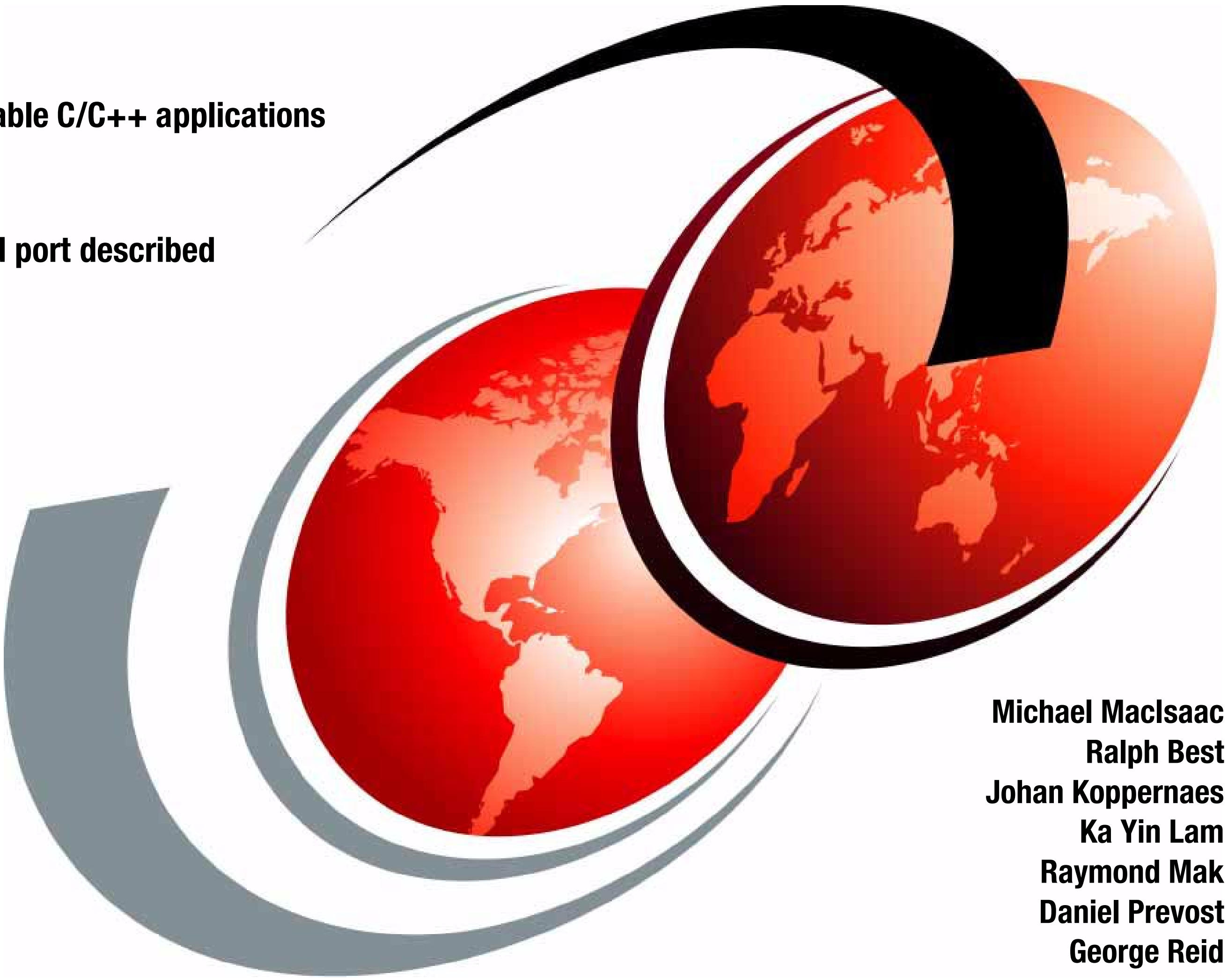
IBM

C/C++ Applications on z/OS and OS/390 UNIX

Port UNIX C/C++ applications to z/OS

Write portable C/C++ applications

Real-world port described



Michael MacIsaac
Ralph Best
Johan Koppernaes
Ka Yin Lam
Raymond Mak
Daniel Prevost
George Reid

Redbooks

ibm.com/redbooks

International Technical Support Organization

C/C++ Applications on z/OS and OS/390 UNIX

December 2001

SG24-5992-01

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 205.

