyping System". *SIGPLAN Notices.* V. 24. No. 3. March 1989. pp. 93-166.

(Goldberg and Robson 1983)

Goldberg. A. and Robson. D.. *Smalltalk-80: the language and its implementation.* Addison-Wesley. 1983.

(Hoare 1974)

Hoare, C. A. R., "Monitors: An Operating System Structuring Concept". *CACM* 17, *10*, October, 1974, pp. 549-557.

(Liskov et al 1987)

Liskov, B., Curtis, D., Johnson, P. and Scheifler, R. "The Implementation of Argus", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, November, 1987, pp. 111-122.

(Liskov et al 1988)

Liskov, B., Bloom, T., Gifford, D., Scheifler, R., and Weihl, W., "Communication in the Mercury System", *Proc. of the 21st Annual Hawaii Intl. Conf. on System Sciences*, Kailua-Kona, HI, January 1988, pp. 178-187.

(Pu et al 1988)

Pu, C. and Massalin, H. and Ioannidis, J., "The {Synthesis} Kernel", Computing Systems, Vol 1(1), Winter 1988, pp. 11-32.

(Sun 1988a)

Sun Microsystems, "Lightweight Process Library", *SunOS Reference Manual, Sun Release 4.0*, 1988, section 3L.

(Sun 1988b)

Sun Microsystems, "Working Beyond the Lisp Environment", *Sun Common Lisp 3.0 Advanced User's Guide*, chapter 5, part no. 800-3049-10, August 1988.

(Swinehart et al 1986)

Swinehart, D., Zellweger, P., Beach, R., Hagmann, R., "A Structural View of the Cedar Programming Environment", *TOPLAS* 8, *4*, October, 1986.

(Tanenbaum et al 1983)

Tanenbaum, A.S., van Staveren, H., Keizer, E. G., Stevenson, J. W., "A Practical Tool Kit for Making Portable Compilers", *Communications of the ACM*, Vol. 26(9), September 1983, pp. 654-660.

(Weiser in preparation)

Weiser, M., "Garbage Collection as an Operating System Primitive", in preparation.

(U.S. DOD 1983)

U. S. Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815 A, January, 1983.

(Xerox 1985)

Xerox Corporation, *Interlisp-D Reference Manual*, October, 1985.

Experiences Creating a Portable Cedar

Russ Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser

CSL-89-▶◀ ▶March◀, 1989 [▶P87-xxxxx◀]

Abstract: Cedar is the name for both a language and an environment in use in the Computer Science Laboratory at Xerox PARC since 1980. The Cedar language is a superset of Mesa, the major additions being garbage collection and runtime types. Neither the language nor the environment was originally intended to be portable, and for many years ran only on D-machines at PARC and a few other locations in Xerox. We recently re-implemented the language to make it portable across many different architectures. Our strategy was, first, to use machine-dependent C code as an intermediate language, second, to create a language-independent layer known as the Portable Common Runtime, and third, to write a relatively large amount of Cedar-specific runtime code in a subset of Cedar itself. By treating C as an intermediate code we are able to achieve reasonably fast compilation, very good eventual machine code, and all with relatively small programmer effort. Because Cedar is a much richer language than C, there were numerous issues to resolve in performing an efficient translation and in providing reasonable debugging. These strategies will be of use to many other porters of high-level languages who may wish to use C as an assembler language without giving up either ease of debugging or high performance. We present a brief description of the Cedar language, our portability strategy for the compiler and runtime, our manner of making connections to other languages and the Unix operating system, and some measures of the performance of our "Portable Cedar".

CR Categories and Subject Descriptors: D.3.4 **[Programming Languages]:** Processors – Code generation, compilers, run-time environments; D.2.7 **[Software Engineering]:** Distribution and Maintenance – Portability.

Additional Keywords and Phrases: Languages, Portable Cedar

This paper appears in Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon, June 21-23, 1989.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Introduction

Cedar is a large complicated language with many machine dependent constructs. Its original compiler was targeted for a single proprietary architecture, the D-machine [Lampson and Pier]. A large amount of Cedar code is in use (over 2 million lines). All of these constraints seemed to make a portable Cedar language, and a portable Cedar environment, almost impossible. We have a success story to report: it was not that bad. Furthermore, our success story is one with many lessons, both detailed and general, for others attempting to make portable versions of modern languages and environments.

One key lesson is that C [Kernighan and Ritchie] is a feasible portable intermediate language if you treat it as pure intermediate code. People complain that the original C++ implementation [Stroustrup] generates intermediate C which must be actually worked with by people (*e.g.* for debugging); people worry that C intermediate code means inefficient final code (although there have been few measurements to support this, it remains folklore). Our generated C is machine dependent (along a few efficiency-driven dimensions like word length and byte order), very efficient, almost completely unreadable, and almost never seen by humans. We use unmodified Unix* source debugging tools on Cedar-language source. We present measurements that our code is as efficient as directly compiled hand-written C code for both simple (*e.g.* dhrystone) and complicated (*e.g.* page rendering) programs.

A second key lesson is our technique of implementing modern language features in a portable language-independent and operating system-independent layer. Our experience is that a featureful language like Cedar (or, we conjecture, Smalltalk-80† [Goldberg and Robson] or Common LISP [Steele] or Modula-3 [Cardelli, *et al.*], *etc.*) need neither force its language requirements onto the depths of the operating system (as Cedar formerly did and Lisp-machine-style Lisps [Weinreb and Moon] still do), nor develop a thick insulating layer from the operating system (as most Unix LISPs do). Our approach to integrating Cedar into its base system is lightweight, and consists of several layers which together provide the language-independent features like garbage collection, exception handling, runtime types, and threads. We have run identical large Cedar applications on D-machines running the Cedar environment, on Sun SPARC‡ and Motorola 68020 processors running SunOS, on 68020's running Mach [Accetta, *et al.*], and on homebrew imbedded SPARC-based controller boards with no operating system at all.

length: 1.5 in thickness: 0.4 pt

* Unix is a trademark of AT&T Bell Laboratories.

† Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

‡ SPARC is a trademark of Sun Microsystems, Inc.

Related Work

We think of the C language as our target machine language, and can then use any competent C compiler as a platform. Several other translators have used C at various levels to take advantage of its wide availability. Where the source language is reasonably close to C, a preprocessor is sufficient, as the first C++ implementation shows. However, C has also been used for languages whose features differ substantially from C [Yuasa and Hagiya] [Bartlett] [Weiner and Ramakrishnan]. Rather than generate C source, some translators use just the code generation phase of the C compiler to achieve a measure of target machine independence [Feldman] [Kessler]. Languages other than C have been sometimes been chosen as targets in an attempt to be portable [Albrecht, *et al.*].

Mimosa: A Compiler from Cedar to C

The Mimosa compiler translates the Cedar language into machine dependent C. There is a front end, compiling into a simple intermediate form, and a back end, translating from the intermediate form to C code. Another back end generates machine code for the Xerox Dragon processor [McCreight]. In the future, back ends may generate other machine codes, other assembler languages, or other versions of C.

Although treating C as merely an intermediate language has significant advantages, the decision to keep program maintenance in Cedar was made independently on the merits of the Cedar language. We were not willing to do a one-time translation from Cedar to C (even readable C). For example, such things as compiler-generated runtime checks should not be exposed to the programmer for maintenance.

Front end

The front end is a descendant of the Mesa [Xerox] compiler used to generate code for the D-machine. It has been substantially modified over the past three years to be retargetable. We parameterized the front end to cover a large variety of architectures. Word size, addressing granularity, and byte order are the most important of these parameters. Other parameters include floating point format and restrictions on contiguous addresses (if any). While this parameter space does not cover the entire range of commercial machines, we intend it to cover most widely used processors. Currently we support the Motorola 68020 and Sun SPARC processors.

The Cedar language itself exposes details of the target machine, so the parameterization of the front end affects the semantics of the Cedar language. We have adopted a set of guidelines so programmers can keep code portable. As an example, Cedar has a type, INTEGER, that is meant to be used for fast signed arithmetic operations; an INTEGER value occupies one machine word (normally determined by register width). In cases where the machine also supports a wider arithmetic operation, LONG INTEGER is provided, although for most machines the two types are the same. In cases where specific widths are required, for example, to describe externally constrained data, the types INT16 and

INT32 can be used to specify machine integers with specific numbers of representation bits.

The front end has 2M bytes of source in 120 source files, or about 50K lines of code. This does not include a package which manipulates the tree-structured intermediate form shared between front end and back end. The source for that package has 5K lines of code in 12 files. Taking both of these sections as the front end gives about 55K lines of source in 132 files.

Everything in the front end was affected by the retargeting, and some files were completely rewritten. The use of an intermediate code is completely new. The old code generator was changed to produce the intermediate code, in 16 files and over 8K lines of code. A few other files are completely new, for about 30% of the source lines. Since substantial changes have been made to other parts of the compiler, it would be reasonable to estimate that over 50% of the source lines have received major change (not counting the lines that were completely discarded). We estimate that about 2 person-years overall were spent in redoing the front-end.

Back end

The back-end generates machine dependent C code from intermediate form. It was written completely from scratch for the port, consists of 10K lines of Cedar in 24 modules, and took about 6 person-months to write.

We originally chose C so we could start compiler work before we had selected a target architecture. The platforms we want to run on all come with C compilers, many of them with target-specific optimizers. By choosing C as the machine code to be generated, a high degree of portability is achieved. However, we chose to generate efficient C for a specific target machine, rather than generating portable C. Being one step removed from the actual machine makes it harder to control certain details such as layout of local frames, allocation of registers, *etc.*. Parameterization of the back end provides enough knowledge of the target architecture to generate production-quality C. However, the parameterization of the back end for the Motorola and SPARC architectures is identical, since the architectures do not differ in ways that affect the generated C code.

We also chose C because it was lenient enough for our needs. A language with stronger type rules would have hindered us more than it would have helped. There is hardly any type information in our C source, in part because C types and Cedar types are only partly compatible, and in part because we wanted to avoid many of the type coercions that are part of the semantics of C. All of the variables and parameters in our generated C are unsigned words or bytes, and are cast whenever it is necessary to generate typed operations. Since the front end lays out records, arrays and non-local references to frames, the back end generates more addressing arithmetic than is usual in hand-coded C. This leads to C code that is barely readable, but has good performance. By not exposing the intermediate C source to human readers we can transform a program to enhance its performance without caring about loss of readability. For example, we can access a single bit of a packed structure more efficiently if it happens to be a sign bit. Often the code includes constructs that even a C programmer would balk at, though fortunately our C compilers are able to process it.

Only some of the primitive operations of the Cedar language have corresponding operations in C. More complex features, such as nested procedures, are implemented in C with standard compiler techniques. There is no direct way to translate Cedar's signals or lightweight processes into the C language: this discrepancy is resolved by introducing a runtime system. The implementation of such features in C is by procedures and data structures implemented in the runtime system, just as it would be if we were directly compiling Cedar into machine code.

PCR: A Runtime System

The Cedar runtime environment, to which the generated C code is targeted, is written mostly in Cedar, the rest is written in C, except for a small amount of assembler code.

The environment is built in layers. The lowest layer is akin to an operating system, and provides dynamic loading, threads support, and storage management (including garbage collection). This is about 20K lines of C code and less than a 100 lines of assembler (either SPARC and Motorola 68020 at the moment). This layer is not specific to Cedar, and is in fact intended to provide a language-independent base for high level languages. Called the Portable Common Runtime (PCR), it is described elsewhere [Weiser, *et al.*]. The PCR is described below only where its functionality is particularly important to implementing Cedar features.

The next layer provides the lowest level of Cedar-specific support: imports and exports of Cedar interfaces; and a small library of basic utilities like bit moves and typed storage allocation. It is about 5K lines of C. Called CedarPreBasics, it is the last layer of non-Cedar code, and supports the complete Cedar language except for the ATOM, LIST and ROPE data types, and exception handling.

Support for the remaining features of Cedar is provided by the penultimate layer, called CedarCore, which contains 400 lines of C code and 10K lines of Cedar. From CedarCore on, the full Cedar language is supported. The final support layer, BasicCedar, while not necessary for the language itself, contains services that are considered essential for most Cedar applications. For example it includes several kinds of hash table mechanisms and a general-purpose stream I/O package.

Cedar lightweight processes, interface binding, and exception handling are handled by the runtime system. Portable implementations of these features are discussed below. They are representative of the functionality found in each layer of the runtime system.

Threads

The Cedar language includes a primitive operation FORK, which creates a new "lightweight" process (or *thread*) running in the same address space as its parent. The language also has a MONITOR mechanism based on [Hoare], including variables of type CONDITION with WAIT, NOTIFY and BROADCAST primitives to provide synchronization between processes. An ABORT operation can be used to wake up a process that may be waiting on an unspecified condition.

The D-machine Cedar implementation had microcode support for processes, but such support is not essential. Efficient threads implementations on conventional hardware already exist as part of Mach [Accetta et.al.]. In addition, Unix-based threads packages of varying degrees of sophistication are becoming widely available [Cooper] [Kepecs] [Doeppner]. It is important to note, however, that Cedar is quite demanding in this area if the excellent "feel" of the D-machine implementation is to be retained. It would not be acceptable if a compute-bound thread could seize the processor, or if execution of an I/O operation by a single thread caused all other threads to block. Thus, a simple coroutine threads package, which might be adequate in a simulation environment, would not meet our needs. The PCR threads package provides the features we require in a reasonably portable way by using the signal handling, shared memory and asynchronous I/O features available in advanced Unix systems. Some advanced debugging facilities, such as the ability to freeze, examine and thaw individual threads under program control, will be needed for eventual implementation of a full debugger.

Interface binding

The CedarPreBasics layer implements the loadstate: the dynamically constructed mapping from interface items to their implementations. The loadstate implementation is responsible for the final steps in static type checking which insures that dynamically loaded modules mesh correctly with the already-existing types in the system.

An interesting aspect of the loadstate implementation was the method we chose to convey type, import, and export information from the Mimosa compiler to the loadstate. The D-machine Cedar compiler incorporates parts of its symbol tables in the files containing executable code. The D-machine loadstate implementation shares knowledge of these data structures with the compiler, and can interpret them to build the loadstate. In making Cedar portable we realized that issues of byte order, cross compilation, and performance strongly argue against sharing symbol table structures. Instead, the necessary information is conveyed in executable code, encapsulated in an installation procedure for each module. When a module is dynamically loaded into a running Cedar world its installation procedure is called, calling in turn upon the loadstate to type check and bind imported and exported interfaces.

Exception handling

Cedar's signal mechanism lets programmers write uncluttered code for normal cases, isolating the code for exceptional cases in catch phrases. Since signals propagate over procedure call boundaries, the exceptional cases can be handled where it makes most sense to do so. Enable scopes are either blocks or single procedure applications. Every catch phrase ends by disposing of the signal in one of three ways: it is rejected, forcing catch phrases in ancestral procedure frames to examine and dispose of the signal; it is resumed, supplying a value for the original signal application; or it is terminated.

Termination forces the procedure call stack to be unwound to the procedure invocation where the catch phrase was established, which then continues execution with variable values as of the time that the locus of control last left that frame.

Our implementation of Cedar's signal semantics uses a per-thread catch stack to record active catch phrases. Entering and exiting enable scopes must be cheap because they are part of the execution path in frequent cases. Raising signals and processing them according to the catch stack can be more expensive because it happens relatively infrequently. Entering an enable scope requires pushing several words of data onto the catch stack, *i.e.* about the cost of a procedure call. Raising a signal requires traversing the catch stack, invoking each catch phrase in turn and processing its result: reject, resume or terminate. The catch stack interpretation is itself Cedar code and uses Cedar signals.

The hardest problem in this area was the correct implementation of the termination case of a catch phrase. We use essentially the C library setjmp and longjmp procedures, but for Cedar we must ensure that local variables are restored to their state at the time of the last call out of the frame. In the case of a register machine, this means restoring the registers to the values they had when the stack frame was last left (not necessarily their values at setjmp time). On the SPARC processor or the VAX these semantics are provided by the setjmp and longjmp in the standard library. For the 68020, a considerably more complicated mechanism is required. Our implementation of longjmp for the 68020 walks up the frame stack, restoring registers by interpreting the procedure entry code for each active procedure to determine the registers saved there. For some language implementations on some processors this becomes much more difficult; the C implementation for the DEC WRL Titan processor, for instance, saves and restores registers at arbitrary points in the program text, making a principled register-restoring stack walk difficult [Powell].

One of the noticeable features is the system level approach. The compiler front end notes the scope of each catch phrase and compiles each catch phrase into a separate procedure. The compiler back end generates code to enter and exit the enable scopes at the appropriate points. Exiting enable scopes is complicated by certain styles of exits from blocks. The runtime system then implements the signal handling mechanism. Three pieces of the system share responsibility for this language feature.

Building on Unix and C

One of our major goals in making Cedar run on commercial hardware was to take advantage of software developed in the larger computing community. Further, we wished to begin making Cedar's superior facilities for building large systems available in the Unix environment. To this end several tools have been developed and the Cedar language has been extended. Intercalling between Cedar and C programs is provided. First, Cedar programs can call arbitrary C entry points using a new variant of the MACHINE CODE construct in the Cedar language. Of course, such uses are inherently unsafe, so their use is restricted. Second, tools are available to generate the necessary calls on the loadstate to import Cedar procedures and variables into C programs. Finally, tools to describe Cedar data structures in C and vice versa have been written.

Using these facilities, we have described a substantial portion of the Unix system call interface in Cedar interfaces, including file and socket I/O. Cedar programs use Cedar interfaces for type-checked access to C procedures. Applications of this technique include an X window client [Scheifler and Gettys] and a SunRPC-based Mimosa compilation server.

Performance and Practice

Porting Effort

One of the advantages of retargeting an existing translator is that you have on hand a large body of code to translate. We quickly gained considerable experience with porting code from the D-machine version of Cedar. We have also gained experience with writing code such that it runs unmodified on several machine architectures.

In excess of 365K lines of Cedar code have been ported in the year that the compiler has been available. These packages range in size from small (1K lines for an arbitrary-precision number package) to huge (the compiler itself is 50K lines). A measure of our success in using the same source for both architectures is that only 12K lines of architecture-specific code has been created for the port so far. The modular structure of the Cedar language has allowed us to hide those architecture-specific implementations behind interfaces that exist in both worlds. The layered structure of Cedar encourages us to believe that most of the architecture-specific implementations that are needed for applications have already been written.

The first reasonably large program to be ported was a Scheme [Rees and Clinger] interpreter: 14 files with 9K lines of Cedar. This occurred in January, 1988, and took one person approximately a month (including discovering many compiler and runtime bugs). The next large program was the Cedar Imager [Swinehart, *et al.*], soon followed by an interpreter for the Interpress page description language, together consisting of 60 modules and 40K lines of code. It took one person about three elapsed months. This code was older and more tuned for efficiency, and so uncovered more compiler and runtime bugs. The compiler itself was ported some months after that and took about 6 elapsed weeks.

Some programs are more easily ported than others. In general, Cedar programs that do not exploit knowledge of the D-machine (e.g. word size, addressing granularity, or other hardware features) can just be compiled with the new compiler and run on C platforms. In some cases, data structures can be preserved across architectures by the use of types with specific representations, though possibly at some cost in execution speed.

Performance of generated code.

In practice, the performance of code generated by Mimosa for a given task is comparable to the performance of hand-written C code for the same task. In some cases the hand-written C code is slightly better, although this gap continues to narrow as the compiler gains maturity.

In one case, Dhrystone II [Weicker], the Mimosa-generated C code for the Cedar version actually runs faster than the hand-written C code. The reason is that Cedar strings are word-aligned and length-containing, while C strings are zero terminated. Faster routines for string comparison and string copying, gave the Cedar code a significant advantage for this program. This example illustrates one of the dangers of performing naive comparisons between compilers and between languages.

For good display and printing performance using bitmap graphics it is important to have fast bit block moving operations. One particular routine, written in Cedar, performs a bitwise OR of a block into unaligned memory. This routine, running on a Sun-4/260, transfers blocks that are 20 bits wide by 20 bits high at over 34,000 blocks per second. A careful examination of the C code produced by the Mimosa compiler revealed no significant room for improvement.

We do not pay an execution penalty for writing in Cedar and then translating to C code. This result is in agreement with similar results for other languages [Bartlett]. Since we can exploit optimizing C compilers, we can get good performance without our having to invest in the expertise needed to generate production quality machine code for our current and future platforms.