Second Edition (December 2001)

This edition applies to version 1, release 2 of z/OS and earlier releases of OS/390, program number 5647-A01.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. HYJ Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Contents

Preface

In this IBM Redbook, we focus on how to move applications written in C/C++ from other UNIX operating systems to z/OS UNIX System Services. We highlight the traditional strengths of z/OS, and describe some of the subsystems not always found on other UNIX variations. We address application development tools, the C/C++ compiler, and open source development code such as the C++ Standard Template Library (STL) and the Adaptive Communication Environment (ACE). We also suggest some performance tuning techniques.

Finally, a "real world" port of a C/C++ application is detailed. First we describe the application itself, and then discuss the following aspects of the port:

Before	What we did to set up the application development environment
During	Which issues we encountered while porting to z/OS
After	How we tuned the application once it was running

In conclusion, we summarize our overall findings. Many additional appedixes are included for reference such as:

- A comparison of z/OS and GNU compilers and make tools
- OS/390 C/C++ compiler ASCII support
- STLPort information
- Discussion of dumps
- Performance analyzer output
- OS/390 UNIX Porting Guide - process management

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization Poughkeepsie Center.

Michael MacIsaac is a team leader for S/390 redbooks and workshops at the ITSO Poughkeepsie Center. He writes about and teaches classes on z/OS UNIX and Linux for S/390. Michael has worked at IBM for 13 years, mainly as a UNIX programmer.

Ralph Best is an instructor with IBM Learning Services. He has worked for IBM for 17 years. He has 24 years of data processing experience in applications and systems programming. Most of that time has been spent working with z/OS and MVS. For the past 5 years he has specialized in UNIX and UNIX-related applications.

Johan Koppernaes is a programmer with Data Kinetics Ltd. in Canada. He has 20 years of experience in C/C++ development. He is the team lead on the net.TABLES project. He has a Masters Degree in Mathematics and Computer Applications from the Technical University of Nova Scotia.

Ka Yin Lam is a compiler developer in the IBM Toronto Lab in Canada. He has worked on the z/OS C/C++ compiler team for more than 2 years. He holds a degree in Computer Science from the University of Toronto.

Raymond Mak is a compiler developer in the IBM Toronto Lab in Canada. He has worked on the z/OS C/C++ compiler team for more than 5 years. Before joining the compiler team, he worked in application development and has experience in MVS, UNIX, Oracle database and client server applications.

Daniel Prevost is a programmer with Data Kinetics Ltd. in Canada. He has 10 years of experience in C/C++ development.

George Reid is a programmer with Data Kinetics Ltd. in Canada. He has 20 years of experience in MVS. He holds a degree in mathematics from Dalhousie University.

Thanks to the following people for their contributions to this project:

Rich Conway, Dotti Still, Cindy Curley, Ken Trowell
International Technical Support Organization, Poughkeepsie Center

Al Devito, William Olders
Data Kinetics

Bob Haimowitz
International Technical Support Organization, Raleigh Center

Tom Klensk
IBM Endicott

Dr. Doug Schmidt, David Levine
ACE Project

Elizabeth Holland-Kern, Don Ault
IBM Poughkeepsie

David Cargill
IBM Essex Junction

Thanks also to Terry Barthel, Alison Chandler and Alfred Schwab for their editorial assistance and to Ella Buslovich for her graphics assistance.
International Technical Support Organization, Poughkeepsie Center

The initial proposal and subsequent funding for this project were provided by the IBM PartnerWorld for Developers, S/390 team located in Poughkeepsie, NY.

Special notice

This publication is intended to help UNIX programmers port C and C++ applications to z/OS UNIX. The information in this publication is not intended as the specification of any programming interfaces that are provided by the z/OS C/C++ compiler. See the PUBLICATIONS section of the IBM Programming Announcement for the z/OS C/C++ compiler for more information about what publications are considered to be product documentation.

IBM trademarks

Comments welcome

Your comments are important to us!

We want our IBM Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an Internet note to:

  redbook@us.ibm.com

- Mail your comments to the address on page ii.

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Summary of Changes
for SG24-5992-01
for C/C++ Applications on z/OS and OS/390 UNIX
as created or updated on December 20, 2001.

December 2001, Second Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

New information

- Description of the z/OS V1R2 C/C++ compiler which is now ISO C++ 1998 compliant - see Chapter 4, “z/OS C/C++ compiler” on page 57.
- Description of the Standard Template Library (STL) that is now included with the z/OS V1R2 C/C++ compiler - see Chapter 5, “Standard C++ Library” on page 83.
- Discussion of porting the package Xalan - see Chapter 13, “Porting Xalan-C++” on page 135.