Debugging

We think of the C language as the assembler language of our target architecture, but we are not willing to debug using only assembler level debugging tools. We use the dbx debugger provided with most of our platforms to access the implementation details of the generated C code [Linton]. Dbx supports several languages, but in all those translators debugging information is provided solely by the front end, and is passed unchanged through the assembler. Since we are using the C compiler as our assembler, providing debugging information is a little more complex. Some of the information needed for debugging is known only to our front-end, *e.g.* decisions about parameter passing, record layout, *etc.*, and all type information. Some of the information needed for debugging is known only to the C compiler, *e.g.*, frame layouts, register allocations, addresses of procedures, *etc.* These two sources of information are merged during a post-processing step by replacing the name, type, and location information from the C compiler with the corresponding information from the Cedar front-end. Dbx then sets and reports breakpoints by reference to Cedar source and uses the Cedar names for the corresponding C variables and procedures. Since we were not willing to modify dbx, we make do with the type system permitted by dbx's interpreter and dbx's pretty-printing of data structures.

Since Cedar, and in fact the PCR in general, supports dynamic loading, we have worked out a scheme for using dbx on dynamically constructed load images. As each module is dynamically loaded the runtime system dribbles symbol table and relocation information to a log file. When the debugger is invoked, the log file is used to construct a synthetic a.out file, with only a symbol table, representing the state of the currently loaded modules. The synthetic a.out can be used to debug the dynamically constructed process.

A better job of supporting Cedar (and inter-language) debugging is awaiting the construction of something like the Cedar Abstract Machine [Swinehart, *et al.*]. We think we can impose such a system on a C platform in the same way we have handled dynamic loading in the PCR: any data structures we need can be constructed at load time by executing code in the module being loaded. This technique frees us from all but the very lowest levels of interactions with the target platform and allows us to be independent of the debuggers provided on those platforms.

Conclusions and Recommendations

We have taken a featureful language designed to be executed on a proprietary architecture and made it portable by having its compiler generate C. We have taken a large body of code written in that language and ported it to industry-standard platforms using the new compiler. We have achieved excellent efficiency – as good as hand-coded C – and we have gained leverage from work already done for other languages and other systems. We have learned several important lessons along the way, including how to use C as an assembler language, how to use C debuggers for debugging Cedar source, and how language-independent layers can support what are ordinarily language-dependent runtime features like threads and garbage collection. The techniques we have developed will be of growing importance as the computing world is based increasingly on interoperability and the use of existing tools.

Acknowledgments

We thank the following people who helped us build and use the Mimosa compiler and the language tools that support it: Jean-Christophe Cuenod, Jim Foote, Linda Howe, Bill Jackson, Christian LeCocq, Andy Litman, Eduardo Pelegrí-Llopart, Bryan Lyles, Michael Plass, and all our other colleagues who have been willing to help us find the bugs in the system. We also thank the program committee members who gave us feedback on the extended abstract.

References

Accetta, J.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F. Tevanian, A., and Young, M.W., "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of Summer Usenix*, July, 1986.

Albrecht, P., Garrison, P., Graham, S., Hyerle, R., Ip, P., Krieg-Brückner, B., "Source-To-Source Translation: Ada to Pascal and Pascal to Ada", *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, SIGPLAN Notices*, **15**, *11*, November, 1980.

Bartlett, J., "SCHEME->C a Portable Scheme-to-C Compiler", Research Report 89/1, DEC Western Research Laboratory, January, 1989.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G., "Modula-3 Report", DEC Systems Research Center, August, 1988.

Cooper, E. and Draves, R., "C Threads", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, March, 1987.

Doeppner, T.W. Jr., "Thread Calls", unpublished manuscript, Brown University, 1987.

Feldman, S.I., "Implementation of a Portable Fortran 77 Compiler Using Modern Tools", *Proceedings of the ACM SIGPLAN 1979 Symposium on Compiler Construction, SIGPLAN Notices*, **14**, *8*, August, 1979, pp. 98-106.

Goldberg, A. and Robson, D., "Smalltalk-80: the language and its implementation", Addison-Wesley, 1983.

Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", *CACM*, **17**, *10*, October, 1974.

Kepecs, J.H., "Lightweight Processes for UNIX Implementation and Applications", *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall, 1978.

Kessler, P.B., "The Intermediate Representation of the Portable C Compiler, as Used by the Berkeley Pascal Compiler", Unpublished manuscript, Computer Science Division, EECS, University of California, Berkeley, CA, April, 1983.

Lampson, B. and Pier, K., "A Processor for High-Performance Personal Computer", *SIGARCH/IEEE Proceedings of the 7th Symposium on Computer Architecture*, La Baule, May, 1980, pp. 146-160.

Linton, M.A., "dbx", *Berkeley UNIX User's Manual*, Computer Science Division, EECS, U. C. Berkeley, California, April, 1986.

McCreight, E., "The Dragon Processor", *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, October, 1987, pp. 65-69.

Powell, M. private communication.

Rees, J., and Clinger, W. (Eds.), "Revised^3 Report on the Algorithmic Language Scheme", *SIGPLAN Notices*, **21**, *12*, December, 1986.

Scheifler, R. and Gettys, J., "The X Window System", *ACM Transactions on Graphics*, **5**, *2*, April 1986, pp. 79-109.

Steele, G., *Common LISP: The Language*, Digital Press, 1984.

Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, 1986.

Swinehart, D., Zellweger, P., Beach, R., Hagmann, R., "A Structural View of the Cedar Programming Environment", *TOPLAS*, **8**, *4*, October, 1986.

Weicker, R., "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules", *SIGPLAN*

Notices, 23, 8, August 1988.

Weiner, J.L. and Ramakrishnan, S., "A Piggy-Back Compiler for Prolog", *Proceedings of the SIGPLAN '88 Conference on Programming Design and Implementation, SIGPLAN Notices*, 23, 7, June, 1988.

Weinreb, D. and Moon, D., *Lisp Machine Manual*, MIT AI Lab, 1981.

Weiser, M., Demers, A., and Hauser, C., "The Portable Common Runtime Approach to Interoperability", submitted to *Twelfth ACM Symposium on Operating Systems Principles.*

Yuasa, T. and Hagiya, M. *Kyoto Common Lisp Report*, Research Institute for Mathematical Sciences, Kyoto University, no date.

Xerox Corporation, Mesa Language Manual, XDE3.0-3001, Version 3.0, November, 1984

Appendix – Code Examples

Here is a small example Cedar program with one global variable, a global procedure, some local variables, and a nested procedure that makes several non-local variable accesses. This program is not a typical Cedar program.

```
Test: CEDAR PROGRAM = {
  global: INT;
  Outer: PROC [] = {
    local: INT;
    a: PACKED ARRAY [0..31] OF [0..15];
    Nested: PROC [index: [0..31]] = {
      local ← global + a[index];
      };
    Nested[index: 5];
    };
  }.
```

Below is the C code which is generated for the example program. It begins with type declarations which declare everything to be unsigned integers or characters or types constructed from those basic types. Symbols from the Cedar code have had unique suffixes added to to them, so that similar names don't interfere in the C code (which has much more restrictive name scopes).

The translation of the outer procedure is straightforward, since all it does is set up a procedure descriptor for the nested procedure and call it with the appropriate actual parameters. Procedure descriptors are used for nested procedures, as well as for procedure variables and procedure parameters.

The first two lines of the nested procedure set up its addressing environment: a pointer to the global frame and a static link derived from the procedure descriptor passed as its last argument. The remaining line is the translation of the body of the nested procedure. The non-local references are fairly self-explanatory. The first use of the index variable selects a byte offset within the array, and the second use of the index variable selects either a 0 bit or a 4 bit shift of the selected byte.

```
typedef unsigned word, *ptr;
typedef unsigned char byte, *bPt;
typedef struct {word f0, f1, f2, f3, f4, f5, f6, f7} W8;
typedef word (*fPt)();
typedef struct {word f0, f1, f2} W3;
typedef struct {W8 f; W3 r} W11;
static void Nested__60();

static void Outer__30() {
        W11 var__1406;
        (* (( (ptr) &var__1406)+4) ) = ( ((word) (fPt) Nested__60) );
        (* (( (ptr) &var__1406)+5) ) = 1;
        (void) Nested__60(5, (word) (( (bPt) &var__1406)+16)/* var__1390 */);
};
static void Nested__60(index__1330, formal__1438)
        word index__1330;
        word formal__1438;
{
        register ptr gf__1422 = (ptr) &globalframe;
        formal__1438 = (formal__1438 - 16);
        (* (( (ptr) formal__1438)+6) ) = ((* (( (ptr) gf__1422)+4)/* global__1190 */ ) + (((*
         (bPt) ((( (bPt) formal__1438)+28) + (index__1330 >> 1)) ) >> (4 - (((index__1330 &
         1) << 2)))) & 017));
};
```

Calling C procedures from Cedar

Trusted Machine Code Procedures

Usage of a prefix and underscore is recomended: otherwise the name could crash with a compiler generated name.
Also dont use names with exactly one underscore followed by a single letter and then digits: the compiler generates those too.
The ← character in Cedar is an underscore.

Warning: not all features are ratificated by the language design commitee: some features might change, but it seems unlikely.

It might be good to stick to the "Simple", the "Complex" and the "ExternalNames" example. The "Crazy" examples and the "More complete syntax" contain features of questionable value.

Simple examples

1) Include just a plain C procedure

```
PutChar: PROC [ch: CHAR] = TRUSTED MACHINE CODE {
    "Mumble"
    };
```

generates

```
Mumble(ch)
```

However, normally you have to get a declaration of the procedure you are calling, so this case might be a little bit oversimplified.

2) Include a C procedure with an include file from the standard Cedar place

```
PutChar: PROC [ch: CHAR] = TRUSTED MACHINE CODE {
    "Foo.Mumble"
    };
```

generates

```
#include <cedar/Foo.h>
...
Mumble(ch)
```

This case is made to look like in Cedar external procedure: e.g. the extension .h for the include file need not be specified.

3) Include a C procedure with an include file from the standard Unix place

```
PutChar: PROC [ch: CHAR] = TRUSTED MACHINE CODE {
    "<Foo.h>.Mumble"
    };
```

generates

```
    #include <Foo.h>
    . . .
    Mumble(ch)
```

Here we are thinking Unix: specify a unix h file including the .h.

4) Include a C procedure with an include file from the <u>working directory</u>

```
PutChar: PROC [ch: CHAR] = TRUSTED MACHINE CODE {
    """Foo.h"".Mumble"
    };
```

generates

```
    #include "Foo.h"
    . . .
    Mumble(ch)
```

Here we are thinking Unix: specify a unix h file including the .h.

Complex examples

```
DRealAddI: UNSAFE PROC [ret, x, y: PDREAL] = UNCHECKED MACHINE CODE {
    "+extern void XR←DRealAddI (ret, x, y) W2 *ret, *x, *y; {\n";
    "   DRealPtr(ret) = DRealPtr(x) + DRealPtr(y);\n";
    "   };\n";
    ".XR←DRealAddI";
    };
```

Note that machine code procedures can use multiple lines.
The "+" denotes that the following stuff is included after the type declarations of the module.
In this exampole DRealPtr needs to be definerd:

```
DefDRealPtr: PROC = TRUSTED MACHINE CODE {
    "+ #define DRealPtr(x) (*((double *) (x)))\n."
    };
```

Here we note that there is no actual procedure to be called: the machine code text has nothing following the period. This machine code procedure was thought to be called in the module initialization to provide common definitions.

Crazy examples

Using assignments and declarations

Avoid usage of this feature for assignments and declarations; it is quite baroque

E.g. for the Unix errno feature [Its brain damaged to use global variables, but Unix users have no choice]. BTW: it is better to use the runtime feature than to try to do this with machine code procedures. The runtime feature is much more correct. E.g, the runtime feature knows how to deal with multiple threads, whereas this method does **not**.

```
ErrNo: PROC [] RETURNS [INT] = TRUSTED MACHINE CODE {
   --call a variable, and make sure it is declared (extern)
   "!$errno"
   };

ClearErr: PROC [] = TRUSTED MACHINE CODE {
   --include this statement as is
   "@errno = 0"
   };

EvenMoreUgly: PROC [] RETURNS [INT] = TRUSTED MACHINE CODE {
   --make a declaration as is. e.g. to fool C2C's types
   --and, make a call to this variable
   "+extern int evenMoreUgly:.$evenMoreUgly"
   };

P: PROC [] = {
   i, k: INT;
   ClearErr[];
   i ← ErrNo[];
   k ← EvenMoreUgly[];
   };
```

generates
```
   extern word errno;
   extern int evenMoreUgly;
   ...
   errno = 0;
   i←543435 = (word) errno;
```

More complete syntax

You wouldn't believe how archaic machine code procedures are parsed. It is better to stick to the well supported examples than to understand the complete parsing algorithm. When new features need to be included I intend to keep the simple examples correct, but can't make any guarantees about the general algorithm.

First, split the text into the piece for the declarations [prefix part], and, the piece to be included in line [procedure name part]; this is done at the position of the rightmost dot. To the left of the dot is the prefix part, to the right of the dot is the procedure name part.

prefix part syntax:

We introduce the *entry*; thats the unit used in caching; Each entry is cached and handled only once; this is used to get only single includes. Users must be carefull and use exactly the same spelling if two machine code procedures have same entries.
consume leading %: it says that each line is a separate entry
other leading letter: this starts the single entry for the whole prefix part

entry:
entries are cached: the same entry is included only once into a C file; An entry may be multiple lines; separated by either /n or actual lines in the source.
leading ": makes an #include "....
leading <: makes an #include <....
leading *: include the rest of the entry before the type declarations
leading +: include the rest of the entry after the type declarations
leading ~: include .h file from standard Cedar place
leading =: include .h file from standard Xr place
alphabetic leading letter: think cedar module name; adds include .h file from standard Cedar place.
"adds include .h file" means: Rope.Cat["<standard-place/", entry, ".h>"]

procedure name part syntax:
leading !: make line to declare the name [extern]
leading &: don't use (void), even if procedure has no return parameters
leading $: no paranthesis, e.g. for constants, variables
leading @: no paranthesis, don't use (void) e.g. for constants, variables
leading :: stop consuming further leading letters [so reserved characters can be used in the name]
alphabetic leading letter: this starts the name

Calling Cedar procedures from C

The "problem"

1) Make the generated C procedure have a C name under programmer control.
2) Make sure the "signature" calling sequence of the Cedar procedure matches the C procedure.

There are currently two methods to force programmer defined procedure names.
a) Use TRUSTED MACHINE CODE to specify inside the program what the names should be
b) Use an external file to define the names of the procedures.

As of today method b) has fallen out of my favor. Avoid this because I might retract it when I can make sure all uses are fixed.

Warning
This mechanism makes Cedar procedures available for extern use without enforcing the module initialization to run first. However, correct behaviour of Cedar procedures may depend on doing the module initializations.

a) Name definition using TRUSTED MACHINE CODE procedures

Example:

```
ExternalNames: PROC [] = TRUSTED MACHINE CODE {
   "↑ExternalNames\n";
   "Xyz      XR←Xyz\n";
   "Foo      XR←Foo\n";
   };
```

In this example, the trusted machine code procedure ExternalNames must be used, so C2C will actually see it in the code tree.
The line "↑ExternalNames\n"; specifies that this machine code procedure has the purpose of defining C names.
This makes the Cedar procedure Xyz have the C name XR←Xyz; as well as the Cedar procedure Foo have the C name XR←Foo.

It is required [but not tested] that all CedarProcedureName's are existing top level procedures.
It is an error if multiple procedures [even if not top level] with name CedarProcedureName exist.

Design rational

The external name can be explicitely specified [and is not simply a translation of the Mesa name]. This allows to make "exportable" names with prefix and underscore.

b) Name definition using external file

Avoid this because I might retract it. The problem is, that usage of an external file needs a special switch and knowledge by several programming environment tools.

Uses -R switch to mimosa to make exteRnal procedure declarations.
This causes reading in a modulename.externalProcs file describing external names of procedures. Only the procedures described in .externalProcs file will get external names.

Syntax of .externalProcs files

File is line oriented
Lines starting with "-" are comments
The first non comment line must be
  extern procs
All other non comment lines have the form
  CedarProcedureName CProcedureName
The CedarProcedureName must have the syntax of a Cedar name.
The CProcedureName must have the syntax of a C name.

Syntactical Restrictions

It is required [but not tested] that all CedarProcedureName are existing top level procedures.
It is an error if multiple procedures [even if not top level] with name CedarProcedureName exist.
Module name is used instead of file name to find .externalProcs file. [As long as C2C

does not know file names]

Questionable

Are the names of features part of the mesa program or not? Logically they might not have anything to do with the mesa program, but then why did we care in first place.

Comparison with Custer [see Custer's own doc]

Custer allows a C program to simulate the Cedarish import mechanism; it requires the C program to be specialy written for this purpose. Custer has the advantage of full Cedarish re-importing possibilities.

This simply exports Cedar procedures the C-ish way; no version checking or reloading is possile. Its advantages are less work and, it works for unmodified C programs not knowing that they call Cedar procedures. By not requiring to modify the caller, this mechanism may also be used from Languages other than C.

Other Interoperability Hints

See the modules UXProcs, UXStrings.

UXProcs contains procedures to transfer between Mesa procedure values and C procedure values. This module hides the fact that procedure values in Mesa have one level of indirection more then procedure values in C.

UXStrings contains procedures to transfer between Cedar ropes and Unix strings. The problems are the immutability requirements of Cedar ropes and the null termination of unix strings.

Salient
Mesa and C Intercallability

Working draft–not approved by any working group

Author:	Litman
Date:	1/22/88
Filed:	(K9:OSBU North:Xerox)/SalientDesignWorking/Intercallability (K9:OSBU North:Xerox)/SalientDesignWorking/Intercallability.ip

Revision history:

Approvals:

David Elliott
Manager, Advanced Software Development

Linda Bergsteinsson
Manager, Workstation Applications

Ron Boyd
Manager, Network Services & Communications

XEROX

DOCUMENT SYSTEMS BUSINESS UNIT
Workstations and Network Systems
Advanced Software Development
Sunnyvale El Segundo

Intercallability

1 Introduction

This document describes intercallability between Mesa and C, as supported by Salient language tools. Paradigms for software design are proposed which guarantee as much type safety as possible.

Readers are assumed to be familiar with:
    Mesa
    C
    basic workings of Mesa to C translation
    basic C development tools (cc, lint, ld)

2 Mesa calling C

2.1 MACHINE CODE

Support for Mesa calling C is provided by a new form of MACHINE CODE procedure. If a string literal appears in a MACHINE CODE procedure, it is converted by Mimosa into a C procedure call. For instance,

```
Add: PROC[a,b: CARDINAL] RETURNS[CARDINAL] =
   MACHINE CODE { "cadd" };
Add[1,2];
```

appears in the C output after compilation as

```
cadd(1,2);
```

Such MACHINE CODE procedures may appear in either interfaces or programs. Arguments passed to a MACHINE CODE procedure are Mesa expressions, not C expressions.

In the above example, while the call to Add is typechecked by the Mesa compiler, the call to cadd is not type checked by the C compiler. If cadd expected three arguments rather than two, an error would occur at runtime. We can take advantage of the type checking that C provides by defining cadd in a header file, and including the header file in the C output. This is done by specifying the header file in the MACHINE CODE procedure. Say the header file is called header.h. Then

```
Add: PROC[a,b: CARDINAL] RETURNS[CARDINAL] =
   MACHINE CODE { "<header.h>.cadd" };
Add[1, 2];
```

appears in the C output after compilation as

```
#include <header.h>
cadd(1, 2);
```

In the C output, the include line appears before the procedure containing the call. The header file may also be included as follows:

```
Add: PROC[a,b: CARDINAL] RETURNS[CARDINAL] =
   MACHINE CODE { "\"header.h\".cadd" };
Add[1, 2];
```

appears in the C output after compilation as

```
#include "header.h"
cadd(1, 2);
```

To do: talk about C procedures as first class objects.

2.2 Transducers

To implement a Mesa interface procedure using calls to UNIX or other C libraries, arguments to the procedure must be converted into C variables and passed to the desired library routines. We call the code that does this a *transducer*. Converting Mesa variables to C variables is described in [Yamamoto].

A transducer may be long and complicated. If one were implementing NSFile by calls into the UNIX file sytem, for instance, converting the Mesa arguments into C variables would be tricky, and several calls into UNIX might be needed for each NSFile procedure.

A transducer may be simple. If one were implementing a procedure

```
Alloc: PROC [nwords: INT] RETURNS [LONG POINTER];
```

in the obvious way, the transducer could merely multiply nwords by two to get the number of bytes and pass that to malloc, a C library routine.

A transducer may not be needed at all if the parameters of the Mesa interface procedure are compatible with the parameters of the C procedure.

2.3 Sherman

Sherman is a program which translates a Mesa symbol table into a C header file. The header file may be used to typecheck C files by defining C "equivalents" of Mesa types. By equivalent we mean that the bit layout of the C and Mesa types is the same. Some attempt is made to preserve symbol names for readability, but symbol names are modified or created where necessary to make a valid header file.

There are cases where Sherman fails. These are listed in ShermanRestrictions.doc. Because Sherman may make mistakes, we consider the typechecking enabled by Sherman to be weaker than Mesa interface typechecking or C header typechecking.

Sherman is entirely optional in the intercallability schemes described below. If Sherman is not used, C headers may be written by hand.

2.4 Schemes

There are two recommended schemes for Mesa calling C, depending on whether the transducer is written in C or Mesa. The choice should be based on the programmer's personal preference for Mesa or C, and any other advantages the programmer perceives.

To do: talk about Lint

2.4.1 C Transducer

Figure 1 shows the scheme for a C transducer. The application is some client whose source code we'd rather not modify. The interface might be, say, a public BWS interface. Each procedure in the interface has been changed to a MACHINE CODE procedure that contains the name of a C routine implemented in transducer.c. Interface.h was produced automatically by Sherman from Interface.bcd. UNIX.h represents either a UNIX header, some other C library header, or a header which the programmer has created.

The major work in this scheme is writing transducer.c. Adding the MACHINE CODE to Interface.mesa can be done in one step using the XDE editor.

2.4.2 Mesa Transducer

Figure 2 shows the scheme for a Mesa transducer. Again, the application is some client whose source code we'd rather not modify. In this scheme, the interface is not modified either. The interface procedures are implemented in transducer.mesa. A bridge interface containing the MACHINE CODE procedures has been written by hand.

The major work in this scheme is writing transducer.mesa, but writing Bridge.mesa requires work as well. If the programmer is calling into UNIX or an existing C library, Bridge.mesa is created manually by translating the procedure types from

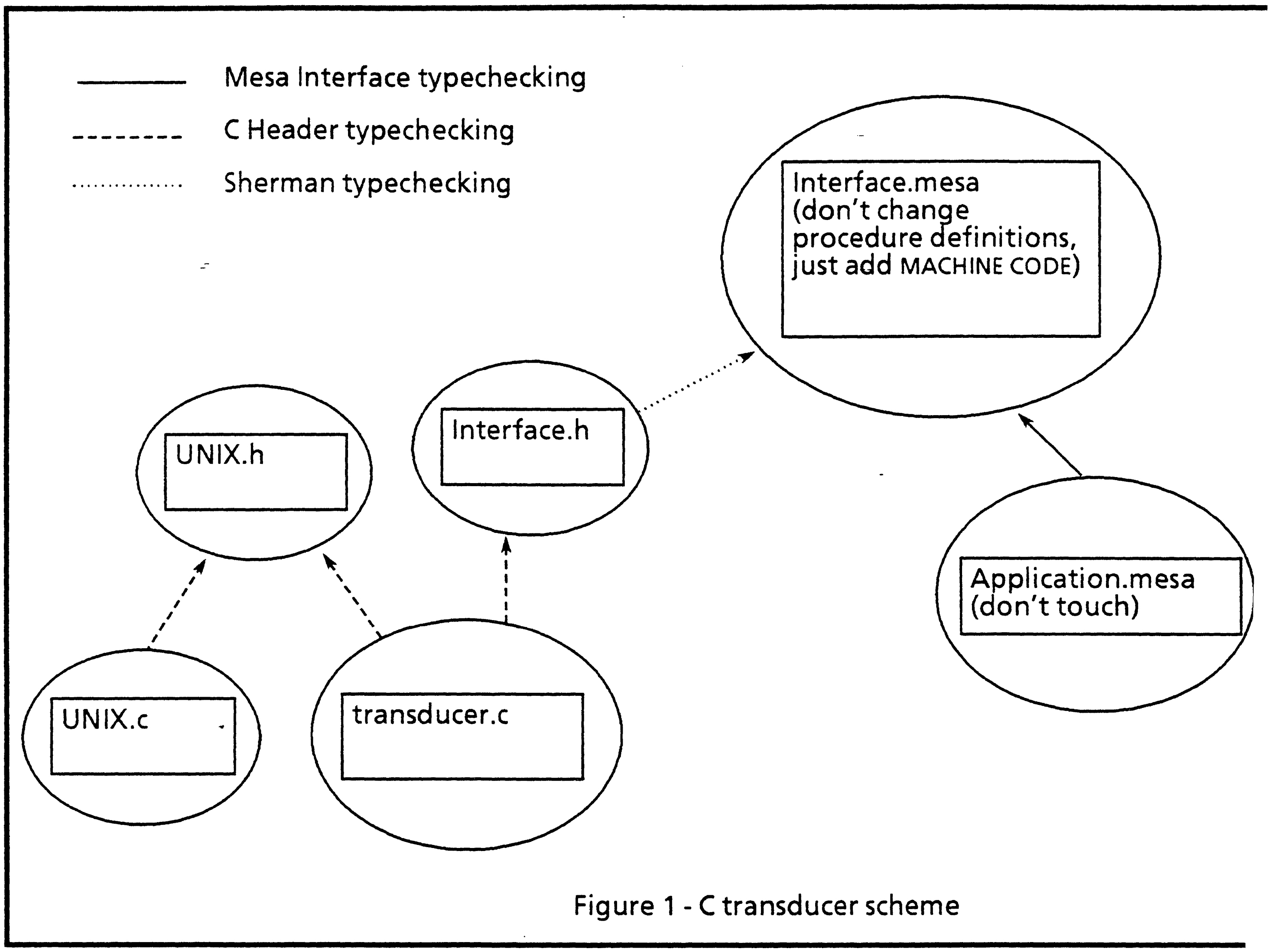

Mesa Interface typechecking
C Header typechecking
Sherman typechecking
Interface.mesa (don't change procedure definitions, just add MACHINE CODE)
UNIX.h
Interface.h
Application.mesa (don't touch)
UNIX.c
transducer.c

Figure 1 - C transducer scheme

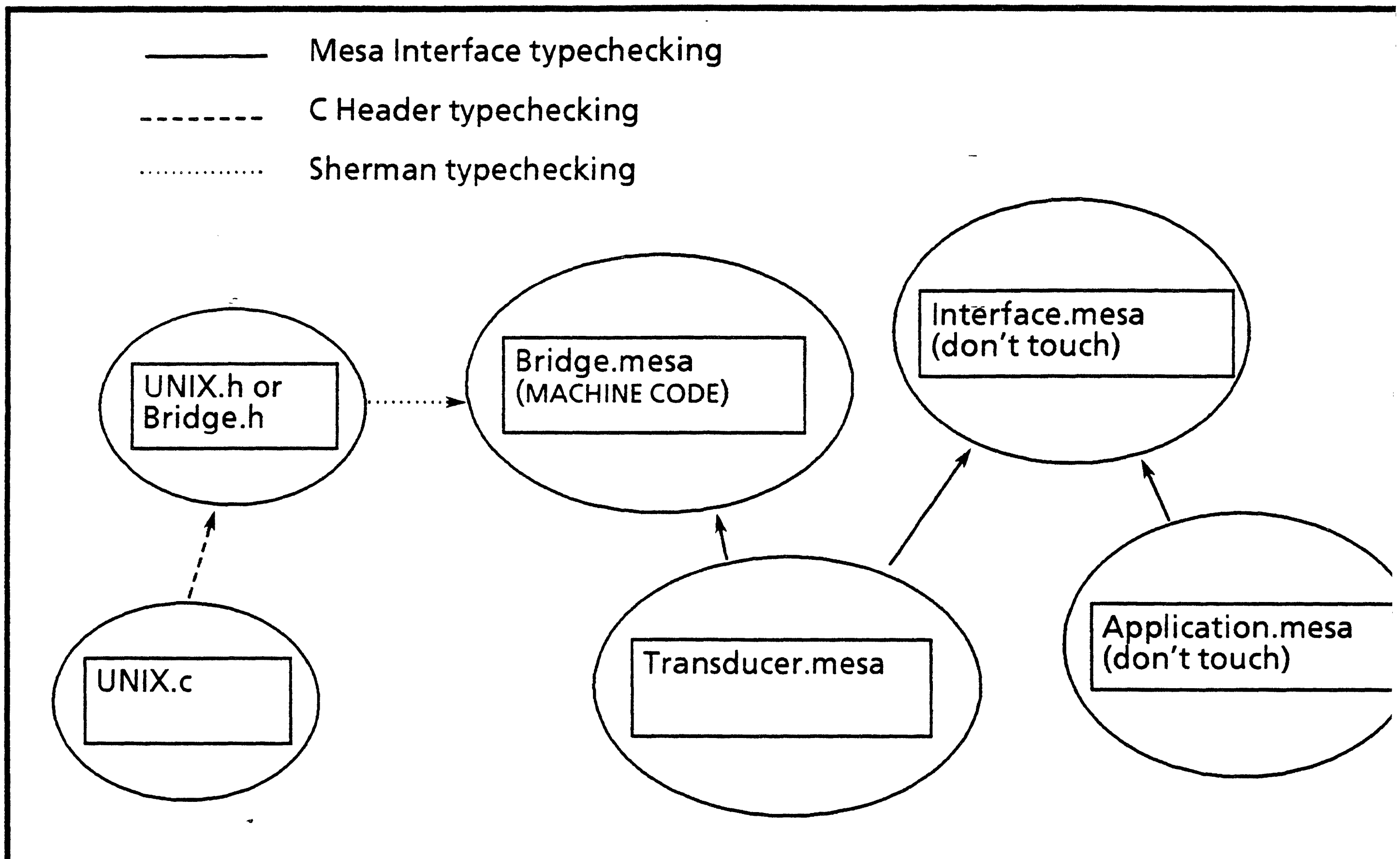
UNIX.h and UNIX.c into equivalent Mesa types. This hand translation is **not typechecked**, and so must be done carefully. It may be helpful to use Sherman to convert the Bridge back into a header file, and check that it matches the original.

If the programmer is calling into a C library they have just written, then Bridge.h can be created automatically using Sherman. In this case we have the Sherman typechecking shown in the figure.

3 C calling Mesa

Support for C calling Mesa has not been designed yet. It is believed that C calling Mesa will not be as common as Mesa calling C, and so it has lower priority.

Converting C variables to Mesa is presumed to be easier than converting Mesa variables to C, since the Mesa type system



Mesa Interface typechecking
C Header typechecking
Sherman typechecking
UNIX.h or Bridge.h
Bridge.mesa (MACHINE CODE)
Interface.mesa (don't touch)
UNIX.c
Transducer.mesa
Application.mesa (don't touch)

Figure 2 - Mesa transducer scheme

subsumes C. Hence the transducers may be simpler for C calling Mesa.

In order to call a public Mesa procedure, a C program must access the procedure through an interface, in the same way that another Mesa program would. An interface with the correct version can be found by calling into the Mesa runtime. (We may provide runtime routines which relax the version checking.)

It is possible to write the runtime calls by hand, although this is tedious. Here are two vague ideas for automatically generating the runtime calls:

> The programmer puts a special construct into the C source code whereever there is a Mesa call. Then either a preprocessor or a postprocessor creates and inserts the runtime calls.
>
> Develop a tool similar to Sherman. It reads a Mesa interface and produces a transducer containing the appropriate runtime calls to access the interface.

4 References

Yamamoto, "Language Interoperability," [Goofy:OSBU North:Xerox]<PortDesignWorking>Language. Interoperability.ip. This needs to be rewritten for Salient.

Using Cinder

XEROX Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

For Internal Xerox Use Only

Introduction

Cinder is a revision of the Mesa Binder for the Portable Cedar project. The idea is that to get the bindings we desire in a C environment we replace the work done by the Binder and the Loader with a set of runtime calls. Cinder takes a C/Mesa file and produces a C module containing the set of runtime calls to InstallationSupport, which is part of the Portable Cedar library. The C module can then be compiled and linked with the component modules of the configuration, which are translated into C by Mimosa. Cinder produces a Mesa object file (with extension *.mob*) which represents the configuration, and is used as input for further Cindering. Cinder also produces a UNIX script (with extension *.ld*) which can be run on a UNIX machine to produce an executable file.

Cinder does version checking on the components of the configuration. The Mesa runtime also does version checking at load time, but most users prefer to be told about version mismatches at Cinder time. Version checking can be disabled with a switch (see below).

This document does not describe the C/Mesa language that is used to specify the production of system configurations from independent modules. [See the C/Mesa section (section 7.9) of the Mesa Language Manual for that.] However, additions to the C/Mesa language which facilitate multiple language configurations and loading on demand is described in this document.

Installing Cinder

BringOver the public part of Cinder.DF from the Mimosa directory, into a directory that will be on your search path when you want to use it -- preferably to a Commands subdirectory [See CompilerDoc].

Invoking Cinder

Like the Compiler, there ought to be a simple Cinder command that allows full path names and the "slash"-style format in file names, and a ComplexCinder command, providing a wider range of configuration options but restricting filename specifications (due to conflicts in the syntax).

At present, the one and only Cinder command has the "ComplexCinder" syntax, but it includes a simple option which most people use most of the time.

[see CreateButtonDoc] for ways to produce Command tool buttons for invoking Cinder.

The "Cinder" command -- *simple version*

Syntax

Most users will only need to issue Cinder commands of the form:

```
% Cind switches configurationFileName
```

where configurationFileName is a simple fileName (default extension ".config"), which may not include host or directory specifications, and switches is either omitted or the string "/S", which evokes a useful symbol-copy option.

Semantics

Reminder: [See the C/Mesa section (section 7.9) of the Mesa Language Manual for a description of the syntax and semantics of Cedar configuration files].

This simple form of the Cind command is almost always sufficient to produce even the most complicated systems.

Examples

```
% Cind DoSomethingPackage
```

where "DoSomethingPackage.Config" contains:

```
DoSomethingPackage: CONFIGURATION
    IMPORTS Atom, DoSomethingPrivate, Process
    EXPORTS DoSomething
    CONTROL DoFirst, DoSecond, DoThird = {
        DoFirst;
        DoSecond;
        DoThird;
     }.
```

This produces "DoSomethingPackage.c" and "DoSomethingPackage.mob" from the configuration description and from the previously-compiled modules "DoFirst.mob", "DoSecond.mob" and "DoThird.mob". The output file "DoSomethingPackage.mob" contains a description of the resulting configuration.

```
% Cind /S DoSomethingPackage
```

This does the same thing, but includes all the symbols, rather than references to the components that contained them, in the output mob file.

The "Cinder" command -- ***complete syntax***

Syntax

The full Cinder syntax includes the following forms:

% Cind outputFile ←

globalSwitches inputFile[id_1: $file_1$, id_2: $file_2$, ..., id_n: $file_n$] localSwitches

% Cind [mob: mobFile, symbols: symbolFile] ←

globalSwitches inputFile[id_1: . . .] localSwitches

File names may not include host or directory specifications. Switches are letter strings introduced by the "/" character. A "-" or "~" preceding a switch specifies a FALSE setting.

Semantics

Recommendation: *Ignore this second form of the command. The full syntax is included for completeness.* Mesa was designed to allow configurations either to be self-contained by including copies of everything in their inputs, or to contain only references to earlier configurations from which symbol information could be obtained at load time. The second Cinder form allows one to be explicit about where to put everything, and under what names. The local and global switches provide an abbreviated way to specify what copy options should be invoked. At the present time, Cinder apparently supports the full syntax, but will not produce the separate output files. Cedar configurations are always single .mob files that optionally contain symbols from all their components.

The outputFile may be omitted, resulting in an output file derived from the input name, as in the previous section.

The id list may also be omitted. If present, it allows one to specify the file names from which any or all of the component names mentioned in the configuration file should be obtained. Cinder will assume that the file name matches the component name for any component that is not mentioned in this list. This feature continues the grand Mesa tradition of allowing renaming at all levels as formal parameters are bound to actual values. This binding list form, along with the ability to name the output file explicitly, can occasionally be useful when a named component has multiple implementations from which other configurations must select.

Examples

% Cind DidSomething ←
DoSomethingPackage[DoFirst: PrimusImpl, DoThird: TertiusImpl]

This produces a configuration named "DidSomething.mob", using the same configuration mentioned in the previous example. However, the file "PrimusImpl.mob" will be used to supply the component named "DoFirst" (which must appear as the configuration or program name in the source used to produce "PrimusImpl.mob"), and similarly for the "DoThird" component. "DoSecond", as before, comes from

"DoSecond.mob".

C/Mesa Language Changes

Two language constructs, "STATIC REQUESTS" and "DYNAMIC REQUESTS", have been added to C/Mesa to facilitate building multiple language configurations and loading on demand. Both constructs are added to the C/Mesa file after the EXPORTS clause, and each is followed by a list of filenames.

For static requests, Cinder puts the filenames in the "ld" command of the ".ld" file; this allows non-Mesa derived object files to be linked into the configuration. Static Requests should be used to link a module written in C into a configuration, for instance.

For dynamic requests, Cinder puts the filenames in runtime calls to XR←request("file"), which are executed when the configuration is installed. XR←request is a cedarboot routine that loads the file if it has not already been loaded. Dynamic Requests should be used to access common packages. Note that the interface checking is done by the loader, and the file found at loadtime may not export the interface you wanted.

Examples

```
Foo: CONFIG
  IMPORTS ...
  EXPORTS ...
  STATIC REQUESTS "foo.o", ...
  DYNAMIC REQUESTS "/usr/local/lib/cedar/BarPackage", ...
  CONTROL ...
```

When this configuration is sent through Cinder, "foo.o" will appear in the link command in Foo.ld. "XR←request(/usr/local/lib/cedar/BarPackage") will appear in Foo.c.

Switches

```
   StandardDefaults: PACKED ARRAY CHAR ['a..'z] OF BOOL ~ [
      FALSE, -- A  Copy all (code and symbols)
      FALSE, -- B  TRUE => make install proc call XR←StartCedarModule
      FALSE, -- C  Copy code
      FALSE, -- D  Call debugger error
      TRUE, -- E  Make installation procs be extern rather than static
      FALSE, -- F  Unused
      TRUE, -- G  TRUE => errlog goes to cinder.log, FALSE => use root.errlog
      FALSE, -- H  TRUE => Link together a packaged world. The "MakeBoot" switch.
      FALSE, -- I  Unused
      FALSE, -- J FALSE => old behaviour, ie -lxrc -lm. TRUE => no library search
      FALSE, -- K  Unused
      FALSE, -- L  Unused (used to be the lf/cr swtich - makedo still issues it, but it's obsolete)
      FALSE, -- M  MakeDo switch - if true => generate .c2c.c extension vs. .c
```

```
FALSE, -- N  Unused
FALSE, -- O  Unused
FALSE, -- P  Pause after config with errors
FALSE, -- Q  Generate .ld file with cc -pic
FALSE, -- R  Generate .ld file with no -r option to ld
FALSE, -- S  Copy symbols
FALSE, -- T  Generate .ld file with cc -PIC
FALSE, -- U  Unused
FALSE, -- V  Do version checking on the input files
FALSE, -- W  Pause after config with warnings
FALSE, -- X  Copy compressed symbols (not implemented)
FALSE, -- Y  Unused
FALSE];-- Z  Unused
```

Log

Cinder produces a file named Cinder.log on the working directory. It opens a viewer on this file if there were any errors during binding.