Changed information

- “OS/390” replaced with “z/OS” in most places
- Appendix A, “Sample JCL Procedures” removed

1

z/OS and OS/390 tools

z/OS, formerly named OS/390, is the flagship operating system that runs on IBM's S/390 enterprise servers (after OS/390 V2R10 in September of 2000, z/OS V1R1 was released in March of 2001). S/390 servers store and support programs that process much of the data used by the world's largest corporations. The strength of z/OS lies in its ability to concurrently and reliably support thousands of users using many diverse applications.

Herein lies the major difference between the z/OS environment and most UNIX environments. UNIX environments usually support a single server, application, or related set of users. So if UNIX is supporting all three types of applications, three UNIX servers are used.

z/OS, on the other hand, rarely supports a single application or set of users. Rather, it might be supporting a mix of open and traditional z/OS applications. A single z/OS operating system image (copy of z/OS) could be supporting Web browsers, e-mail, transaction managers, database managers, TSO, and batch users.

z/OS provides an extensive set of tools and facilities to enable an installation to gain the maximum benefit from this rich and diverse environment. It provides a WorkLoad Manager (WLM) to allow classes of applications to be prioritized so that each user and application gets its fair share of z/OS resources. It allows a wide variety of System Management Facilities (SMF) records to be created for resource accounting, performance analysis, and auditing purposes.

z/OS also supports multiple layers of recovery routines and a Parallel Sysplex environment to provide essentially continuous availability. It provides a full set of Resource Measurement Facility (RMF) monitors and reports for analyzing the performance or capacity of a z/OS system.

z/OS provides a number of interfaces that users can use in requesting z/OS services. This rich and comprehensive set of interfaces and tools provides z/OS with the flexibility and versatility to match the multitude of applications that run under z/OS. This chapter provides a brief look at some of the most important tools, facilities, and applications in the z/OS environment. But first we discuss ways of accessing services on the z/OS system.

1.1 User access under z/OS

z/OS provides several classes of traditional operating system interfaces, but the two most important are *batch* and *TSO*:

- Batch is the main vehicle for invoking programs (binary executables) in z/OS.
- TSO provides an interactive command line interface to z/OS operating system services.

TSO in z/OS plays a role similar to the shell in UNIX. However, very few installations still use TSO. Most use its enhanced add-on called Time Sharing Option/Extended (TSO/E). Therefore, we will look at TSO/E, but let us first examine batch and the Job Control Language (JCL) environment.

1.1.1 JCL and batch

z/OS runs work in the background via batch job submission. The three most important JCL statements are:

- JOB statement
- EXEC statement
- DD statement

Any interesting job stream (job) in z/OS contains all three types of statements. Only the DD statement is optional, but you will probably never see a job without one. There are several types of datasets (*dataset* is the z/OS equivalent of a UNIX file), and numerous dataset characteristics in z/OS. Thus it is the DD statement that allows the large number of parameter and subparameter coding combinations. It is the most difficult statement to learn to code. We briefly describe all three types of statements.

1.1.2 JOB statement

The Job statement marks the beginning of the job. It identifies the user, specifies accounting information, and sets limits on the z/OS resources that can be consumed by the job.

1.1.3 EXEC statement

The EXEC statement identifies a job step and a program (or JCL procedure) that is to be invoked in that step. It also provides processing information for that step.

1.1.4 DD statement

The DD statement describes a dataset that will presumably be used by the program referred to on the EXEC statement. The DD statement identifies input and/or output resources for the dataset.

A sample job

Here is a sample job containing one statement of each type:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//ACT      EXEC PGM=IEFBR14
//SOUT DD DSN=RALPHB.STD.OUT,SPACE=(TRK,(1,1)),UNIT=SYSDA,DISP=(,CATLG)
```

This job will cause z/OS to execute a program, IEFBR14. z/OS will also create and catalog a disk dataset, RALPHB.STD.OUT. Syntax information and more detail about these statements can be found in the publication *OS/390 MVS JCL Reference*, GC28-1757.

JCL is either directly or indirectly involved almost every technique that is used to invoke executable programs in z/OS. Batch is just one of several vehicles that exploit JCL. But it is the JCL and not the batch that is really important. It takes time to gain a comprehensive, proficient working knowledge of JCL, but it is not usually necessary to learn a lot of JCL to work in the z/OS environment.

A number of enhancements and components have been introduced over the years to minimize the amount of JCL you need to learn to be proficient. Some of these enhancements and components are JCL procedures, Storage Management Subsystem (SMS), and even the Interactive System Productivity Facility/Product Development Facility (ISPF/PDF). But JCL is still fundamental to the environment. Most JCL is created today through ISPF/PDF. The interface that supports ISPF/PDF is TSO/E.

1.1.5 TSO/E

TSO/E provides a z/OS interactive, command-line interface similar to the shell in the UNIX environment. Of course, this similarity cannot be taken too far. A sample TSO/E command is:

```
ALLOCATE DA('RALPHB.STD.OUT') OLD
```

This command will allocate the existing z/OS disk dataset, RALPHB.STD.OUT. However, this dataset could also have been created through TSO/E dynamic allocation by using the same ALLOCATE command. Syntax information and more detail about these commands can be found in *OS/390 TSO/E Command Reference*, SC28-1969.

We compared TSO/E to the shell and mentioned that both use a command line interface. Many people think the command line interface is difficult to work with. This was evidenced by the acceptance of Windows over the older DOS operating system. The distaste for the command line interface also gave rise to the X Window system in UNIX.

In z/OS, TSO/E has given way to ISPF/PDF, which uses panel-driven dialogs, action bars, and pull-down menus. It lets users make alphanumeric requests from various panels, and provides action bars and pull-down menus as an alternative to the panel dialog interface.

1.1.6 ISPF/PDF

This is a widely used interface that almost everyone uses with TSO/E. ISPF/PDF takes a little time to enunciate, so many people refer to it simply as ISPF.

We mentioned that ISPF/PDF is used to build JCL job streams and submit the jobs. It also supports the System Display and Search Facility (SDSF), which is used to monitor job execution and then review the jobs after termination.

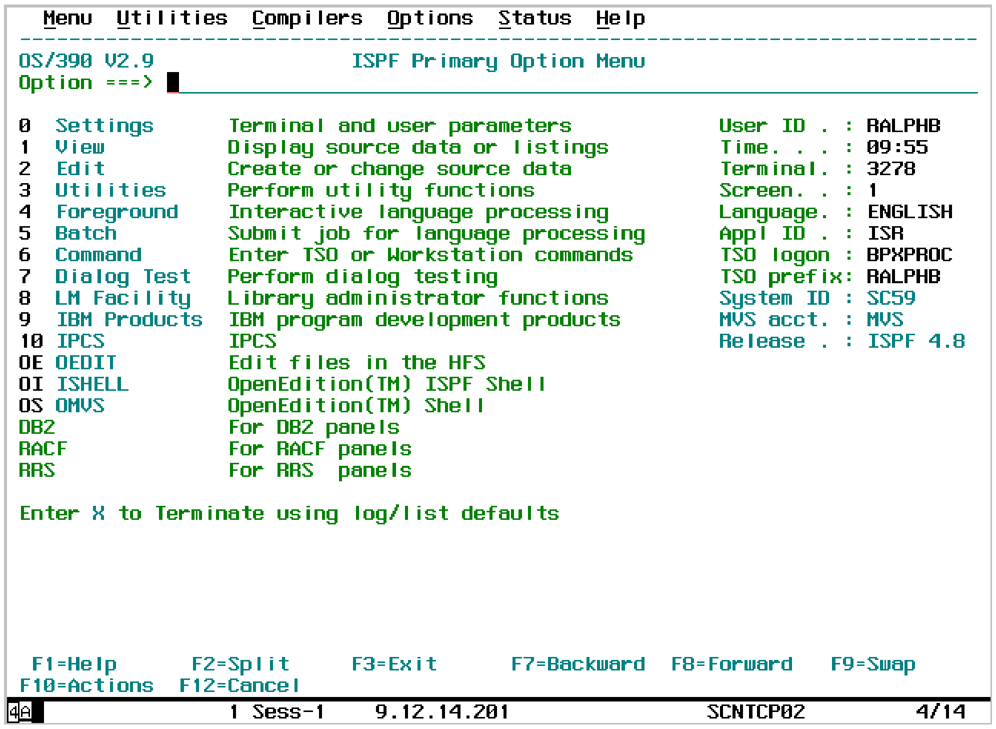
But ISPF/PDF is used for much more. It provides a user-friendly full screen editor. It allows users to invoke a number of z/OS utilities or issue TSO/E commands. It also provides an environment that facilitates the development and management of interactive applications.

A wide range of other TSO/E-based interactive applications are supported and invoked through ISPF/PDF. For example, a few of the other interactive applications that can be invoked from under this ISPF/PDF umbrella are the OMVS UNIX shell, the ISHELL, RACF panels for securing the z/OS traditional and open environments, WLM interfaces for classifying (grouping) users into service classes and prioritizing those classes, IPCS for reading dumps and traces, ISMF for systems management operations relating to disk and tape datasets, SDSF for monitoring batch jobs and issuing MVS System Commands, and SMP/E for installing products and applying maintenance to z/OS—the list goes on and on.

Interactive dialog interfaces for CICS, DB2, and IMS can even be defined and accessed from ISPF/PDF. As you can see, basic familiarity with ISPF/PDF is a key to working effectively in the z/OS environment.

ISPF/PDF provides different interfaces for the user. Users can request ISPF/PDF services by entering alphanumeric selections on ISPF/PDF panels, or users can use action bars and pull-down menus. ISPF/PDF even provides a GUI interface for workstation platforms. It offers good versatility because there are usually several ways of doing most operations under ISPF/PDF.

Many installations set up ISPF/PDF so that it is the default application that comes up after you log onto TSO/E. The ISPF/PDF Primary Option Menu is both the master menu and often the initial menu that appears. This menu is customizable, so it differs from one installation to another. However, a representative ISPF/PDF Primary Option Menu is shown in Figure 1-1.