Cinder and UserProfile

Default settings for Cinder switches can be specified in the user profile by including a Cinder.Switches entry. [See the compiler/binder section of UserProfileDoc]. You might want to include the following entry in your profile, in order to copy symbols when binding, *but be sure to override it when you produce components for a Cedar Release:*

Cinder.Switches: S

Cinder and PCR-Packaged Worlds

The cinder includes mild support for building PCR-Pacakged applications. Starting with PCR version 2←14 the packaged world semantics have been architected such that most of the work of creating a package can be done by Cinder. To use the Cinder to create a packaged application:

1. Create a config file which describes all of the Cedar code to be contained in your application. For example, the PCedarTools.config looks like:

```
PCedarTools: CONFIGURATION
  EXPORTS ALL
  CONTROL CedarCore, BasicCedar, CommanderSysPImpl, CommanderOpsImpl,
    UnixDirectoryImpl, DatagramSocketImpl, MiscRegistryImpl, SystemNamesImpl,
    ArpaImpl, PFSPackage, RopeFileOnPFSImpl, FSOnPFSImpl, TFSOnPFSImpl,
PFSCommandsImpl, PFSPrefixMapInit, RunCommandsImpl,
CommanderBasicCommandsImpl, CommanderFileCommandsImpl,
CommandToolCompatibilityImpl, FileNamesOnPFSImpl, IntToIntTabImpl,
UserProfileImpl, LoaderImpl, StructuredStreamsPackage, MimosaStubsImpl,
DisplayStubs, TRope, TJaMPackage, RosaryImpl, TBasePackage,
TiogaExecCommands, TextReplaceImpl, IntCodePackage, MobMapperImpl,
Mimosa, C2CPackage, Cinder, MobListerUtilsPackage, XLister, SimpleStreamsImpl,
CedarProcessImpl, VersionMapImpl, ArgsPackage, ExtendADotOutPackage,
DFPackage, DFCommandsImpl, MoberyImpl, ContainersStubsImpl,
ButtonStubsImpl, MakeDoPackage, StandardMMCmds, CcCommandsImpl,
MakeDoCommands, PCedarToolsImpl, CommanderOnStandardStreamsImpl ~ {

  CedarCore;
  BasicCedar;
  CommanderSysPImpl;
  ...
  PCedarToolsImpl;
  CommanderOnStandardStreamsImpl;
  }.
```

2. Then include a C string (no, a Mesa STRING won't do!) in your configuration which will be used by PCR at startup time to describe the default command line arguments. For example, PCedarToolsImpl.mesa (part of PCedarTools.config) includes the lines:

```
DefineDefaultArgs: PROC = TRUSTED MACHINE CODE {
    "+char *defaultArgs = \"-msgs 0 -slaveiop 1 -mem 550000 -stack 90000 \\\n";
    "-tmpdir . -- -h 4000000 -install←and←run←package --\";\n.";
    };

-- mainline code:
DefineDefaultArgs[];
...
```

3. Then create a .switches file for your config which directs MakeDo to use Cinder's -h flag. For example, PCedarTools.mob.switches:

```
-hm
```

4. Then create a sun4>Package.MakeIt and sun4-o3>Package.MakeIt file which directs MakeDo to link the config with a version of the PCR. For example, sun4-o3>PCedarToolsPackage.MakeIt:

```
-- { sun4-o3/PCedarTools.c2c.o }

ComplexRsh -cmd "cc -Qpath /net/kimball/usr2/pjames/lang/ld/sparc/ -Bstatic -o
  sun4-o3/PCedarToolsPackage /pseudo/xrhome/3←0.X/LIB/OptThreads-
  sparc/XRRoot.o /pseudo/xrhome/3←0.X/LIB/OptThreads-sparc/DebugNub.o
  sun4-o3/PCedarTools.c2c.o /pseudo/xrhome/3←0.X/LIB/OptThreads-sparc/xr.a
  /pseudo/xrhome/3←0.X/LIB/OptThreads-sparc/libxrc.a"
```

5. Then run MakeDo to build your package.

Building A Packaged World

Mark Weiser
January 1, 1991

Abstract: Making a packaged PCedar world is a pain, and could be automated. Meanwhile, here is a description of the current steps. It all starts with an existing packaged world which is similar to (but smaller than) what you want. You run the base world, run the commands in that world that will load everything you want to be included, and then use the dynamic load output to edit the base world's .df's, .mesa, and .config. Along the way there are a few rules of thumb.

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Introduction

A packaged world is a Unix a.out file that contains a number of PCedar packages and code to start them automatically when the file is run.

A packaged PCedar world is somewhat analogous to a DCedar boot file.

PCedar is always ready to dynamically load new applications when they are invoked by typing their name to a Cedar Commander. (A Commander is like a Unix shell, or a Lisp listener). A packaged world has some applications "pre-loaded", so they can start immediately.

PCedar is almost always run from a packaged world. The smallest such world is *Commanderworld*, which contains just enough code to start a Cedar Commander listening to a tty, and load other things as needed. Other common packaged worlds are *X11ViewersWorld*, *RawViewersWorld*, and *PrintingWorld*.

There are several reasons for making a packaged world:

1. To start up more quickly. A packaged world starts much more quickly than the same world dynamically loaded.
2. To make more efficient use of VM and swap. A packaged world contains its non-dynamically loaded code in the normal Unix text space. This can be several megabytes for a large package (X11ViewersWorld has 5.6MB just for code as of 1/1/91). This space is more compactly allocated in the text area than when it is dynamically loaded. Unix also recognizes the a.out text area as read-only, and pages in directly from the a.out rather than using additional disk swap space.
3. To decrease the work of the garbage collector (GC). Some GC overhead is required for skipping over dynamically loaded program files. Pages in the a.out text area are not seen by the collector.
4. To simplify the environment needed to run PCedar. PCedar dynamic loading requires a complicated system of files in well-known places to work. As long as it does not try to dynamic load, a packaged world needs none of this (with a few exceptions, like if it needs fonts for viewers or imaging). Packaged worlds are thus more portable and less dependent on the PARC PCedar environment conventions.

Building a packaged world breaks up into two parts: initializing a new world, and adding to an old world. To make a new kind of packaged world, one initializes a new world by making a copy of all the appropriate files from an old world. Then, one treats the new world as an old world, and adds to it.

Initializing a New World

Finding a world to start from.

To make a new world, find an old packaged world to start from. Right now the "PackagedWorlds" directory in PCedar2.0 is a bad place to start, because it contains three packaged worlds at the same time, and it hard to tease them apart. Perhaps someday soon there will be a separate "PackageInitWorld". Meanwhile, the cleanest place to start (but far from the smallest) is PrintingWorld.

For the sake of generality we'll call the starting world StartingWorld below. We'll call the new package you want to make NewWorld.

Turning the old world into a new world

Create a clean -ux directory called NewWorld. Give it sun4-o3 and sun4 subdirectories, like this:

```
mkdir NewWorld
cd NewWorld
mkdir sun4-o3
```

Now bringover the StartingWorld into NewWorld:

```
bringover -m StartingWorld-Suite.df
```

note well: the bringover creates symbolic links. If you do the work below from PCedar, the links will be broken at the proper time and all will be well. Otherwise, disaster may happen as you accidently change the original files instead of your copy.

Finally, do a lot of renaming. First, the files. There will be the following ten files that need to have the StartingWorld part of their name changed to NewWorld. (Other files starting with StartingWorld, like StartingWorldImpl.c2c.c and StartingWorldImpl.mob, can be ignored.):

```
rename NewWorld-Suite.df ← StartingWorld-Suite.df
rename NewWorld-PCR.df ← StartingWorld-PCR.df
rename NewWorld-Source.df ← StartingWorld-Source.df
rename NewWorld.config ← StartingWorld.config
rename NewWorldImpl.mesa ← StartingWorldImpl.mesa
rename  NewWorldDoc.tioga ← StartingWorldDoc.tioga
rename NewWorld-sun403.df ← StartingWorld-sun403.df
rename BuildNewWorld.cm ← BuildStartingWorld.cm
rename NewWorld ← StartingWorld
rename    sun4-o3/NewWorldPackage.MakeIt    ←
     sun4-o3/StartingWorldPackage.MakeIt
```

Now inside the files.

1. The four .df's must have their internal references to any files on the above list, and any other files referencing StartingWorld files (now including things like StartingWorldImpl.mob) changed to the new name.

2. NewWorldImpl.mesa and NewWorld.config must have their packagenames changed to the NewWorld form. NewWorld.config has a reference to StaringWorldImpl that needs to be changed to NewWorldImpl. Generally, look all around and use common sense.

3. sun4-o3/NewWorldPackage.MakeIt will have references to two StaringWorld files: StartingWorld.c2c.o and StartingWorldPackage. Change these to their new world form.

4. BuildNewWorld.cm needs to have all its references to StartingWorld files changed to the appropriate NewWorld files.

5. NewWorld is a shell script that will eventually need to be placed in

/project/pcedar2.0/bin by Willie-Sue. Edit it to replace references to StartingWorld files to NewWorld files.

6. Of course, NewWorldDoc.tioga needs to be completely redone. [Please take the time to write some documentation.]

7. Don't forget to change the initial comment node in each file to refer to the new name.

Now check. You should be able say the following and construct a new world that acts just like the starting world:

```
BuildNewWorld
```

Then the following command will execute it:

```
sun4-o3/NewWorldPackage -thread 80
```

Note that this is not the normal way to execute this world, which will be via the NewWorld shell script, which will set all sorts of other parameters to the world. But this is enough to give a quick test. (The "-thread 80" is because the default number of threads, 40, is too small for many worlds. The symptom of running out of threads is the world quietly stopping. The RemoteDebugTool or Cirio will show a number of threads equal to the maximum.)

If the above worked, you have new initialized a new package world just like the old package world, and you are ready to add new things to it. See the next section.

If the above did not work, then you may have made a mistake, or these instructions may contain a mistake. Try grepping all the files for the word "StartingWorld" or "Starting". They should not appear anywhere.

If you find a mistake in these instructions, please let me know.

Add to an Old World

In this section, we assume you have a packaged world already working for you, and you want to add some contents. To continue the names used in the previous section, we will call the world you are modifying "NewWorld". There are now four files that need lots of editing: NewWorld-PCR.df, NewWorld-sun4O3.df, NewWorld.config, and NewWorldImpl.mesa. All must be updated to reflect the things you want to add to your packaged world.

Finding out what to add

The easiest way to find out what to add is to first run the existing packaged world that you want to augment. To a commander in that world, execute a sufficient number of commands to dynamically load all the things you want added to the packaged world. Then execute the "installed" command. *Save that all that Commander output!* The output of the "installed" command and the names of all the .c2c.o's that were loaded will be used in the following steps.

Preparing a working directory

Now you want to make a place into which to bringover the packaged world to be modified. Create a clean -ux directory called NewWorld (-vux is still a little bit broken for these purposes). Give it a sun4-o3 [and sun4?] subdirectories, like this:

```
mkdir NewWorld
cd NewWorld
```

```
mkdir sun4-o3
```

Now bringover NewWorld into the new directory:

```
bringover -m NewWorld-Suite.df
```

Modifying the .df's

This is the most tedious part. For each .c2c.o loaded, you must add it to the -sun4O3.df file, and add its corresponding .mob file to the -PCR file. For instance, if the loader gave you the message:

```
Ran /PCedar/HostCoordination/sun4-o3/HostAndTerminalImpl.c2c.o!1
```

Then add the following to NewWorld-PCR.df:

```
Imports [PCedar2.0]<Top>HostCoordination-PCR.df Of ~ =
        Using [HostAndTerminalImpl.mob]
```

and add the following to NewWorld-sun4O3.df:

```
Imports [PCedar2.0]<Top>HostCoordination-sun4O3.df Of ~ =
        Using [sun4-o3>HostAndTerminalImpl.c2c.o]
```

Probably you need to already know how to use .df files to make sense of the above. If you are not a .df file user, get someone who is to help you.

Modifying the..config

The .config consists of three parts that matter for this step: "STATIC REQUESTS", "CONTROL", and "~ {" (the main body). Editing each part is discussed below.

For parts two and three, the assumption is made that the name of a package is the same as the name as the file it is in. Since much of makedo and tiogacomforts break down without this assumption, it is a pretty safe one. If things keep going wrong and you are at wits end, question your assumptions.

STATIC REQUESTS

The first part has the keywords "STATIC REQUESTS". Files mentioned here are C-based, not Cedar-based, "glue" files that do not fit into the normal rules for Cedar packages. These files generally have all of the following characteristics:

the word "Glue" in its name
name ends in ".o", but not ".c2c.o"
the file has no corresponding ".mob" file
the loader announced when loading the file: "(it has no install or start proc)".

If any of the dynamically loaded files fit these clues, add them here. Some typical glue files mentioned under STATIC REQUESTS are: "ThreadsTLI.o", "XNSRouterGlue.o", "CMUXGlue.o", "NetworkStreamSupportTCPGlue.o", and "PCRCmdGlueImpl.o".

Once you put a file in the STATIC REQUESTS line, *do not* also mention its name anywhere else in the .config file.

CONTROL

Packages are listed here in the order in which they are to be started. Generally you add to the end (*almost!*) of this list, and in the order in which the files were dynamically loaded (since that was the order they were started, and it worked). The package names are not quite added to the end of the list because two special packages must be mentioned second-to-last and last.

Second-to-last must come NewWorldImpl, the package resulting from the NewWorldImpl.mesa that we will be editing in the next section. Last must come CommanderOnStandardStreamsImpl, since this starts the command interpreter on the current stdin stream and consequently never returns.

~ { (main body)

Following the CONTROL section is simply a listing of all the packages which are to be included in the packaged world. CedarCore must be the first file mentioned, otherwise order does not matter. Just add your new package names to the end. These will be exactly the same names that you added to the CONTROL section, above.

A duplicate name here will cause MakeDo to mysteriously crash. Make sure each name appears exactly once.

Modifying the .mesa

NewWorldImpl.mesa is the final file that needs detailed editing. Generally it contains lots of mysterious stuff that is better left alone. But there are two lines that need your attention:
DefineDefaultArgs
and
[] ← CommanderOps.DoCommandRope["Installed ...

The line beginning **DefineDefaultArgs** enables you to set different defaults for your packaged world. The values set in **DefineDefaultArgs** are overridden by specific values given on the Unix command line when your package world is run, otherwise they become the default. It is generally kind to set values for -thread and -mem as small as possible for your package world to run. Finding the smallest value can only be done by trial and error, unfortunately. If in doubt, don't change this line. But if your packaged world change involves running significantly many new programs that may have reason to create threads, you may want to up the -thread argument. Generally playing with these values is a black art, but you need to know that a wrong value can mess you up.

The line beginning **[] ← CommanderOps.DoCommandRope["Installed** is any easy one to get right. Remember way back when you ran your packaged world, you ran the "installed" command after you had loaded everything you wanted in the package? Well, now you want to shift-select the output of the "installed" command you did back then into the list of names following the world **Installed** above. Just replace all the words following **Installed** up to the closing quote (") with the saved output of the "installed" command. This step ensures that your packaged world is told about all the things it has inside.

The function of some of the other lines in NewWorldImpl.mesa are pretty obvious, like:

```
[]    ←    CommanderOps.DoCommand["CommandsFromProfile
      CommandTool.NoViewersBootCommands", NIL];
```

Feel free to change these if you feel adventurous.

Build, Test, Repeat

Now you need to try your world out, fix the bugs that may have occured during the editing steps. try again. and repeat. In order to debug the .df's during this stage. it is useful to smodel and bringover at each round. Thus steps are:

1. smodel NewWorld-Suite.df

1a. (click RESET on any viewers you had open which were modified by the smodel step. like viewers on .df files).

2. bringover NewWorld-Suite.df. This brings over any new files you added to the .df files.

3. BuildNewWorld

4. try running sun4-o3/NewWorldPackage (don't forget the -thread argument. if you need it. See above).

5. Repeat.

For an even more serious test. after the "smodel" delete the directory and everything in it and really start from scratch with the bringover step. Don't forget to make the sun4-o3 and sun4 subdirectories. as needed.

If you are successfully building sun4-o3/NewWorldPackage's at each step. then be careful because each of your smodel steps is moving five or more megabytes onto a shared backup disk containing /pseudo/pcedar2.0 that can fill up quickly. You may want to delete sun4-o3/NewWorldPackage before doing the smodel step. Smodel will complain. but work fine. Of course. don't delete it when you are finally done. that's when you want it to go.

Things that can go wrong

This section will try to be a placeholder for hints and tricks that people accumulate.

1. The final step in BuildNewWorld forks off a Unix "ld" process that will require at least twice as much VM as the final a.out is large. It is not uncommon to not have enough VM available. and to have this step die with a message like: "ld: /usr/lib/crt0.o: out of memory for relocation symbols". or some variation. If this happens you can use the unix command "pstat -s" to see how much VM is available. and try to make more. You may need to kill off your Viewers world and do the final step from a PCedarTools world. or do it while bridged or rlogined to another machine with more VM at the moment.

2. If your packaged world uses more threads than are specified in the command line or the NewWorldImpl.mesa. then it will mysteriously come to a quiet and sedentary state of nothing happening. Use Cirio or the RemoteDebugTool to look at it and count the number of threads. If they are exactly equal to the number of default threads. you are likely out of threads. Try running the packaged world again with a -threads value somewhat larger on the command line.

Appendix -- complete example of adding packages to PrintingWorld

This appendix contains a transcript of the main steps in adding the package NetCommander to PrintingWorld. It consists of two major parts: a Unix shell transcript. and a Cedar Commander transcript. First the Unix. trying to find out what to load. The transcripts have been augmented with comments explaining what is going on.

Initial Unix shell transcript

Run PrintingWorld in a Unix shell.

```
zwilnik% PrintingWorld
Running /pseudo/pcedar2.0/printingworld/sun4-o3/printingworldpackage.~4~
Welcome to Basic PCedar 2.0.5 of September 25, 1990 2:20:01 pm PDT.
Using /pcedar/EssentialStyles/cedar.style!15 . . . ok
[[COMMANDER←INITIAL←COMMAND=]]
Commander %
```

Now give the NetCommander command to PrintingWorld, and save the resulting list of loaded files.

```
Commander % NetCommander
Ran /PCedar/NetworkStream/sun4-o3/NetworkNameImpl.c2c.o!2
Ran /PCedar/SunRPCRuntime/sun4-o3/SunRPCRuntime.c2c.o!3
Ran /PCedar/SunPMap/sun4-o3/SunPMapClientStub.c2c.o!2
Ran /PCedar/SunYP/sun4-o3/SunYPBindClientStub.c2c.o!3
Ran /PCedar/SunYP/sun4-o3/SunYPClientStub.c2c.o!3
Ran /PCedar/SunYP/sun4-o3/SunYPFindImplP.c2c.o!4
Ran /PCedar/SunYP/sun4-o3/SunYPAgentImpl.c2c.o!4
Ran /PCedar/NetworkStream/sun4-o3/NetworkNameSunYPImpl.c2c.o!3
Ran /PCedar/NetworkStream/sun4-o3/NetworkNameEtcHostsImpl.c2c.o!4
Ran /PCedar/CommTimer/sun4-o3/CommTimerImpl.c2c.o!2
Ran /PCedar/NetworkStream/sun4-o3/NetworkStreamImpl.c2c.o!3
Loaded /PCedar/NetworkStream/sun4-o3/NetworkStreamSupportTCPGlue.o!1 (it has no install
or start proc)
Ran /PCedar/NetworkStream/sun4-o3/NetworkStreamSupportTCPImpl.c2c.o!3
Ran /PCedar/NetworkStream/sun4-o3/NetworkStreamTCPImpl.c2c.o!1
Ran /PCedar/NetworkStream/sun4-o3/NetworkStreamSPPOnBasicStreamImpl.c2c.o!2
Ran /PCedar/NetCommander/sun4-o3/NetCommanderImpl.c2c.o!1
```

Now do the "installed" command, for use later.

```
Commander % installed
Args Artwork BackStop BasicCedar CedarCore CedarProcess CodeTimer Commander
CommandTool CommTimer CrRPC CubicSplinePackage DES DisplayStubs Draw2d Feedback
FS GargoyleCore GargoyleModeler GGToIP Imager ImagerFontFilter ImagerMemory Interp
Interpress InterpressToCompressedIP Lines2d Math NamedColors NetCommander
NetworkName NetworkStream PCRCmd PFS ProcStream Rosary StackTrace SunPMapClient
SunRPCRuntime SunYPAgent TBase TFormat TFS TiogaExecCommands TiogaImager
TiogaImagerCommands TJaM TRope UserCredentials UserProfile UserProfileCommands
Viewers XNSAuthentication XNSBasicTypes XNSClearinghouse XNSCredentials XNSPrint
XNSPrintingClient XNSPrintingUI XNSServerLocation XNSStuff XNSTransport
Commander %
```

That's all for the unix side, for now.