```
  Menu  Utilities  Compilers  Options  Status  Help
 ------------------------------------------------------------------------------
 OS/390 V2.9                 ISPF Primary Option Menu
 Option ===> 

 0  Settings      Terminal and user parameters           User ID . : RALPHB
 1  View          Display source data or listings        Time. . . : 09:55
 2  Edit          Create or change source data           Terminal. : 3278
 3  Utilities     Perform utility functions              Screen. . : 1
 4  Foreground    Interactive language processing        Language. : ENGLISH
 5  Batch         Submit job for language processing     Appl ID . : ISR
 6  Command       Enter TSO or Workstation commands      TSO logon : BPXPROC
 7  Dialog Test   Perform dialog testing                 TSO prefix: RALPHB
 8  LM Facility   Library administrator functions        System ID : SC59
 9  IBM Products  IBM program development products       MVS acct. : MVS
 10 IPCS          IPCS                                   Release . : ISPF 4.8
 OE OEDIT         Edit files in the HFS
 OI ISHELL        OpenEdition(TM) ISPF Shell
 OS OMVS          OpenEdition(TM) Shell
 DB2              For DB2 panels
 RACF             For RACF panels
 RRS              For RRS  panels

 Enter X to Terminate using log/list defaults




  F1=Help       F2=Split      F3=Exit       F7=Backward   F8=Forward    F9=Swap
 F10=Actions   F12=Cancel
 4A                1 Sess-1     9.12.14.201              SCNTCP02          4/14
```

Figure 1-1 ISPF/PDF Primary Option Menu

Most users start by selecting some alphanumeric ISPF/PDF option and entering it on the Option line. For example, an OMVS shell could be generated by the following two steps:

1. On the Option line, type `6`, and press Enter.
2. On any blank line of the next menu, type OMVS, and press Enter.

This would place the user in an OMVS shell. However, TSO/E command entry is only one of the frequently used ISPF/PDF options. Some of the more heavily used ISPF/PDF options are options 0, 1, 2, 3, and 6.

A brief description of each of these five options follows:

Option 0	Specifies ISPF/PDF defaults such as function key assignments, colors, list dataset characteristics, and terminal characteristics.
Option 1	Allows a user to view or browse a text dataset or browse a binary dataset.
Option 2	Provides the capability to edit a text dataset.
Option 3	Provides interactive utilities which are used for creation, deletion, copy, move, list, print, edit, browse, compress, and related operations.
Option 6	Used to issue TSO/E commands.

As another illustration, we could have used ISPF/PDF to create the dataset RALPHB.STD.OUT (listed in “A sample job” on page 2), as follows:

1. To start, on the Option line of the Primary Option Menu, enter `3`.
2. Then, on the Option line of the resulting menu, enter `2`.

   (Alternatively you could *fastpath* by coding `=3.2` on any ISPF/PDF Option or Command line.)
3. On the Option line, type `A`. Then, on the option line that asks for it, type the dataset name (be sure to enclose the name in single quotes).

After you supply a few additional allocation parameters and press Enter, ISPF/PDF will create the dataset.

This provides only a very small sampling of the capabilities of ISPF/PDF; for further detail, refer to the *OS/390 ISPF User’s Guide*, SC28-1239. But one of the major uses of ISPF/PDF is to edit and submit a job. The job output can then be reviewed under SDSF, which we will examine next.

1.1.7 SDSF

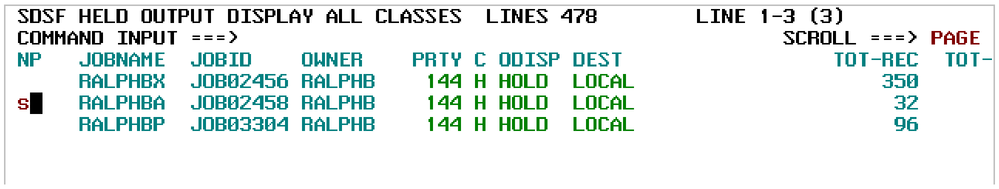
Through ISPF/PDF Edit mode, users can *edit* and *submit* batch job streams. Using SDSF, users can *monitor* their jobs during execution and then analyze the output upon job termination.

The advantage of SDSF is that it allows job output to be examined *immediately after execution*. It also allows you to interactively search the listing for user-defined keywords. At that point you can decide whether or not to print the job; you don’t need to wait for a job output listing to find out if the job executed successfully.

SDSF is usually invoked from ISPF/PDF, but you can always access it by entering SDSF under native TSO/E. Unfortunately, it is available only for JES2 environments. There is also an equivalent `sdsf` utility for shell users, but the installation must be running JES2 and must be using SDSF. This **`sdsf`** shell utility is available for the Tools and Toys Web page at:

http://www.ibm.com/servers/eserver/zseries/zos/unix/

Now suppose the job in “A sample job” on page 2 is submitted under ISPF/PDF. To examine the job output, invoke SDSF (to do this, type `SDSF` under native TSO.) Enter **`H`** on the first SDSF menu, and press Enter. You should get a screen like that shown in Figure 1-2.