PCedar Commander transcript

Make a fresh directory, and bringover PrintingWorld into it.

```
 % cd PrintingWorld
  Not a directory: /net/gharlane/zuni/mark/PrintingWorld
  % mkdir PrintingWorld
  Ran /PCedar/UnixCommands/sun4-o3/UnixSpawnImpl.c2c.o!4 (s=15572 m=October 31,
  1990 12:06:19 pm PST)
  Ran /PCedar/UnixCommands/sun4-o3/UnixCommandsImpl.c2c.o!3 (s=12490 m=March
  6, 1990 10:52:55 am PST)
  % cd PrintingWorld
  /net/gharlane/zuni/mark/PrintingWorld/
  % mkdir sun4-o3
  % qbo -m PrintingWorld-Suite.df


    -- many many lines of output omitted


  0 errors, 0 warnings, 178 files acted upon
  %
```

go to work on the PrintingWorldImpl.mesa file.

```
  % ls *.mesa
  /net/gharlane/zuni/mark/PrintingWorld/
   PrintingWorldImpl.mesa    2421 21-Nov-90 15:07:02 PST
  -- 1 files, 2421 total bytes
  % Open PrintingWorldImpl.mesa
    Created Viewer: /net/gharlane/zuni/mark/PrintingWorld/PrintingWorldImpl.mesa
  % -- edit in the results of the installed command
```

go to work on the PrintingWorld.config file

```
% ls *.config
/net/gharlane/zuni/mark/PrintingWorld/
 PrintingWorld.config      3951 21-Nov-90 15:12:54 PST
-- 1 files, 3951 total bytes
% -- edit in the .c2c.o's that needed to be loaded into the config file, twice, in order.
% Open PrintingWorld.config
        Created Viewer: /net/gharlane/zuni/mark/PrintingWorld/PrintingWorld.config
```

go to work on the .df's

```
% -- edit in mentions of the .c2c.o's that were loaded.
% Open PrintingWorld-sun4O3.df
        Created Viewer: /net/gharlane/zuni/mark/PrintingWorld/PrintingWorld-sun4O3.df
% -- edit in mentions of the .mobs corresponding to the .c2c.o's
% open PrintingWorld-PCR.df
        Created Viewer: /net/gharlane/zuni/mark/PrintingWorld/PrintingWorld-PCR.df
```

smodel, bringover, and try to build

```
% smodel PrintingWorld-Suite.df
```

-- many messages omitted

```
% qbo -m PrintingWorld-Suite.df
% ls *.cm
/net/gharlane/zuni/mark/PrintingWorld/
 BuildPrintingWorld.cm      562 12-Oct-90 00:09:50 PDT
-- 1 files, 562 total bytes
% BuildPrintingWorld
Forking Mimosa -lc PrintingWorldImpl
Done with Mimosa -lc PrintingWorldImpl
Forking Cind -hm PrintingWorld
Forking MMCCMesa -class sun4 -name PrintingWorldImpl -dir sun4-o3/ -mSw "-c -O3" -uSw ""
Done with MMCCMesa -class sun4 -name PrintingWorldImpl -dir sun4-o3/ -mSw "-c -O3" -uSw
""
Cinding: PrintingWorld-hm.... no errors, 23 warnings.
End of cinding
W
Forking MMCCConfig -class sun4 -name PrintingWorld -dir sun4-o3/ -mSw "-c -O3" -uSw ""
-raux    sun4-o3>XNSRouterGlue.o   sun4-o3>PCRCmdGlueImpl.o
sun4-o3>NetworkStreamSupportTCPGlue.o sun4-o3>CMUXGlue.o [long list omitted]
Done with MMCCConfig -class sun4 -name PrintingWorld -dir sun4-o3/ -mSw "-c -O3" -uSw ""
-raux    sun4-o3>XNSRouterGlue.o   sun4-o3>PCRCmdGlueImpl.o
sun4-o3>NetworkStreamSupportTCPGlue.o sun4-o3>CMUXGlue.o [long list omitted]
Forking Source [net]<gharlane>zuni>mark>PrintingWorld>sun4-o3>PrintingWorldPackage.MakeIt
Done   with   Source
[net]<gharlane>zuni>mark>PrintingWorld>sun4-o3>PrintingWorldPackage.MakeIt
Failed: Cind -hm PrintingWorld
1 step failed; 4 ok.
2 goals OK; 0 not.
%
```

*-- the 1 step failed is just the warning messages from the cind step. It is the "*2 goals OK*" that really counts.*

-- tried running PrintingWorldPackage, all worked fine. See second Unix transcript, below.

-- All done, so do a final smodel, and call it quits.

```
% smodel PrintingWorld-Suite.df
Start: PrintingWorld-Suite.df
Start: PrintingWorld-PCR.df
[net]<gharlane>zuni>mark>PrintingWorld>PrintingWorld.c2c.c  -->
[PCedar2.0]<PrintingWorld>PrintingWorld.c2c.c {01-Jan-91 15:24:11 PST}
[net]<gharlane>zuni>mark>PrintingWorld>PrintingWorldImpl.c2c.c  -->
[PCedar2.0]<PrintingWorld>PrintingWorldImpl.c2c.c {01-Jan-91 15:23:47 PST}
[net]<gharlane>zuni>mark>PrintingWorld>PrintingWorld.mob  -->
[PCedar2.0]<PrintingWorld>PrintingWorld.mob {01-Jan-91 15:24:11 PST}
```

```
[net]<gharlane>zuni>mark>PrintingWorld>PrintingWorldImpl.mob -->
[PCedar2.0]<PrintingWorld>PrintingWorldImpl.mob {01-Jan-91 15:23:47 PST}
PrintingWorld-PCR.df --> [PCedar2.0]<Top>PrintingWorld-PCR.df {01-Jan-91 15:35:57 PST}
('#')
End: PrintingWorld-PCR.df
Start: PrintingWorld-Source.df
[net]<gharlane>zuni>mark>PrintingWorld>PrintingWorldImpl.mesa -->
[PCedar2.0]<PrintingWorld>PrintingWorldImpl.mesa {01-Jan-91 15:21:21 PST}
PrintingWorld-Source.df --> [PCedar2.0]<Top>PrintingWorld-Source.df {01-Jan-91 15:36:17 PST}
('#')
End: PrintingWorld-Source.df
Start: PrintingWorld-sun4O3.df
[net]<gharlane>zuni>mark>PrintingWorld>sun4-o3>PrintingWorldImpl.c2c.o -->
[PCedar2.0]<PrintingWorld>sun4-o3>PrintingWorldImpl.c2c.o {01-Jan-91 15:24:28 PST}
[net]<gharlane>zuni>mark>PrintingWorld>sun4-o3>PrintingWorldPackage -->
[PCedar2.0]<PrintingWorld>sun4-o3>PrintingWorldPackage {01-Jan-91 15:29:30 PST}
PrintingWorld-sun4O3.df --> [PCedar2.0]<Top>PrintingWorld-sun4O3.df {01-Jan-91 15:36:25
PST} ('#')
End: PrintingWorld-sun4O3.df
End: PrintingWorld-Suite.df (DF file is unchanged)
10 files acted upon
```

-- After the Smodel. try the PrintingWorld command in a shell. from the usual PrintingWorld command. Check that it gets the new version from the smodel. check everything else.

-- all done.

Final Unix Transcript

```
zwilnik% cd PrintingWorld
zwilnik% cd sun4-o3
zwilnik% PrintingWorldPackage -threads 30
Welcome to Basic PCedar 2.0.5 of December 11. 1990 11:23:21 am PST.
Using /project/pcedar2.0/imagerfonts/styles/cedar.style . . . ok
[[COMMANDER←INITIAL←COMMAND=]]
Commander % netcommanderon 4800
Connection command: NetCommander [13.1.101.88]:4800 TCP
Commander %
```

-- All is well. the NetCommand command did no dynamic loading.

PCRDoc.tioga
Russ Atkinson (RRA) December 1. 1988 12:00:07 pm PST
Weiser. May 24. 1989 5:35:09 pm PDT

PCRDoc

-- Portable Common Runtime

Russ Atkinson et. al.

Abstract: PCR is a program that provides runtime support for garbage collection, dynamic loading of modules, and lightweight processes (threads). Its primary use currently is to support the Cedar programming language, but it is not limited to such use. PCR can be used to mix Cedar code, C code, and (even) Kyoto Common Lisp code.

Created by: Russ Atkinson et. al.

Maintained by: Mark Weiser <Weiser.pa>

Keywords: PCR, Cedar, PCedar, Sun, garbage collection, dynamic loading, lightweight processes, threads, xrsh

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

For Distribution Outside Xerox

1. Introduction

PCR is a program that provides runtime support for garbage collection, dynamic loading of modules, and lightweight processes (threads). Its primary use now is to support the Cedar programming language, but it is not limited to such use. PCR can be used to mix Cedar code, C code, and (even) Kyoto Common Lisp code.

Warning! This documentation is tentative and temporary. It currently applies to the threads and non-threads variants of PCR. Non-threads is rarely used--assume you are using threads unless you know otherwise. Please let Mark Weiser know about mistakes.

2. Switches, Commands, and Variables

Switches

Switches that always work

Switches are taken from the command line that starts PCR. These switches come *after* the threads switches on the command line.

-c[+] *name*	Read cmds from *name* after BasicDotLoadeeCommands and .xrboot
-c- *name*	Read cmds from *name* before any other command files
-d	Don't create DBX init file.
-D	Don't maintain a debugging symtab file (increases dynamic loading speed slightly)
-g	Turn on garbage collector verbose mode (default is off).pcr
-G	Turn off garbage collection. Never try to collect. Storage grows without bound.
-h *#bytes*	Add <# bytes> to heap before starting.
-l	Don't attempt to initialize dynamic loading (default is off)
-m	Be miserly in growing the heap (minimizes working set at the expense of extra GC's.)
-n	Don't read cmds from BasicDotLoadeeCommands
-N	Don't read cmds from .xrboot
-t	Turn off printing timing info after each pcr command (default is on)
-tmp *dir*	Use directory *dir* for temporary files (default is /tmp/)
-v	Turn on dynamic loader verbose mode.

Switches for Threads only

The switches below, for controlling details of the threads world, must preceed the switches above in the command line. The special switch "--" separates the two. Thus the generic PCR command line is:

PCR [threads-switches]* -- [non-threads switches]*

If the "--" is missing, the threads-switches will be reinterpretted again as non-threads switches. Usually this causes no damage, just lots of error messages.

The threads switches are pretty much for wizards only. About the only thing you might want to change on your own is the "-thread" switch, to increase the maximum number of threads.

-vp *number*	Number of virtual processors. Default is 1. Must be 1 for packaged applications.
-slave *number*	Number of slave processors. Default is 0. For future use.
-iop *number*	Number of IOP processors. Default is 3, of which one is the garbage collector process, leaving 2 to cache file descripters.
-thread *number*	Maximum number of simultaneous threads. Default is 20.
-mem *number*	Size of startup system memory. The default is the right size for non-packaged applications. This value must be enlarged, by trial and error, for packaged applications.
-swapdir *filename*	Filesystem directory in which to create a backing file for shared memory used by PCR. Because of bugs in SunOS, this directory cannot be on an NFS mounted partition, but must be on the local disk.
-log *logfilename*	File into which threads tracing output should be printed. *WARNING--FOR THREADS WIZARDS ONLY.*
-[no]dbp	**dbp** is the **de-bugger process**. This switch is for use by CIRIO, or other special debuggers. *WARNING--FOR THREADS WIZARDS ONLY.*
-[no]trace	This switch turns on or off internal tracing of the threads code. Traces are printed to the log file specifed by the *-log* switch. *WARNING--FOR THREADS WIZARDS ONLY.*
--	End of the threads switches. All of the switches up to and including this one will not be passed on for further switch processing by PCR.

Commands

Commands can be issued to the small command interpreter in PCR.

?, help	print a brief help message
@*name*	read commands from *name*
call *subr*	call subroutine *subr* in last dynamically loaded module (*subr* must be a full internal name, including "←" prefix. E.g., to call C

	procedure "main", use "call ←main".
callall *subr*	call subroutine *subr*. Same interpretation of *subr* as for "call" command.
dbx, debug	start running a dbxtool pointed at the current cedarboot. *THIS PROBABLY DOES NOT WORK. AVOID.*
gc *switch*	turn garbage collection on (*switch*: on) or off (*switch*: off). Note: turning collection off with this switch will still take working set hits when the GC *would* have run, because it will still go through the motions of collecting without actually doing so. Use the -G switch to really make the GC go dead.
q, quit	leave PCR forever
set *var value*	set the value of *var* to *value*. For C variables *var* should include the "←" prefix.
show *var*	print the value of variable *var*. For C variables *var* should include the "←" prefix.
uload, unixload	load a unix .o into the PCR world. *Following* the file name exactly one of the following switches may occur: -r -d. -d means don't update the debugger file, -r means don't try to run this file after loading it.

Command lookup is case-insensitive, so "HeLp" is equivalent to "help".

Environment Variables

The pcr looks certain environment variables when starting up.

XR←HOME - Should be set to the location where the PCR was created from the distribution tape.

XR←VERSION - Should be set to the version of pcr that you wish to run. The default is INSTALLED.

XR←CONFIG - Should be set to either **PseudoThreads** or **Threads**, depending on the type of pcr world you wish to be running on.

XR←MACH - Should be set to the target machine architecture (currently mc68020 or sparc), default is the machine on which it is currently running.

/usr/local/bin/pcr is a shell script which executes *${XR←HOME}/${XR←VERSION}/BIN/${XR←CONFIG}-${XR←MACH}/PortableCommonRuntime*

LIBRARIES

The Portable Common Runtime has a set of libraries that can be used by regular C program to link against so that they may run on top of the pcr. The libraries are *libxrc.a*, *libxrpixrect.a*, and *xr.a*. They can be found in: *${XR←HOME}/${XR←VERSION}/LIB/${XR←CONFIG}-${XR←MACH}/*

3. Additional information

The uload command loads in a module. The file name of a .o format file should be given as the argument. After each load command a synthetic symbols file is placed in the temporary directory with the name symtab.pidXXX. where XXX is the process id of cedarboot. If the .o file contains procedures with the names "XR←install" or "XR←run", these are called after loading, XR←install first. The procedures do *not* need to be externally visible to be called. (E.g. for C they can be declared **static.**)

When PCR first starts, the shell command file XrDBX is written into the working directory. XrDBX can be used to start a dbx debugger on the most recently started version of PCR in that working directory.

The call command calls a procedure in the most recently loaded module. The C language internal name should be used (i.e. the declared name should be prefaced with an underscore.) When the call returns some statistics will be printed.

The set and show commands can be used to print and change the values of 32-bit variables. For instance, "set ←safe←to←gc←print 0" will set the value of the variable "safe←to←gc←print" to 0. Notice the leading '←'--this is necessary for C generated names. The command "show ←safe←to←gc←print" will print the value of this variable. 32 bits are set or read each time -- using set or show on variables of smaller size may give unpredictable results.

The quit command cleanly exits cedarboot, removing any temporary files that it might have created.

The debug command starts up dbxtool on the current instance of PCR. Note that if you use the debug command, and later the load command, the old dbxtool will not know about anything in the later load. In order to run dbxtool you must be running cedarboot inside SunView and not from a remote login or Bridge.

Theimer. April 21. 1992 8:36 pm PDT
Willie-s. May 26. 1992 4:29 pm PDT

cedarrpcgenDoc

SunRPC stubs for Cedar

Marvin Theimer

Abstract: cedarrpcgen is a modified version of Sun's RPCGen that accepts interface specifications in Sun's RPC language and emits stubs written in Cedar. The stubs allows C and Cedar programs to communicate via remote procedure call using Sun's RPC protocol.

Created by: Marvin Theimer

Maintained by: Marvin Theimer <Theimer:PARC:Xerox>

Keywords: RPC, Stub compiler, Sun RPC

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Using cedarrpcgen

Invoking cedarrpcgen

cedarrpcgen is a C program that must be run from a Unix command shell. To run the stub generator, use the command

/project/cedar10.1/bin/cedarrpcgen *module*.x

where *module*.x is the name of the file containing the interface specification. This command will generate up to five files (where "*module*" is replaced by the actual name supplied):

module.mesa	-- A Cedar interface for this module.
*module*ClientImpl.mesa	-- Client stubs that export the interface and make RPC calls using the SunRPC or SunRPCStream package.
*module*ServerImpl.mesa	-- Server stubs that import the interface and incoming RPC requests into calls on the interface.
*module*GetPut.mesa	-- An interface for marshalling and unmarshalling the types in *module*.mesa.
*module*GetPutImpl.mesa	-- Its implementation.

If no options are specified, then cedarrpcgen will produce stubs for Cedar10.0. Options can be used to produce either UDP or TCP stubs for PCedar2.0:

-PCedar	-- Produce PCedar2.0 stubs that use SunRPC to produce UDP-based RPC calls.
-PCedarStream	-- Produce PCedar2.0 stubs that use SunRPCStream to produce TCP-based RPC calls.

Two additional options are available that provide "creature comforts" in the generated Cedar stubs. It is envisioned that users will normally invoke these options; they are not the default for the sake of backwards compatibility.

-p	-- Strip module name prefixes from symbol names. For example, if the type definition foo←bar exists in the RPC interface file **foo.x** then this option will cause the Cedar stubs to use the name foo.bar instead of foo.foobar. This option also provides a slightly modified version of naming for union types that produces more readable Cedar code.
-e	-- Expand procedure input arguments. SunRPC .x files allow specification of only a single input argument to a procedure. This option will expand any procedure input variable that is a record type into a list of arguments comprised of the fields of the record definition in the emitted Cedar stubs (both the client and the server stubs).

Note: For PCedar2.0 you can either produce UDP-based stubs or TCP-based stubs, but not both. I could have made the TCP-based stubs be emitted under different file names - so that one could produce both sets of stubs - but have not made that extension because PCedar2.0 will soon be obsolete and current PCedar2.0 clients need one or the other, but not both.

Input files

The input language is Sun's RPC language. For a detailed description, see "rpcgen Programming Guide", in the *Network Programming* manual.

Require files

To run the resulting code in Cedar, you will need require files to ensure that the proper packages are running. For the client, the file should look like this:

```
-- ▸module◂Client.require
-- Copyright ■ 1990 by Xerox Corporation. All rights reserved.
-- Generic require file for cedarrpcgen clients.

-- SunRPC and SunRPCAuth
require PCedar SunRPCRuntime SunRPCRuntime

-- SunPMapClient
require PCedar SunPMap SunPMapClient

-- The client stub.
run ▸module◂GetPutImpl
run ▸module◂ClientImpl
```

For the server, it should read:

```
-- ▸module◂Server.require
-- Copyright ■ 1990 by Xerox Corporation. All rights reserved.
-- Generic require file for cedarrpcgen servers.

-- SunPMapLocal
require PCedar SunPMap SunPMapLocal

-- SunRPC
require PCedar SunRPCRuntime SunRPCRuntime

-- The server stub
run ▸module◂GetPutImpl
run ▸YourServerImplementation◂
run ▸module◂ServerImpl
```

DF files

Your *package*-PCR.df should contain the following lines to get the .mob files needed to compile the stubs:

```
Imports [PCedar2.0]<Top>SunRPCRuntime-PCR.df Of ~ =
 Using [SunRPC.mob, SunRPCAuth.mob]

Imports [PCedar2.0]<Top>RuntimeSupport-PCR.df Of ~ =
 Using [Basics.mob]

Imports [PCedar2.0]<Top>SunPMap-PCR.df Of ~ =
 Using [SunPMap.mob, SunPMapLocal.mob]
```

Imports [PCedar2.0]<Top>Communication-PCR.df Of ~ =
 Using [Arpa.mob]

Imports [PCedar2.0]<Top>SunPMap-PCR.df Of ~ =
 Using [SunPMapClient.mob]

Run-time considerations

cedarrpcgen produces an "interface object" for each program defined in *module.x*. Interface objects contain a procedure variable for each procedure defined in the corresponding SunRPC program. Interface objects are imported into a client program using the procedure ImportFooN, where Foo is the name of the program and N is the version number of the program. Interface objects are exported by server programs using the procedure ExportFooN.

The best way to get a feeling for how to use the stubs produced by cedarrpcgen is to look at the example program in the cedarrpcgen directory. The fortytwo program in this directory is an example that is currently compiled for use with UDP (the ImportFoo and ExportFoo routines to use with TCP are shown as comments). The program uses three different .x files: fortytwo.x, fourtwo.x, and fortyone.x. fortytwo.x includes the other two. The client program is named fortytwoClient and the server program is named fortytwoServer. See their respective config files for more information about their composition.

Note: The fortytwo program also illustrates a problem with SunRPC* in PCedar2.0; namely that XR←DepositField for real and double real numbers doesn't seem to work. If you run either fortytwoClient or fortytwoServer against some other server or client, respectively, then you will see them get part of the way through execution and then generate an exception signal. This problem is supposedly fixed in Cedar10.1.

2. Generated code

This section describes how the various types allowed in Sun's RPC language are translated into Cedar code.

Simple types

The simple data types in C translate as follows

`bool`	BOOLEAN
`char`	BYTE
`u←char`	BYTE
`short`	INT16
`u←short`	CARD16
`int`	INT32
`u←int`	CARD32
`long`	INT32
`u←long`	CARD32

Note that `char` arrays may not do what you expect, as they transmit a 32-bit word for each character. Instead, try `string` or `opaque`.

Structures

Structures are translated component by component.

Unions

Unions are translated into REFs to variant records, with the first field being the tag type declared in the RPC file, and the actual Cedar tag being an anonymous enumeration. The case arms specified in the RPC description are used in a SELECT statement to marshall and unmarshall the correct variant. *Warning: The value of the user-defined tag field must match the Cedar tag or the marshalling routines will get an exception trying to NARROW a REF.*

When unions are transmitted, an allocation is done on the receiving side to hold the incoming value.

Fixed-length Arrays

Fixed-length arrays are translated into packed arrays of the corresponding type.

Variable-length arrays

Variable-length arrays are translated into REFs to SEQUENCES with a max length of "size". When a variable-length array is transmitted, a new sequence is allocated on the receiving side.

Pointers

Pointers are translated into REFs. The data pointed to by the pointer is transmitted, and an allocation is done on the receiving side to hold the incoming data. If the transmitted pointer is NIL, then the receiver winds up with a NIL pointer. Pointers can thus be used to transmit recursive data structures, such as linked lists, but the stack frame is used for each pointer when marshalling or unmarshalling.

Strings

All strings are converted to ROPEs, regardless of max length.

Opaque

Opaque data declared as a fixed-length array is converted to a packed array of BYTE. Opaque data declared as a variable-length array is translated to a REF TEXT, regardless of max length.

3. An Example

The directory containing the source code for cedarrpcgen also contains an example client-server set of programs that employ cedarrpcgen. The SunRPC interface files for this example are:

fortytwo.x
fourtwo.x

fortyone.x

The client program that uses the RPC interface defined by these files is:

fortytwoClientMain.mesa

while the server program that implements the interface is:

fortytwoServerMain.mesa

To recompile and run this example do the steps listed below. You will obtain two programs: fortytwoClient and fortytwoServer. (You will also get cedarrpcgen recompiled if the local version in your directory is out-of-date.) *Note*: These two programs currently expect to be running on the same host (I haven't yet added a command-line parameter to the client to allow specification of a host for the server).

From a Unix command shell issue the commands

```
cedarrpcgen  -p -e fortytwo.x
cedarrpcgen  -p -e fourtwo.x
cedarrpcgen  -p -e fortyone.x
```

From a Cedar commandtool issue the command

```
makedo -df CedarRPCGen
```

From a Cedar commandtool issue the commands (assuming an -ux working directory)

```
pma /cedar/cedarrpcgen -ux:$(pwd)
fortytwoServer
```

From another Cedar commandtool issue the commands

```
pma /cedar/cedarrpcgen -ux:$(pwd)
fortytwoClient
```

Note: In PCedar you end up in the Cirio debugger because PCedar doesn't correctly implement the floating point case for RPC.

Sun RPC Runtime

Runtime support for Sun RPC on UDP and NetworkStreams

A. Demers

Abstract: As the name suggests, SunRPCRuntime is a collection of (client and server) runtime support routines for RPC using the Sun protocols. Clients may choose to create UDP handles through the SunRPCOnUDP interface or TCP handles (actually NetworkStream handles) through the SunRPCOnNetworkStream interface. The created handles are then passed to procedures in SunRPC, itself, to do the work.

Created by: A. Demers

Maintained by: A. Demers <Demers.pa>

Keywords: Interoperability, Sun, RPC, UDP, TCP, NetworkStream

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Impenetrable description

A number of Sun RPC remote programs are described in Sun documentation and elsewhere. We have client implementations for quite a few of them, and servers for a couple. They all have the following structure.

For remote program *Blort* there are Mesa interfaces *Blort.mesa*, *BlortClient.mesa*, and *BlortServer.mesa*. *Blort.mesa* consists of Mesa type definitions corresponding to the data types and procedures of *Blort*. *BlortClient.mesa* and *BlortServer.mesa*, which are identical except for their names, consist of procedure declarations using the types from *Blort.mesa*. Clients of *Blort* import from *BlortClient.mesa*; the server exports to *BlortServer.mesa*. A Mesa program *BlortClientStub.mesa* does serialization/deserialization on the client side. It exports to *BlortClient.mesa* and invokes the communication primitives defined in *SunRPC.mesa*. Another Mesa program, *BlortServerStub.mesa*, does serialization/deserialization on the server side. It registers itself with *SunRPCOnUDP.mesa* or *SunRPCOnNetworkStream* to be invoked when an RPC call is received, and imports from *BlortServer.mesa*.

This structure is a bit baroque, but please adhere to it. See Section 3 for info about automatically generating these files.

2. How to understand it, really ...

~~Look at an example. A reasonably straightforward client is the SunNFS client package on CedarChest. I don't yet have a good example of a server; there is a SunPMap server, but it's slightly nonstandard since it's the only SunRPC program that listens at a well-known UDP port.~~

Look at an example: the LocalRegistry package contains examples of both a client and a server using UDP transport.

3. cedarrpcgen (Marvin Theimer)

Look at /project/ubi/src/utils/cedarrpcgen/ (source) and /project/ubi/bin/cedarrpcgen (executable) for a Sun rpcgen-based stub generator using this runtime package.

4. A scheme stab at machine-generated stubs (Michael Plass)

Look at *RPCGenerate.scheme*, and Example.rpc. You may be able to figure it out. Or maybe not; be warned it needs some polishing to make it real. In particular, its outputs are partitioned and named differently than section 1 suggests; this inconsistency ought to be fixed.

Try this:

```
% Bringover -p /PCedar/Top/SunRPCRuntime-Source.df
Start: [PCedar]<Top>SunRPCRuntime-Source.df!7
[net]<palain>palain>plass>try>SunRPCRuntime-Source.df <--
[PCedar2.0]<Top>SunRPCRuntime-Source.df {11-May-90 15:54:00 PDT}
```

```
[net]<palain>palain>plass>try>Example.rpc <-- [PCedar2.0]<SunRPCRuntime>Example.rpc!1
{10-May-90 10:14:54 PDT}
[net]<palain>palain>plass>try>RPCGenerate.scheme <--
[PCedar2.0]<SunRPCRuntime>RPCGenerate.scheme!3 {11-May-90 14:35:45 PDT}
[net]<palain>palain>plass>try>RPCGenerate.$cheme <--
[PCedar2.0]<SunRPCRuntime>RPCGenerate.$cheme!1 {11-May-90 14:36:17 PDT}
End: [PCedar]<Top>SunRPCRuntime-Source.df!7
4 files acted upon
 { 20.75 sec }
% Scheme
(user) (load "RPCGenerate")
(cedarsunrpcgen)
(user) (cedarsunrpcgen "Example.rpc")
ExampleSunRPC.mesa ... ExampleXDRImpl.mesa ... ExampleSunRPCClientImpl.mesa ...
ExampleSunRPCServerImpl.mesa ... #!unspecified
(user) (quit)
 { 43.19 sec }
%
```

The Cirio Debugger

a multi-language, multi-target-world debugger

Howard Sturgis and Marvin Theimer

Abstract: Cirio is a debugger intended to support the debugging of systems written in multiple programming languages and running on multiple target machines. Currently Cirio understands the Cedar and C programming languages and understands about programs run on D-machines or in PCR-based systems. The current version is still very much a prototype, with much of its functionality missing.

Created by: Howard Sturgis and Marvin Theimer

Maintained by: Howard Sturgis <Sturgis.pa> and Marvin Theimer <Theimer.pa>

Keywords: debugging

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Cirio News

June 2, 1992

(1) Changed Cirio's C type parsing protocol (in RCTW and ObjectFiles) to return a CirioTypes.Type (rather than a "DotOType" string).

(2) This allows Cirio to accomodate different C type grammars.

April 27, 1992

(1) Fixed the bug introduced last time around.

(2) Made another small improvement in the handling of gcc output.

April, 1992

(1) Made some changes to the handling of Sun a.out files so that gcc results as well as Sun cc results can be interpreted. Also improved the handling of Sun cc output. Also introduced a bug.

(2) Added a rudimentary attempt at finding C source files. Adding directories to the global search list helps.

March 30, 1992

(1) Workaround for the following AIX ld bug installed in XCOFFFiles.mesa.

On AIX (both 3.1.5 & 3.2), it seems static symbols are not relocated by ld. This causes Cirio to be unable to find the correct "versionStamp" address for cinded object files. Single object files work fine.

(2) Performance improvement for Cirio's inital connection time found and implemented by Carl Hauser. It seems to be about twice as fast now.

(3) Minor change to hot frame detection. It should now do a better job of predicting the hot frame for uncaught errors. It sure would be nice if PCR could provide this information.

March 6, 1992

RS6000 debugging works.

March 5, 1992

New Features:

(1) Opaque types can now be "seen" remotely. This functionality requires support in CedarCore and new DebugNub functionality.

(2) Some RS6000 remote debugging capability. All of the following functions are working:

Connection manipulation

connect to /disconnect from an RS6000 pcr world
stop / resume / kill execution of an RS6000 pcr world

Thread manipulation
select (add) threads by filter (callingdebugger, ready, all)
list available threads
get thread summary
detail thread
proceed, abort, etc. thread

Frame manipulation
walk stack
show source position
show frame
expression evaluation including ampersand procedures, GLOBALVARS and EXPRTYPE

Basically everything except breakpoints is working. Breakpoints should be finished shortly.

User Interface Changes:

(3) [Viewers]
Added Find and Word buttons to CirioRemote Viewers interface.

(4) [CommandLine]
Improved CommandLineCirio to give a helpful message when a command is issued out of context.

(5) [CommandLine]
Changed the Walkstack command in CommandLineCirio to accept "Co" for "Cooler" and "W" for "Warmer". The syntax is now:

WalkStack [[Co[oler] | W[armer] [NumberOfFrames]] | AbsoluteFrameNumber] [C | Cedar]

Bug Fixes:

(6) Timeout errors during attempts to StopRemoteWorld now do not cause a Cirio crash.

February 13, 1992

(1) Added another &-proc:
&RdXStringBody: PROC [addr: CARD, char-count: CARD];

&RdXString couldn't be used when only an XStringBody was available.

Also fixed a problem with the implementation.

(2) Fixed memory dump procedures. The last release only printed 3 out of every 4 words.

(3) NIL answers to KillRemoteWorld verification in CommandLineCirio are now handled correctly.

(4) CirioHelp command in CommandLine cirio only displays currently available commands, instead of all CommandLineCirioCommands. If "all" is passed in as an argument, all commands are listed.

(5) New argument ("listed") accepted for Kill, Freeze, Proceed, Abort, and Summary commands in CommandLineCirio. When the new argument is supplied, the invoked command does its operation on all threads that have been added.

(6) XCOFF stab reader now parses stab strings better, and source positions can be determined.

February 3, 1992

(1) Cirio now understands how to do pointer comparisons (=, #, <, >, <=, >=), e.g.

```
&1 ← @r
 105F5588H@
&2 ← @s
 105F558CH@
&3 ← @s=@r
 FALSE
&4 ← @s=@r+4
 TRUE
&5 ← @s#@r+4
 FALSE
&6 ← @s<@r+4
 FALSE
&7 ← @s>=@r+4
 TRUE
```

(2) Corrected thread id displaying - it sometimes displayed the wrong number even though it did the right operation.

January 29, 1992

(1) Attempts to print unbound variant records now result in an error message - not a crash.

(2) Attempts to print sequences with integer indices now succeed.

(3) Corrected CommandLineCirio's treatment of thread indices which caused confusion when threads numbers were not contiguous.

January 23, 1992

(1) Added three more &-procs:
&RdMem: PROC [addr: CARD, bytes: INTEGER, base: CARD];

&IndirectRdMem: PROC [addr: CARD, bytes: INTEGER, base: CARD];

&RdXString: PROC [XString-var-byte-address: CARD, char-count: CARD];

(2) An error is raised when trying to default variables for AmpersandProcs when no procedure with this functionality exists. This condition was not previously checked for.

January 17, 1992

Moved SameBreakWorld* into CirioBase to eliminate the need to recompile Druid to track Cirio changes.

January 16, 1992

(1) Cirio's file location searching order has been modified. Cirio now searches:
- (1) in the user supplied and default working directories
- (2) in the version maps
- (3) in the locations found in the object files
- (4) in other locations it stumbles across.

(2) A new default working directory has been included in Cirio - "-compiled:/CirioCompiledData/". This internal working directory contains the mobs that Cirio needs to make a connection. This makes Cirio startup slightly faster and makes Cirio significantly easier to distribute to sites without PCedar releases.

January 10, 1992

(1) Types and values are now printed on streams, instead of converted to ropes. StructuredStreams are used to format the output nicely with whitespace.

(2) Opaque types are now seen through, in the local case.

(3) Bound COMPUTED and OVERLAID variant record types now work correctly. (Bound non-COMPUTED non-OVERLAID variant record types are now slightly wrong --- the tag is missing).

(4) Most non-understood typestring contructs now produce broken types, rather than invalidating the whole typestring analysis. Array and procedure constructions are now understood.

January 6, 1992

(1) New Command: ChangeCirioMessageLevel

Usage: ChangeCirioMessageLevel [level]

Valid levels:	urgent:	Only displays necessary messages
	normal:	Displays necessary and informative messages
	debug:	All messages available are displayed

If no level is supplied, then the current level is displayed.

Profile Entry:

Cirio.MessageLevel urgent | normal | debug

default is "normal"

There are still problems throughout Cirio with using this, because not all output goes through SystemInterface.ShowReport. Not all levels are guaranteed correct.

December 23, 1991

(1) Further internal reoganization in the inexorable march toward righteousness. Type graph cycles are now broken at REFs when analyzing mob data structures. Records from mobs continue to defer analysis of their fields, just for laziness' sake.

(2) Introduced some analysis of typestrings, applied in the case of REF ANYs. A REF ANY value now appears as a REF to the actual type of the referent, when debugging locally and the analysis of the typestring describing that type succeeds. The analysis will fail if an opaque or painted type from a module not in the $Intermediate version map is encountered. The analysis will also fail on various other unimplemented cases.

(3) Added two more &-procs:
&GetTypestring: PROC [code: CARD] RETURNS [string, whyNot: Rope.ROPE];

&GetConcreteTypecode: PROC [opaque: CARD] RETURNS [concrete: CARD, whyNot: Rope.ROPE];

December 13, 1991

(1) Breakpoints can now be set in C code. Because C doesn't have any general way of testing whether a given object file was compiled from a given version of a source file (unlike Mesa, which records the source file's create date in the object file), the breakpoint-setting operation always works on the most recently loaded object code compiled from the indicated source.

(2) Fixed a swarm of bugs introduced in the October upheaval in connection with multi-word procedure parameters.

December 6, 1991

Fixed two bugs:
(1) A small bug in "CirioRemote" argument parsing.
(2) A bug (#2869) in ObjectFile parsing. The problem would show itself with a message like:
"failure due to internal Cirio error: redefinition of dotOType (0.282)"

November 27, 1991

The CirioRemote command will now accept an initial list of working directories on the command line. This feature is useful if you are trying to debug a world that uses different versions of basic files, e.g. Rope.mob. The syntax is:

CirioRemote <hostname> [<portnumber> [<listOfWorkingDirectories>]]

For example:

CirioRemote swift 4815 /NewCedarFiles /MyTestFiles /MyFriendsTestFiles

October 17, 1991

During the past few months, Cirio has been undergoing some major changes. The goal of these changes is to make Cirio a multi-target debugger - specifically to support XSoft's product goals.

Cirio's treatment of object files, stacks, memory, breakpoints, etc. was revised so that several target-specific implementations could co-exist peacefully.

The SPARC/SunOS4* target should be functionally equivalent to the previous version of Cirio.

The RS6000/XCOFF target is still undergoing changes and is not yet ready for use.

August 17, 1991

(1) Made both local and remote connections start out with this list of directories to search for files: /PCedar2.0/Cirio/, /PCedar2.0/CirioThings/, /PCedar2.0/Atom/, /PCedar2.0/Rope/. This makes the navel-examination done at startup succeed even in the absense of version maps. The files needed are currently: CirioThings/CirioRopeHelper.mob, Cirio/CCTypesImpl.mob, Cirio/CirioTypes.mob, Cirio/NewAmpersandProcs.mob, Cirio/NewAmpersandProcsImpl.mob, Rope/Rope.mob, and Atom/AtomPrivate.mob. For C interpretation, the file /CedarCommon2.0/FamousFiles/c.kipperedParseTables is also needed.

August 9, 1991

(1) 'CirioRemote localhost' should now work reasonably on a machine that's not running NIS (nee YP) and an automounter.

August 8, 1991

(1) Added machine language to those barely understood by Cirio. Cirio cannot interpret any machine-language expressions or statements. But it can print machine-language frames. Such a printout lists the PC, SP, and the input, local, and output registers. When used in conjunction with the disassembler in SparcAids, and the memory dumping available in the Cedar language, this might be just enough to do some debugging of optimized code.

(2) Made the 'frame banner' (what's printed as you WalkStack into a frame) more informative: if the frame is in optimized code, the code base address and module offset are now printed; they were already being printed when the fram is in unoptimized code. The code base address is where the code of the containing object file starts in memory; the module offset is where the code of the compilation unit responsible for the frame in question starts within all the code of the containing object file.

July 12, 1991

(1) Cirio now catches the error that occurs when it interprets an expression that tries to divide by zero.

(2) When the remote world goes away, Cirio now stores that information so that it doesn't try further interactions. The breakcheckdaemon already noticed when the remote world went away, but didn't record this fact in any data structure, it only stopped its own train of action.

July 3, 1991

(1) New Command added

←
Usage: ← <expression>

(2) Fixed procedure called after QuitWorld is called so that it wouldn't try outputting to a non-existent viewer.

(3) Increased the intelligence of hot frame detection. Added some cases to the check procedure for finding hot frames.

July 2, 1991

(1) Cirio now prints less information about a procedure when the printing depth is less than 3.

(2) Cirio now prints some symbolic information about a procedure implemented in an optimized module. When the printing depth is deep (>2), such a procedure is printed as "*loadedFile.guessedComponent.cProcName*"; *loadedFile* is the UNIX name of a file that UNIX ran (such as "/tmp/CommanderWorldPackage") or the PCR incremental loader loaded (such as "/pseudo/pcedar2.0/crrpc/sun4-o3/crrpcimpl.c2c.o.~4~"); *guessedComponent* is a guess at the UNIX file name of the compilation unit (such as "IOCommonImpl.c2c.o") within the loaded file; and *cProcName* is the C-level name of the procedure (such as "←DefaultEndOf←P960"). When the printing depth is shallow, the *loadedFile* is omitted.

(3) Added another ampersand procedure to give access to the incremental loader in the debuggee (ie. not for casual users).

&GetFileEntry: PROC [seqNum: CARD] RETURNS [REF FileEntry];
This procedure provides direct access to a CirioNub procedure.

June 8, 1991

(1) Cirio should now find frame extensions and global links (which I think it doesn't use) that are misplaced by Mimosa into the NIL context. In user-ese, that means Cirio is now able to determine the addresses of more local variables than before.

(2) Cirio now knows where Mimosa hides the variables local to an ENABLE scope.

June 5, 1991

(1) Added the 'CirioBugTreatment' command, which offers the option to change how Cirio reacts to its own uncaught errors (some of which are due to bugs, and some of which are due to erroneous input). The command takes one optional argument, indicating the new treatment: 'catch', the old and default behavior, means to catch and report the error; 'reject' means to not catch the error; and 'oz' means to first call XR←CallDebugger (which punts directly to the debug nub for remote debugging) and then catch and report the error. When invoked with no arguments, 'CirioBugTreatment' simply reports the current state.

June 4, 1991

(1) Major internal re-organization. There is no longer a distinction between type-time and structure-time --- at least for the Cedar part; the C part has been left working the old way, so this revision of Cirio can be brought to a timely conclusion.

(2) The static links out of procedures with no local variables now can be seen. The static links out of ENABLE scopes now can be seen.

(3) Data whose size is not a multiple of 32 bits now are correctly extracted from REF targets and local frames.

(4) Variant records now print without crashing.

(5) Broken remote-Cirio tools can now be deleted.

May 5, 1991

(1) Made a few changes to the way Stack Summaries and Frames are presented:

(1.1) Cirio now attempts to determine the first "interesting frame" and begin the summary with that frame. In general, the "interesting frame" will be the one that caused the debugger to be invoked, raised an error, etc. [The "interesting frame" is determined heuristically since pcr doesn't currently save this information. Hopefully the next version of pcr will be able to supply it.]

The "interesting frame" is indicated by the characters "=>" displayed in the left margin. If you select another frame by "walking the stack", the selected frame is indicated by the characters "->" displayed in the left margin. The only way to see a frame "hotter" than the "interesting frame" is to "walk the stack" to select it.

The summary will start with the "hotter" of the selected frame or the "interesting frame". For example:

```
quick summary for thread with index 1
          debug message = ClientRequest
     ->   6: ←CallDebugger←P60 (pc=88H, from CirioThingsImpl.c2c.o)
     LoadStateAccessImpl reports no guessedEmbeddedFile for abs PC = 19326128
          7: unreadable frame
     =>   8: ←TestSort←P300 (pc=8C0H, from QuickSortTest.o)
          9: ←NoName←Q576 (pc=EFCH, from QuickSortTest.o)
```

```
10: ←XR←Enable (pc=5CH, from SignalsImpl.c2c.o)
11: ←RunQuickSortTest←P360 (pc=A8CH, from QuickSortTest.o)
12: ←QuickSortTest←P0 (pc=208H, from QuickSortTest.o)
13: ←DoStartProgram←P3180 (pc=44C0H, from InstallationScopesImpl.c2c.o)
14: ←DoStartConfig←P3120 (pc=4424H, from InstallationScopesImpl.c2c.o)
15: ←XR←StartCedarModule (pc=4814H, from InstallationScopesImpl.c2c.o)
16: ←XR←run←QuickSortTestConfig (pc=118H, from QuickSortTestConfig or QuickSortTestConfig.o)
....
```

(1.2) When detailing a thread, the "interesting frame" is automatically selected - you don't need to "walk the stack" to select it prior to doing operations like SourcePosition, ShowFrame, etc.