```
SDSF HELD OUTPUT DISPLAY ALL CLASSES  LINES 478       LINE 1-3 (3)
COMMAND INPUT ===>                                           SCROLL ===> PAGE
NP    JOBNAME  JOBID    OWNER    PRTY C ODISP DEST                 TOT-REC  TOT-
      RALPHBX  JOB02456 RALPHB    144 H HOLD  LOCAL                    350
s     RALPHBA  JOB02458 RALPHB    144 H HOLD  LOCAL                     32
      RALPHBP  JOB03304 RALPHB    144 H HOLD  LOCAL                     96
```

Figure 1-2 SDSF Job Review menu

This SDSF screen shows that three jobs submitted by user RALPHB are still being held for review by the system. To review the job RALPHBA, position the cursor under the NP column header to the left of the job. Then type `s` as shown, and press Enter; the output from the job will be displayed.

At this point you can examine the job to determine whether it ran successfully. A good place to start would be to check the return codes generated during the execution of the job steps. To do this, go to the *Command Input* line under SDSF, type **`F 'COND C'`** and press Enter.

To check *all* return codes, press PF5 repeatedly until you've run through all the return codes. To return to ISPF/PDF, press PF3 till you reach it.

Many additional capabilities of SDSF are described in *OS/390 SDSF Guide and Reference*, SC28-1622. However, most of these additional capabilities are normally reserved for the systems staff. For example, SDSF allows you to issue MVS system commands and then see the resulting command output. It allows you to look at the progress and resource consumption of work in the system. It provides you with the capability of looking at system log information for multiple systems in a sysplex. It yields information about JES2 resources, as well as WLM.

In the following sections, we take a quick look at several operating system services in z/OS.

1.2 z/OS system services

It is sometimes difficult to distinguish between user services and system services. After all, even system services are supposedly provided for the benefit of the users. If not, what is the ultimate purpose of the service or the system itself?

However, some system services should be handled by systems staff only. For example, z/OS provides a few job- and step-related priorities to minimize the time sitting on various queues. But these priorities should be determined by the system staff; otherwise, users may maximize them for their own benefit. WLM is one component that can be used to set priorities for work in the system.

1.2.1 WLM

We look at two services provided by WLM:

- Priority establishment for z/OS system environments to assist with the balancing of workloads across multiple z/OS system images in a Parallel Sysplex.
- Address space creation for traditional and open z/OS applications. WLM supplies the address spaces for OS/390 V2R4 or higher.

Why prioritize

At the beginning of this chapter we mentioned that a typical z/OS installation can support a number of different applications. The installation might support Web browsers, e-mail, database applications, transaction managers, TSO, batch, and other applications. It is even possible that all of these applications run concurrently on a single z/OS system. So, which of these applications is the most important? Which is second?

It is probably difficult to provide a firm, absolute ranking in terms of importance. First of all, what does one mean by “important”? Precise definitions will differ from one installation to another. Even if an installation has multiple systems, the definition is unlikely to be the same for all systems. The answer may even depend on the shift, the time of the month, and a number of other factors.

But systems go down, and systems become overloaded. If a system goes down and comes back up, what should be run first? If the system becomes overloaded, what can be postponed until a later time? It is clearly necessary to plan for certain contingencies and periods when the system may be unable to handle the workload.

It is necessary to provide the systems staff with a tool for prioritizing the z/OS workload using flexible, but relative, terms. Due to the number of users and the dynamically changing workload, it is not practical to make these decisions manually. A tool needs to be available that can automate decisions relating to prioritization and the balancing of workloads.

WLM is such a tool. To use it, you first define your z/OS users and applications to WLM. It is then possible to prioritize each relative to the other applications in the system and relative to how long an individual transaction has been running.

The need for simplicity

A second problem has surfaced in z/OS environments in recent years. Human workloads have grown in parallel with system workloads. People have more work to do than ever before, and data processing is changing faster than ever before. People do not have time to learn individual system components at the same level of depth they once did. The system is rapidly increasing in complexity, and there are far more components than there were only a few years ago.

We are moving from an age of specialization to an age of generalization. The person who does the tuning probably does not have as much time to devote to it as was once available.

It used to be that much of the fine tuning was done by adjusting parameters in three members of the system dataset: SYS1.PARMLIB: IEAICSxx, IEAIPSyy, and IEAOPTzz. In fact this can still be done, but the parameters that require adjustment are complex and often difficult to relate directly to the workload; it is sometimes difficult to predict the exact consequence of a small parameter adjustment on a heavily loaded system. And above all, the person who makes these parameter changes needs to be both knowledgeable and experienced.

A simpler way to identify and prioritize user workloads was needed, and IBM introduced WLM, in part, to solve this problem. One of the advantages of the PC revolution is that it taught the general public about response time. Anyone who has browsed the Web on a slow PC understands the need for improved response time. WLM allows you to prioritize applications by specifying *desired* response time for applications.

But before we go further we should mention that WLM provides two modes for identifying work in the system:

- Compatibility Mode
- Goal Mode

WLM Compatibility Mode provides consistency with the previously used System Resources Manager (SRM) environment. It allows the systems staff to continue using a method of prioritizing work that they are probably already familiar with. It requires that you code the performance and capacity parameters in the SYS1.PARMLIB members IEAICSxx, IEAIPSyy, and IEAOPTzz. The parameters in these three members are documented in *OS/390 MVS Initialization and Tuning Reference*, SC28-1752.

WLM Goal Mode was created and introduced with the introduction of WLM. One of its purposes was to simplify the techniques used to prioritize work in the system. The problem with Compatibility Mode, as mentioned, is that it takes too much time to learn what SYS1.PARMLIB parameters to change and what the effects will be. Goal Mode allows much simpler, familiar parameters to be coded.

WLM also provides an added benefit in a Parallel Sysplex environment. A *sysplex* is a set of coupled (interconnected) z/OS systems which present a single system image. That is, they can be viewed, controlled, and managed as if they were a single system.

But systems fail or need to be taken down for maintenance or testing. Systems become overloaded. Therefore, it is sometimes necessary or advantageous to move workloads from one system to another or to balance the workloads across multiple z/OS images in a sysplex. WLM assists with the movement and balancing of these workloads.

However, workload balancing is not supported for UNIX-based workloads across multiple z/OS images in a Parallel Sysplex. Therefore, it is not considered further in this redbook.

Prioritization via WLM

If an installation elects to run in WLM Goal Mode, then several sets of definitions must be made to WLM. The main two sets of definitions are:

- Service Classes
- Classification Rules

A *service class* is a group of users, usually using closely related services, who are supposed to get the same level of service when they run. *Classification rules* are used to identify the users and insert them into service classes.

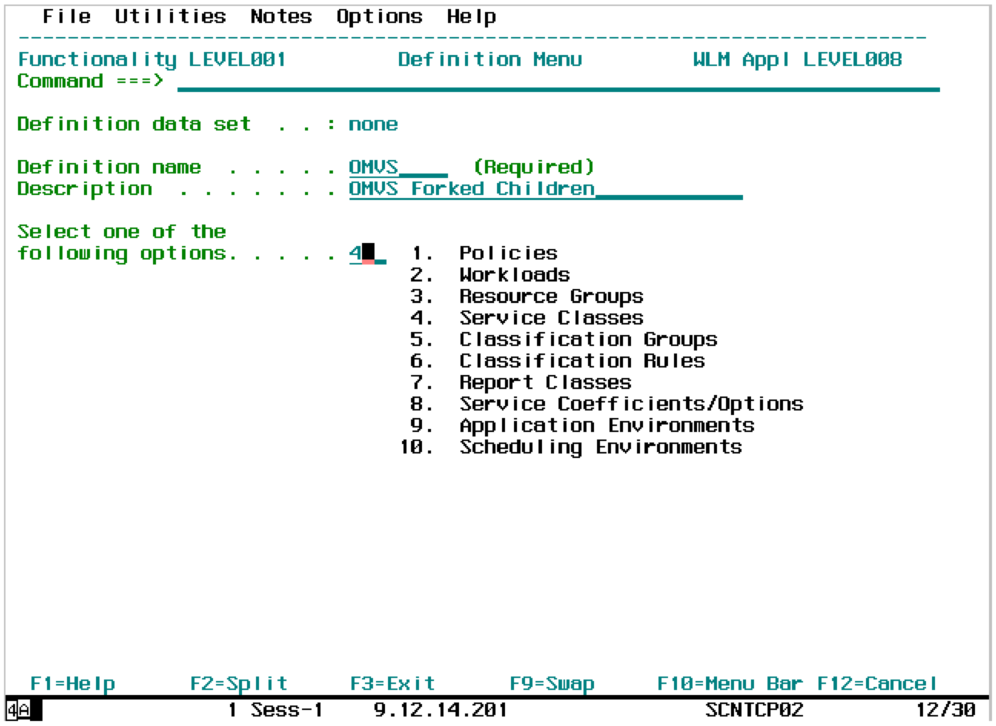
Setting definitions

To set definitions, start at a WLM Administrative Application which runs under ISPF/PDF and do the following:

1. To invoke this application, go to any ISPF/PDF option line or command line and type `TSO WLM`. (Many z/OS installations will also allow you to invoke WLM from a secondary ISPF/PDF panel.)
2. Press Enter on the first response screen, and indicate that you want to select **Create new definition**.

Now we will create a service class for OMVS forked children named OMVS.

1. To create this class, type `OMVS` where you are asked for Definition name.
2. Provide a Description where it is requested, and then specify Option **4** where it is requested. You should get the screen shown in Figure 1-3 on page 9; then press Enter.