(1.3) ShowFrame now includes the address of the frame in the information displayed about the frame. (This address can be used to directly access the frame when Cirio doesn't tell you what you wanted to know.) The new output looks like this:

```
Showing frame:
(frame address: 1461888)
""
<node of unknown type (frame's procedures not implemented)>
        Arguments--
                ↑[min:10, max:19]
        Results--
                ↑[]
        Variables--

---Global Frame ommitted---
```

(2) Changed the "Listed threads:" control buttons interface slightly. The "Listed threads:" buttons now appear only if threads have been added via the "CallingDebugger", "Ready" or "All" threads selection buttons.

April 12, 1991

(1) Added a new set of buttons to the Viewer user interface to provide thread control for a set of "Listed threads:". These buttons are similar to the ones provided for individual thread control, but operate on ALL threads appearing in the threads list.

(2) Some useful (and perhaps previously undocumented) operators for use in expression evaluation:

'@Foo'   prints the address of Foo
'Foo?' prints the type of Foo
'Foo!' each '!' increases the print depth (by 1) and width (by 10) of Foo
'Foo$' turns on copious debugging output for the evaluation of Foo - this is not for the faint of heart!

March 20, 1991

(1) Added StackCirio.ClearBreakPointAtPosition to support specifying a "break to clear" by it's source file position (analagous to SetBreak). This functionality is currently only available from the CommandLine or dbxtool user interface - it is not yet available from the Viewers user interface.

(2) Fixed several address faults.

January 13, 1991

(1) Attention for breakpoint is implemented. When PCR world(Debugnub.o) get a SIGINT, an message "SIGINT: debuggee needs attention." will appear in RemoteCirio / CommandLineCirio. Latest PCR(3←4, dated Jan 14, 1991) is required.

(2) As the deamon is introduced for WaitSig(), the communication between Cirio and Debunnub.o becomes slow in RemoteWorld Running.(Each WaitSig() takes 2 second) But, usual debugging is done in RemoteWorld stopping. So it will affect only StopRemoteWorld/ResumeRemoteWorld.

December 13, 1990

(1) Atoms will print out (as '$"*pname*"' for an ATOM with '*pname*' as its print-name).

(2) Non-understood types will be printed as raw bits, if the length is known.

(3) Broken nodes and types should now include a description of the problem in their printing.

(4) ShowFrame omits the global frame (which is now accessible via GLOBALVARS). Unless the print depth is greater than 3, which is the default - and there's currently no way to override the default.

(5) The stack summaries come in two flavors: (1) with the embedded-module-relative PC, and (2) without. Flavor 1 is under the left mouse button on the summary screen button, and Flavor 2 is under the middle mouse button.

(6) The stack-specific buttons in a remote Cirio tool now include one, named '*', that offers all the functions of the others; these buttons have been PopUpButtons for a while now.

(7) You can now, in principle, take the address of any value with a runtime address. However, the result is always a POINTER TO ..., never a POINTER TO READONLY ..! Be careful not to store through such pointers! Some kinds of values, such as procedures in global frames, don't actually support taking their addresses.

(8) Cirio works a little better now when you load multiple copies and versions of a program. The remaining bug I know about is that the SetBreak operations are not sufficiently choosy about matching up a loaded program with the source indicated.

December 7, 1990

(1) 'GLOBALVARS[Foo]' denotes the global variables of the most recently loaded instance of module Foo. 'GLOBALVARS[Foo, 3]' denotes the global variables of the 4th most recently loaded instance of module Foo. The GLOBALVARS construct may appear on the left side of an assignment.

(2) 'EXPRTYPE[e]' denotes the statically apparent type of expression 'e'.

September 4, 1990

(1) Fixed PopUpDriver2 to call LocalCirio.ReleaseConnection where it should, resulting in faster creation of new local debug tools. Thanks to Carl Hauser for finding this problem and fix.

August 31, 1990

(1) The user now controls the language being interpreted. When Cirio's attention is focused on a particular stack, there's a 'Language' button that is used to control the language used in that stack.

(2) C source line numbers can now be reported; it still will not open a viewer for you.

(3) Conditional expressions (test?ifTrue:ifNot) now work. += and friends now work (but they evaluate the left argument twice).

(4) Procedure values now exist (although you still cannot call them).

(5) Adding integers to pointers now works. The "ptr[idx]" syntax works.

August 28, 1990

(1) C debugging gets better.

'Global' variables (static, extern, and common) should work now.

Assignment should work now.

(2) Opaque values should no longer cause address faults.

August 22, 1990

(1) C debugging begins to appear. Not available yet:

Anything involving C source locations (this includes anything involving breakpoints in C code).

'Global' variables (static, extern, and common). Note that this means ShowFrame will always fail.

Assignment.

Procedure values.

'Ampersand-variables' (and -procedures). I think the way to approach these is to make 'ampersand' be a logical concept, bound to one of underscore (←) or octothorpe (#) in C.

User control over which language is interpreted.

(2) The prompt tells you the language in which your expressions will be interpreted. A prompt that looks like '&42 ← ' indicates Cedar; a prompt like '/*_47 */ ' indicates C.

(3) Various other kinds of cleanup:

I now believe that variables used inside an ENABLE or ! but declared outside should be visible from frames outside the ENABLE or !. Actually, this has been the case for quite a while, but I was confused by other bugs that appeared when I tried this case.

A ROPE that was NIL used to cause a crash; it no longer does.

It's now possible to see SEQUENCE values.

(4) It's still so impossible to see an opaque value that you'll get an address fault if you try.

July 20, 1990

(1) Cirio now has some more ampersand procs, to make it possible to build ROPE parameters passable to the &Lookup.. procedures. This is an interim solution; there is a deeper bogon that needs to be annihilated. Don't try to get ROPEs made with these procedures into the target world (aka debugee), neither by passing them to target-world procedures nor by storing them into target-world data structures nor by any other method you can think of; I don't know what will happen if you try, but I know it won't be helpful. In just the same way, don't try to pass target-world ROPEs to the &Lookup.. or the ..LocalRope.. procedures. Doing these things could cause memory smashes!

&MakeLocalRope5: PROC [c1, c2, c3, c4, c5: CHAR] RETURNS [ROPE];
Makes a ROPE in the debugger (not debugee, aka target) containing up to 5 characters. The first space an following characters, if any, are not included in the result.

&MakeLocalRope10: PROC [c1, c2, c3, c4, c5, c6, c7, c8, c9, c10: CHAR] RETURNS [ROPE];
Makes a ROPE in the debugger (not debugee, aka target) containing up to 10 characters. The first space an following characters, if any, are not included in the result.

&LocalRopeConcat: PROC [base, rest: ROPE] RETURNS [ROPE];
Like Rope.Concat.

&LocalRopeCat: PROC [r1, r2, r3, r4, r5: ROPE] RETURNS [ROPE];
Like Rope.Cat.

&LocalRopeSubstr: PROC [base: ROPE, start, len: INT] RETURNS [ROPE];
Like Rope.Substr.

Spring 1990

(1) A few &Lookup.. procedures weren't documented. They follow. The &Lookup.. procedures are mainly useful for debugging the debugger; they provide lower-level access to loadstate information in the debuggee.

&LookupFileEntryByStemName: PROC [stemName: ROPE, numToSkip: INT] RETURNS [name: ROPE, type, value, size, fileSeqNum: CARD];
This procedure provides direct access to a CirioNub procedure.

&LookupSymEntryByName: PROC [sym: ROPE, caseSensitive: BOOLEAN, externOnly: BOOLEAN, numToSkip: INT] RETURNS [name: ROPE, type, value, size, fileSeqNum: CARD];
This procedure provides direct access to a CirioNub procedure.

June 8, 1990

(0) Cirio now understands ROPEs as such (and Rope.Texts, too). Don't be discouraged by all the things you can't do with them --- you can't do those things with REFs either.

(1) Cirio now has more ampersand procs.

&MemDump: PROC [addr: CARD, bytes: INTEGER ← 16];
The given address is rounded down to the nearest word boundary. Prints, in hex and as characters, the contents of the indicated bytes in the target world. Until the debugee CirioNub is improved, don't give invalid addresses. Note that when debugging from *PCedar, default values don't work (yet).*

&IndirectMemDump: PROC [addr: CARD, bytes: INTEGER ← 16];
Like &MemDump, but first reads the address to start dumping from the word at the given address (which is first rounded down to the nearest word boundary).

&PokeCard: PROC [addr, val: CARD, mask: CARD ← 0FFFFFFFFH];
The given address is rounded down to the nearest word boundary. Alters a 32-bit word, or a subset thereof, in the target world. The value and mask are treated as CARDs: the 2^i's bit of the val and mask affect the 2^i's bit of the word-interpreted-as-a-CARD at the rounded address. Until the debugee CirioNub is improved, don't give invalid addresses.

(2) It is now possible to take the address (with @) of many kinds of varaibles.

(3) It is now possible to assign to the referent of a REF <some numeric type>.

(4) When cross-debugging (from D to P), correct byte orders are now used.

(5) The version map usage has been slightly changed (hopefully for the better). In PCedar, you want the following profile entries:
VersionMap.SourceMaps: [PCedar2.0]<VersionMap>PCedarSource.VersionMap
VersionMap.IntermediateMaps:
[PCedar2.0]<VersionMap>PCedarIntermediate.VersionMap

VersionMap.ExecutableMaps:
[PCedar2.0]<VersionMap>PCedarSparcExecutable.VersionMap
[PCedar2.0]<VersionMap>PCedarSparcOptExecutable.VersionMap

In DCedar. you need the same entries, except that the names are **VersionMap.PSourceMaps**. **VersionMap.PIntermediateMaps**, and **VersionMap.PExecutableMaps**, respectively.

April 17, 1990 2:31:02 pm PDT

(0) When a Cirio tool starts up. it spends an initial period examining its navel" in order to construct the ampersand procedures. This period may include several messages about its activities. The period is completed when the tool prints its date and time, followed by the "&1 ← " prompt.

WARNING: until certain low level locks are repaired, one should avoid loading any packages into the PCedar world until a tool completes its initial period.

(1) Cirio now has ampersand vars. A Cirio prompt now has the form: "&n ← ", where n starts at 1 and increases for each expression evaluated. After evaluation, &n will continue to hold the result of the evaluation. Further, one can name ones own ampersand vars, e.g., one might type "&foo ← exp".

(2) Cirio now has a small collection of ampersand procs. These are procedures that are part of Cirio which can be invoked through expressions typed into the typescript.

(a) &dr[byte-address, word-count]

Accepts two 32 bit cardinal values. Prints the decimal value of word-count 32-bit words, starting at byte-address.

(b) &RdRope[rope-var-byte-address, char-count]

Accepts two 32 bit cardinal values. The first is the address of a rope variable (not the rope). The second is the number of characters to print. Handles ropes built up from components. (Reads the rope structure and converts to a sequence of characters.)

As an example of use, assume that rope is a visible variable of type Rope.Rope.

First, evaluate the expression "@rope". This will proceduce a byte-address of the rope variable, for example, 2345678B.

Second, evaluate "&RdRope[2345678B, 100]".

(c) &LookupMatchingSymEntryByValue[symId: CARD, val: CARD, wantedTypes: CARD, ignoreClasses: CARD, numToSkip: INT]

This procedure provides direct access to a CirioNub procedure. It's behavior is a teensy bit complicated. See me or Alan.

A typical call might be:

&LookupMatchingSymEntryByValue[0, 12345, 37777777777B, 0, 0]

(d) &LookupMatchingSymEntryByName[symId: CARD, pattern: Rope.ROPE, caseSensitive: BOOLEAN, wantedTypes: CARD, ignoreClasses: CARD, numToSkip: INT]

This procedure provides direct access to a CirioNub procedure. It's behavior is a teensy bit complicated. See me or Alan. (This procedure is not usable on the Sun until we get Rope.ROPE procedure parameters working.)

March 21, 1990 11:12:31 am PST

It has become harder to find mobs and other files both remotely from a D-Machine and locally from a Sun. The difficulties have to do with PFS naming conventions.

work-around

Use AddDir to add the full path name of the directory from which the file was executed. BE SURE to use bracket syntax, but DO NOT include "-vux:" or "-ux:". (In order to avoid using "-vux:" or "-ux:", you can make an appropriate prefix map entry.)

Introduction

The Grand and Glorious Goal

To debug anything, anywhere, anytime, anyhow.

In practice, this probably means support for the PCR "world": being able to debug a program running in a PCR world, irrespective of what languages it's parts were written and what machine that world happens to be running on. For example, if a Scheme application has called a Cedar library package, which in turn makes use of a C procedure, then you should be able to look at the call stacks for the application and see the call frame of each procedure shown in a form appropriate to the language it was written in. You should also be able to utter expressions to the debugger for each call frame in the language in which the call frame's procedure was written in.

Specifically, Cirio allows one to examine and modify the current state of some target world, from the point of view of some context in that world. Typically, one does this by evaluating expressions in some language appropriate to the current context. These expressions can include the invocation of procedures which in turn may instigate fairly complex changes in the target world. One may also plant and remove breakpoints in the target world. If execution in a target world thread encounters a breakpoint, that thread will discontinue execution and may be used as a context for debugging.

Cirio has been designed so that (in theory) adding new languages and new target-worlds is easy. This has been done by employing an object-oriented approach for the architecture, so that the details of each language and each target world are hidden to the extent possible.

The Current State of Affairs

Cirio currently only understands the Cedar programming language, although it has handled C in the past and will probably do so again. We will restrict this document to describing the use of

Cirio to debug PCR worlds.

Cirio is generally invoked through a tool. Versions of this tool can be used to debug the same world as that in which it is running, other versions can be used to debug a remote world.

(1) There is a same-world version of the tool that runs in PCedar on Suns, providing debugging access to that PCedar world.

(2) There is a remote version of the tool that runs in PCedar on Suns, providing debugging access to other PCR worlds on the same Sun or on remote Suns.

(3) There is a remote version of the tool that runs on D-Machines, providing debugging access to PCR worlds on remote Suns.

What You Need to Know About Cirio

To use Cirio you need to know/do several things. A summary is given here; each item is described in more detail later on.

You need to install and run an appropriate Cirio debug tool.

You need to know how to operate the Cirio debug tools.

You need to ensure that the PCR world you wish to debug remotely includes the necessary Cirio support code as a part of it.

You need to understand the current limitations of the Cirio tools.

Same-World debugging

Obtaining a Same-World Cirio debug tool

Same-World debugging is by far the simplest mode of operation. The same-world Cirio debug tool provides debugging access to a single PCR thread. The tool may be created in response to an unforseen error (uncaught SIGNAL or ERROR) or upon encountering a previously set breakpoint.

To prepare for this mode of operation, one simply executes the following command:

CirioLocal

This will load the necessary support code for Cirio as well as register Cirio as the local debugger. Following this command one may obtain a Cirio tool in either of two ways. (1) If some thread encounters a previously set breakpoint or if some thread not running under the direct control of the CedarCommander generates an uncaught SIGNAL or ERROR, then an instance of the same-world Cirio tool will "pop-up" in the left hand viewer column. This tool will have as its context the thread that encountered the breakpoint or generated the uncaught SIGNAL or ERROR. (2) If a thread running under the direct control of the CedarCommander generates an uncaught SIGNAL or ERROR, then the Commander reports the situation as follows:

```
*** Uncaught ERROR or SIGNAL: unrecognized error
*** Do you want to try to debug this?
```

Typing "y" followed by a carriage return will cause an instance of the same-world Cirio tool to "pop-up" in the left hand viewer column. This tool will have as its context the thread that generated the uncaught SIGNAL or ERROR.

In addition, one may also obtain an instance of the same-world Cirio tool by executing the following command:

    Interpreter

This version of the tool permits the setting and clearing of breakpoints, but has no thread context to debug. (Executing this command also performs the preparatory functions provided by executing CirioLocal.)

Using a Same-World Cirio debug tool

The same-world Cirio debug tool provides debugging access to a single PCR thread. The tool viewer is divided into two windows. The top window provides a collection of buttons for invoking specific actions and the lower window contains a type script. Cirio uses the typescript to issue reports on actions in progress, as well as accepting expressions from the user for evaluation.

Top row of buttons in the upper window

SetBreak:

If the user left-clicks this button while there is a current selection in some Mesa file, then Cirio will endeavor to find a loaded instance of that file and plant a breakpoint at the beginning of the smallest statement including the first character of the selection.

ListBreaks:

If the user left-clicks this button, then Cirio provides a list (in the typescript window) of the current breakpoints. (Unfortunately, at present time this list does not include the source position of the breakpoints.)

ClearBreak(s):

If the user left-clicks this button while the current stack frame under examination (see the WalkStack button described below) is at a breakpoint, then Cirio will clear that breakpoint.

If the user middle-clicks this button, then Cirio will clear all current breakpoints.

AddDir:

If the user left-clicks this button while there is a current selection containing a directory path name, then Cirio will enter that path name its collection of possible locations for finding files.

FlushCache:

If the user left-clicks this button, then Cirio will forget the fact that it has been unable to find certain files. This is useful if one has just moved a previously unfound file into a directory in which Cirio might look for it.

Stop:

Terminates the Summary list operation. (Perhaps someday it will terminate other operations.)

Second row of buttons in the upper window

frame: n:

This is not a button, but rather a status field showing the index of the current frame in the thread under examination. Hot frames have smaller index numbers than cold frames. The hottest visible frame has index 0.

Summary:

Produces (in the typescript window) a list of the frames in the thread under examination, hottest frame first.

WalkStack:

Moves the focal point within the frame currently under examination. The semantics are a tiny bit complicated, but left click moves one frame colder and right click moves one frame hotter. In more detail, the action depends on which mouse button is clicked, as well as whether the shift or control keys are down, as follows:

(a) the mouse buttons mean:
- left: move to cooler frame
- middle: move to specified frame
- right: move to hotter frame

(b) the shift key means: read current selection for a number
(c) the control key means: work in terms of Cedar frames

Thus, for example, shift-middle with 5 selected means move to the frame with index 5.

One should be aware that working in terms of Cedar frames is fairly expensive. Further, one should observe that not all combinations of mouse buttons and keys make sense.

ShowFrame:

If the user left-clicks this button, then Cirio displays the contents of the variables visible to the frame currently under examination. This includes the variables in all enclosing blocks, including the global frame.

At the moment this operation is fairly slow. This is due to some injudicious mechanisms used for displaying the value of procedure constants, together with the fact that the global frame typically contains many procedure constants.

SourcePosition:

If the user left-clicks this button, then Cirio attempts to locate the Mesa file containing the procedure executing in the frame currently under examination. If the file can be found, and if various other files can also be found, then Cirio opens a viewer on the Mesa file. After opening the file, Cirio places a feedback selection at the beginning of the smallest statement containing the current pc of the frame.

Proceed:

Cirio returns control to the procedure that invoked the debugger.

Abort:

Cirio raises ABORTED in the thread under examination.

typescript window

Cirio uses this window to report actions to the user and to accept expressions for evaluation from the user. The reports are self explanatory.

It is the intention that if one types an expression (followed by a carriage return) in a language appropriate to the frame currently under examination, then Cirio will attempt to evaluate that expression in the context of that frame. For example, if one types the name of a variable visible to the frame currently under examination, then Cirio will print out the current value of that variable. Since assignments are expressions in Cedar, if Cedar is an appropriate language, then one can change the current values of variables by typing appropriate assignment expressions. Further, procedure invocations are also expressions, so one can invoke (visible) procedures by typing the appropriate expressions.

At the moment, there are numerous limitations on this facility. Cirio understands only Cedar. Cirio does not understand all Cedar value types. (In particular, this includes REF ANY and OPAQUE. Further, Cirio's understanding of Rope.ROPE is very limited.) Cirio is frequently unable to find variables that it knows exist.

If Cirio does not understand a value type, it prints "???". If Cirio does not understand how to find a variable, it prints "Node?".

Using a Cirio Same-World Interpreter tool

The interpreter tool (obtained by executing Interpreter) is very similar to the Same-World debugger tool, except that this tool has no thread under examination. Thus, only the first row of buttons is available, but they behave exactly as described above. The typescript window also behaves exactly as described above; however, there are no visible variables to use in expressions.

Remote Debugging

Preparing a Remote PCR world to be remote debugged

One must anticipate the future desire to remote debug a PCR world at creation time.

(1) The PCR world must contain a Cirio Nub running in a slave IOP.
(2) This nub will be connected to some port.
(3) The PCR world must have certain auxiliary packages loaded.
(4) You must know the port number.

The easy way

Items (1), (2), and (3) will be accomplished automatically if you start the PCR world with the shell command:

CedarCommander

In this case, if there are no other PCR worlds on the same Sun and you have not set the shell envirionment variable: CirioPort, then the port number will be 4815. ("for 815", for those of you who have done remote debugging in the old D-Cedar world. This number is complements of Al Demers.)

If there are other PCR worlds on the same Sun, then the Cirio Nub will select the first unused port number that it finds, trying the numbers 4815, 4816, ... in succession.

temporary problem

At the moment, CedarCommander does not load the neccessary auxiliary packages. These will be loaded if you execute "CirioLocal" in a command tool, or if you include the following in your CommandTool.BootCommands user profile entry:

```
run /PCedar/CirioThings/CirioThingsImpl
run /PCedar/CirioThings/RegisterSaveRestore.o
```

Item (4): You must know the port number.

There are four ways to learn your port number:

(a) Know that if you have started the easy way, there are no other PCR worlds on the same Sun, and you have not set the shell envirionment variable: CirioPort, then the port number will be 4815.

(b) It appears in the pcr type script in the following form

```
pcr: CallAll ←CirioNubStart
CirioNubInstall v 6 ... returns 4815
```

(c) If your are in a viewers world, Execute "CirioPortButton" in some command tool, before you perform any operation that invokes the same-world or remote Cirio debugger (such as Interpreter, CirioLocal, or CirioRemote). This will create a button in the upper right hand region of your Cedar viewers containing the Cirio Port number. If you execute "CirioPortButton" too late, the button will appear, but the number displayed will be (incorrectly) 0. If you execute this command without first have created a Cirio Nub running in a slave IOP, I don't know what will happen.

(d) Execute "ShowCirioPort" in some command tool, before you perform any operation that invokes the same-world Cirio debugger (such as Interpreter CirioLocal, or CirioRemote). This will print the port number in the typescript in which you executed ShowCirioPort. If you execute "CirioPortButton" too late, the number displayed will be (incorrectly) 0. If you execute this command without first have created a Cirio Nub running in a slave IOP, I don't know what will happen.

The hard way

If your prefer to roll your own, here are some helpful hints (your author is NOT an expert in these matters):

(1) The PCR world must contain a Cirio Nub running in a slave IOP.
  (a) "-slave 1"
  as a threads parameter to your pcr will provide a slave IOP.
  (b) UnixLoad /pseudo/xrhome/INSTALLED/LIB/Threads-sparc/DebugNub.o
  in your pcr script will load a CirioDebugNub.

(c) CallAll ←CirioNubStart
in your pcr script will will start the debug nub.

(2) This nub will be connected to some port; you must know the port number.

setenv CirioPort num

as a shell command will specify a specific port number forthe debug nub. (If this port is busy, the nub will fail to connect to a port, and will report -2 as its port number.) If you have not set this environment variable, then the debug nub will hunt for a usable port as described above.

In any case, the chosen port number will be reported as described above.

(3) The PCR world must have certain auxiliary packages loaded.

(a) You can load them with the following commands in your CommandTool.BootCommands user profile entry:

run /PCedar/CirioThings/CirioThingsImpl

run /PCedar/CirioThings/RegisterSaveRestore.o

(b) or you can load them by executing the following in a CommandTool:

CirioLocal

(c) It would be nice if you could load them by commands in your pcr script, then your would would be degbuggable "early on". However, because the auxiliary packages are in a versioned directory, the following DOES NOT work:

LoadAndRun /pseudo/pcedar2.0/CirioThings/CirioThingsImpl

LoadAndRun /pseudo/pcedar2.0/CirioThings/RegisterSaveRestore.o

Obtaining a Remote Cirio debug tool

In PCedar, execute the following command:

CirioRemote *machineName* [*portNumber*]

Where *machineName* is the name of the debuggee (or "localhost" if one intends to debug a different PCedar invocation on the same machine). Where *portNumber* is the number of the port to which the Cirio nub on the debuggee is connected. This number need not be supplied if it is "4815", the default. [Note Well: Do not attempt to apply a remote Cirio debug tool to the same PCR world in which it is running.]

On a D-machine, execute the following commands:

pushv commands

qbo -p [CedarChest7.0]<top>Cirio.df

pop

CirioRemote *machineName* [*portNumber*]

Using a Remote Cirio Debug tool

Killing the World and Killing Cirio

The most important thing to know about the remote Cirio debug tool is how to kill both it and the remote world you are trying to debug. You can kill the tool by clicking on the **Destroy** button in its viewer banner. This will destroy the debug tool, *but will leave the remote PCR world unchanged.* That is, if the remote world was stopped by Cirio, then killing Cirio will leave the remote world stopped. (As described below, you can reattach a new debug tool to your PCR world if you so desire.)

To cleanly kill the remote PCR world you can click the guarded button **KillRemoteWorld**. This will invoke the equivalent of PCedarCleanUp (without asking any questions).

Stopping/Starting the World

When the Cirio debug tool comes up, it has not yet done anything to the remote PCR world. You must click the **StopRemoteWorld** button to use Cirio's other features. This button is a toggle button the will either stop the entire PCR world or start it running again, depending on what state Cirio thinks it is currently in (the current state is displayed in the top, middle part of the viewer). If Cirio thinks the world is already stopped then the button will read **ResumeRemoteWorld** instead of **StopRemoteWorld**.

Reconnecting to the World

As mentioned above, you can destroy your Cirio debug tool while leaving the remote PCR world untouched (and possibly stopped). To reattach a new debug tool, simply reinvoke Cirio (i.e. issue the **CirioRemote** command) as you did before. This will bring up a new debug tool viewer and you can now continue debugging. **Note**: you cannot have more than one existing debug tool attached to a particular PCR world. If a debug tool is already attached, then a second tool that attempts to attach will simply hang, waiting for a response from the PCR world's Cirio support code. The response will not be provided until the first debug tool is destroyed.

The new debug tool will not remember the state of the old, destroyed one. Thus, it will come up thinking that the remote world is running and that there are no breakpoints set. The **StopRemoteWorld** button is idempotent, so you can click on it anyway to tell Cirio that you wish to debug something. Note that any old breakpoints that were set by the old incarnation of the debug tool will be unknown to the new incarnation. *I'm not sure what will happen if you try to set a breakpoint on top of an already existing one. Ask Peter Kessler.*

Controlling Threads

When Cirio thinks the remote world is stopped, it will display a new line of buttons, which will include **CallingDebugger**, **Ready**, and **All**. If some thread has tried to call the debugger, either by faulting or by hitting a breakpoint, then clicking the **CallingDebugger** button will show a "button" line that tells the thread number (and some other stuff) and provides a set of buttons to invoke against that thread. Clicking on **CallingDebugger** when none has called the debugger does nothing. Clicking on the **Ready** button will display "button" lines for all threads that are either ready or running. Clicking on the **All** button will display "button" lines for all threads.

Each "button" line contains information about the thread together with thread-specific buttons The information is as follows, from left to right:

The thread index (followed by a colon)

The thread priority

The thread scheduling state (e.g., Run, CVWait, MWWait, etc).

A Debugger message (e.g., None and CallingDebugger).

The buttons displayed for a thread can be used to control it as well as obtain information about it. The most important control button to know about is the **Prcd** button, which proceeds a thread that has called the debugger. You must click this button to get past a breakpoint (in addition to clicking the **ResumeRemoteWorld** button to get the whole world running again).

Thread-Specific Button Commands

Each thread button line contains two information buttons: **Quick** and **Detailed**. Clicking on **Quick** will generate a summary of the call stack for the thread, showing each procedure name and the file from which the procedure was loaded. For example,

```
quick summary for thread with index 0
    debug message = None
    ←XR←Switch  (from ThreadsSwitch.o)
    ←XR←Yield  (from Threads2.o)
    ←XR←Idle  (from Threads2.o)
    ←XR←ForkJumpProc  (from Threads1.o)
    ←XR←Jumpee  (from Threads1.o)
```

Clicking on **Detailed** provides access to each individual call frame of the thread's state. A new row of buttons is displayed and information is printed about the top-most call frame on the thread's call stack (designated frame **0**).

SetBreak etc

Whenever Cirio believes that the target world is stopped, Cirio displays a row of buttons including SetBreak. The buttons in this row behave exactly as do the corresponding buttons in a same-world Cirio tool.

Thread examination buttons

After clicking Detailed for some thread, Cirio displays a row of buttons including WalkStack. This is an abreviated set of buttons corresponding to those described for a same-world Cirio tool as "Second row of buttons in the upper window". From left to right we have:

frame: t.n

This is not a button, but rather a status field showing the index of the thread under examination together with the index of the current frame.

WalkStack

(Behaves exactly as for a same-world Cirio tool.)

ShowFrame

(Behaves exactly as for a same-world Cirio tool.)

SourcePosition

(Behaves exactly as for a same-world Cirio tool.)

Typescript window

(Behaves exactly as for a same-world Cirio tool.)

During Salient R1.0 and R1.1 development, there were several developers who wanted to write C code that ran inside the PCR world or wanted to use UNIX library routines. Due to the light-weight process scheme that PCR implements, there are significant restrictions on doing this. The following list describes the restrictions that we know about. The bottom line is that you should try at all costs to avoid using UNIX library routines and that you should carefully review this list of restrictions before writing C code that will run inside the PCR world (i.e. inside the GlobalView envirnoment).

1. Don't make UNIX I/O system calls. PCR wants to control all of the I/O done in GlobalView so that one thread does not block the whole PCR world. Likewise, don't make any shared memory calls or mapped file system calls. UnixSysCallTranslation.o provides a potential workaround, but as far as we know there aren't many users of it, so there may be some hidden issues with it. For example, there may be some tricks that have to be done when making a PCR package (such as XWSPackage for GlobalView) that contains UnixSysCallTranslation.o.

2. Be careful about using global (or static) variables in the C routine. Unless they are protected by a monitor lock, they will cause problems if more than one thread calls the C routine at about the same time. PCR provides some hooks to handle the global variable named "errno", but that is the only one.

3. Be careful about allocating memory using malloc and/or sbrk. PCR garbage collector needs to know about memory that is being allocated in the huge GlobalView address space, and it has its own allocation routines. There is a routine named malloc in PCR that is intended to be usable in the same way as C malloc, but it allocates collectible storage and GlobalView tries to avoid using collectible storage due to possible performance degradation when the garbage collector runs.

4. Don't use UNIX signals. The various UNIX processes that make up PCR uses SIGUSR1 to signal each other to perform various actions. Also, the garbage collector catches SIGBUS to detect writes to protected pages. And PCR uses SIGALRM to do time slicing amongst the threads. And there are probably other signal considerations as well. If you really must use UNIX signals, there is a limited capability in PCR that might provide the desired functionality.

5. Don't use UNIX library routines that do any of the things listed above. For example, printf() makes system calls; sleep() uses SIGALRM; scanf() has a static variable.

6. Be careful about writing slave IOP code that calls malloc(), since it will call the PCR malloc routine and PCR's garbage collector does not handle slave IOP memory.

7. Be careful about writing slave IOP code that calls UNIX library routines which contain global variables, since such variables will be shared between all the slave IOP's; sleep() is an example of a library routine that contains a global variable.

8. In order to prevent one thread from blocking the entire PCR world, PCR breaks up I/O requests into 16KB chunks. This might affect I/O done through special device drivers. (We encountered a problem with the Rank Xerox scanner driver in this area in Salient R1.1.)

9. Don't manipulate the "no delay" or "non-blocking" flags when using PCR's XR_IOCtl and XR_FCntl routines. PCR uses this flag internally as part of its I/O system.

Michael Plass, August 7, 1991 11:38 am PDT

UnixSysCalls

Carl Hauser, Alan Demers, Mark Weiser

Abstract: This is a Unix$^{(tm)}$ system call interface for PCedar. In order to access Unix system calls from PCedar, you MUST go through the UnixSysCalls interface. Every Unix system call in SunOS 4.0 Unix is mentioned in this interface, but some are mentioned only as comments and must not be called. Explanations for why things are *not* available in this interface, and prospects for future inclusion, are documented here. For more information about exactly what a syscall does in Unix, see the SunOS 4.0 Unix volume 2 documentation.

Created by: Carl Hauser, Alan Demers

Maintained by: Carl Hauser <chauser.pa>

Keywords: Unix, system calls

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

What's going on

As of March 30, 1989, the Portable Common Runtime, also known as PCR, underlies all uses of Mesa or Cedar on Sun workstations. Even though PCR can live on top of Unix, it does not allow free access through to Unix. PCR is like a little operating system in itself. When using PCR, you must use the PCR system calls. Not all Unix system calls are PCR system calls (and vice versa).

As a convenience for Cedar/Mesa programmers using PCR on Unix, a number of Unix system calls have been made accessible in the **UnixSysCalls** interface. The implementation of this interface is NOT to directly call the Unix kernel. Most routines are implemented via a fairly complex route, via PCR IOP's. It is never safe to call the Unix kernel directly from PCR or PCedar.

See **UnixSysCallExtensions** for some interesting PCR system calls that are not Unix system calls. See below for why not all Unix system calls are available in **UnixSysCalls**. See **UIO.h** for the C-language interfaces directly into the PCR Unix system call emulations.

General Principles for inclusion/exclusion in UnixSysCalls

Three principles were used in choosing to include or exclude a Unix system call in **UnixSysCalls**. The meta-principle is: include as little as possible.

Principle 1: If it is obsolete or can be emulated via other calls, don't include it (Ex: CREAT is obsolete, and can be emulated by OPEN.)

Principle 2: If it is available only to the super user, or affects only global Unix state, or is a brash and bold Unixism, don't include it. (Ex: MOUNT.)

Principle 3: If its use would interfere with the PCR kernel implementation, or simply will not work given the PCR implementation, don't include it, or implement only a non-interfering subset. (Ex: FLOCK cannot work with multiple VP's, only a subset of POLL is implemented.)

There is also a sub-principle, which is: some things are hard, and they are not done yet, but perhaps they won't be necessary for a while.

Principle 1 follows immediately from the meta-principle, and will not be explained further.

Principle 2 is more subtle, and stems from the philosophy that PCR, Cedar, and Mesa are sharing the machine with other programs. Therefore, it is not necessary that every system call be available--one always can execute a regular Unix program (via **UnixSysCallExtensions.Spawn**) to get at the others. Furthermore, it is undesirable to do things which affect the global Unix system state. System calls which affect or detect a property of a single Unix heavy-weight process are also generally not implemented, because PCR is a long-lived amalgam of multiple Unix processes and does not make sense to probe or alter the state of just one of those.

Finally, principle 3 reflects a bit of the PCR implementation showing through. PCR implemented on Unix reserves some Unix features for its own use (like passing around file descriptors via Unix domain sockets), because if others also try to use them it can get very confused.

blocking I/O. because the duped descriptor sometimes does. and sometimes doesn't. inherit the non-blocking properties.

EXECVE - reads a new executable image on top of an existing Unix heavy-weight process. PCR owns all the heavy-weight processes. cannot permit itself to be overwritten.

EXIT - terminates the current Unix heavy-weight process. PCR owns all the heavy-weight processes. cannot allow one to terminate.

FCNTL - The full properites of FCNTL would permit the user to thwart PCR's control of file descriptor properties which it needs for the IOP implementation. A subset of FCNTL semantics could be permitted. but is not now implemented or planned.

FLOCK - this locks a file for access by a single Unix heavy-weight process. Since PCR is multiple such processes. it could deadlock itself by letting one lock out others.

FORK. VFORK - makes a new Unix heavy-weight process. PCR owns all the heavy-weight processes. cannot permit a new one. However. see **UnixSysCallExtensions.Spawn** for a way to do the same thing from a light-weight process.

GETITIMER. SETITIMER - PCR uses the interval timer for itself. **Process.Pause**. and **XR←TicksSinceBoot** and timeouts on condition variables have a granularity of about 100ms and should be adequate for most uses.

IOCTL - this *is* in the interface. but use is extremely dangerous. only some uses will work. but no checking is made of those uses. Avoid IOCTL if at all possible. and then only use with permission of the PCR implementors.

~~MMAP. MUNMAP. MPROTECT - change virtual memory mapping. These affect only a single heavy-weight Unix process. but even worse. PCR uses mapping and unmapping for its own internal purposes. and needs to know who is doing what.~~

NFSSVC. ASYNCDAEMON - causes the current Unix heavy-weight process to dive forever into the kernel. PCR owns all the heavy-weight processes. doesn not like them diving away forever.

~~SHMCTL. SHMGET. SHMOP - these are the System V shared memory mapping primitives. These affect only a single heavy-weight Unix process. but even worse. PCR thinks it owns the map.~~

SOCKETPAIR - this one could be done. but hasn't been.

SYSCALL - this is an escape hatch to all the other calls. so cannot be permitted.

Workarounds

For workarounds. the following general rules apply: workarounds for calls excluded by principle 1 are specified in the descriptions below. For calls excluded by principle 2. the workaround is almost always to use **UnixSysCallExtensions.Spawn** to use a real Unix process. For calls excluded by principle 3. workarounds are very very hard. in some cases impossible. and should not be counted on.

2. Detailed explanation of unimplemented system calls

System calls not implemented because they are obsolete or can be emulated (principle 1)

CREAT - obsolete. see OPEN.

GETDIRENTRIES - obsolete. see GETDENTS.

RECVMSG - most uses can be emulated with RECV or RECVFROM. Use of RECVMSG to send or receive a Unix file descriptor is reserved to PCR itself.

SENDMSG - most uses can be emulated with SEND or SENDTO. Use of RECVMSG to send or receive a Unix file descriptor is reserved to PCR itself..

SELECT - can be emulated with POLL. Furthermore. is hard.

UNAME - can be emulated with **gethostname**.

System calls not implemented because they are Unixisms (principle 2)

ACCESS - depends on properties of a single Unix heavy-weight process.

ACCT - can only be executed by super user. and affects global Unix state.

ADJTIME - can only be executed by super user. and affects global Unix state.

AUDIT - can only be executed by super user. and affects global Unix state.

AUDITON - can only be executed by super user. and affects global Unix state.

AUDITSVC - can only be executed by super user. and affects global Unix state.

CHDIR - affects the properties of a single Unix heavy-weight process.

CHOWN. FCHOWN - can only be executed by super user.

CHROOT - can only be executed by super user.

GETAUID. SETAUID - can only be executed by super user.

GETPRIORITY. SETPRIORITY - samples and affects the properties of a single Unix heavy-weight process.

GETRLIMIT. SETRLIMIT - samples and affects the properties of a single Unix heavy-weight process.

~~GETRUSAGE - samples and affects the properties of a single Unix heavy-weight process.~~

SETGROUPS - can only be executed by super user. affects the properties of a single Unix heavy-weight process.

SETHOSTNAME - can only be executed by super user, affects global Unix state.

SETTIMEOFDAY - can only be executed by super user, affects global Unix state.

MOUNT, UNMOUNT - can only be executed by super user, affects global Unix state.

MSYNC - can only be executed by super user, affects global Unix state.

MSGCTL, MSGGET, MSGOP - these are the System V interprocess communication primitives. They are not needed within PCR tasks, where shared memory, monitors, and condition variables work much better. For communicating with other Unix processes outside PCR, we recommend FIFO files.

PIPE - this is the original Unix interprocess communication primitives. They are not needed within PCR tasks, where shared memory, monitors, and condition variables work much better. For communicating with other Unix processes outside PCR, we recommend FIFO files.

PTRACE - this monitors one Unix heavy-weight process from another.

QUOTACTL - can only be executed by super user, affects global Unix state.

REBOOT - can only be executed by super user, affects global Unix state.

SEMCTL, SEMGET, SEMOP - these are the System V semaphor primitives. They are not needed within PCR tasks, where shared memory, monitors, and condition variables work much better. Furthermore, blocking on a semaphor would block the entire PCR world. For communicating with other Unix processes outside PCR, we recommend FIFO files.

SETPGRP - affects the properties of a single Unix heavy-weight process.

SETREGID, SETREUID - samples and affects the properties of a single Unix heavy-weight process.

SETUSERAUDIT - can only be executed by super user, affects global Unix state.

SIGBLOCK, SIGPAUSE, SIGSETMASK, SIGSTACK, SIGVEC - these sample and control Unix signals, which are nothing like Cedar/Mesa signals. Unix signals make sense only in a single Unix heavy-weight process, and only one without threads.

SWAPON - can only be executed by super user, affects global Unix state.

UMASK - affects the properties of a single Unix heavy-weight process.

VADVISE - affects the virtual memory paging algorithm of a single Unix heavy-weight process.

VHANGUP - this releases a Unix control terminal. This should not have any meaning in PCR, so should not be necessary.

WAIT, WAIT3, WAIT4 - causes one Unix process to hang waiting for, or check the status of, another Unix process. See **UnixSysCallExtensions.Spawn** for how to do an implicit fork/wait safely from a PCR thread.

System calls not implemented because they will not work with PCR (principle 3)

BRK, SBRK - allocates storage in a single Unix heavy-weight process. All allocation must be via PCR allocation primitives.

DUP, DUP2 - duping a file descriptor can sometimes interfere with PCR's need to use non-