```
  File  Utilities  Notes  Options  Help
 --------------------------------------------------------------------------
 Functionality LEVEL001          Definition Menu          WLM Appl LEVEL008
 Command ===> ______________________________________________________________

 Definition data set  . . : none

 Definition name  . . . . . OMVS____    (Required)
 Description  . . . . . . . OMVS Forked Children___________

 Select one of the
 following options. . . . . 4__  1.   Policies
                                 2.   Workloads
                                 3.   Resource Groups
                                 4.   Service Classes
                                 5.   Classification Groups
                                 6.   Classification Rules
                                 7.   Report Classes
                                 8.   Service Coefficients/Options
                                 9.   Application Environments
                                10.   Scheduling Environments

  F1=Help       F2=Split      F3=Exit       F9=Swap      F10=Menu Bar  F12=Cancel
4A                1 Sess-1   9.12.14.201                  SCNTCP02          12/30
```

Figure 1-3 Creating a WLM service class

WLM will ask you for information such as the number of periods, desired response time, and importance (level). Fill in this information, press Enter, and the first service class period will be created.

Classification rules for identifying users are defined in a similar way. Similar service class and classification rule definitions must be created for all the z/OS users. Then you will be ready to run in WLM Goal Mode. Other WLM sample screen definitions for z/OS UNIX System Services can be found in *OS/390 UNIX System Services Planning*, SC28-1890. A general reference for WLM is *OS/390 MVS Planning: Workload Management*, GC28-1761.

The second function of WLM in the z/OS environment is to supply address spaces when they are needed. The UNIX component is a heavy user of z/OS address spaces.

WLM Supplied Address Spaces

Prior to OS/390 V2R4, the APPC Transaction Scheduler, ASCH, was responsible for supplying address spaces for z/OS UNIX System Services. Starting in OS/390 V2R4, either ASCH or WLM can be used to supply these address spaces. However, WLM is more efficient, and it is the default method for supplying these address spaces.

WLM uses a JCL procedure, BPXAS, in a proclib dataset, SYS1.PROCLIB, to create these address spaces. It uses the same procedure regardless of whether the system is running in Compatibility or Goal Mode. Fortunately, no setup or tuning is required for WLM to use BPXAS.

Of course, WLM and various other system components collect information about the jobs and users while they are in execution. Much of this information is made available through the System Management Facilities (SMF). We will next look at SMF.

1.2.2 SMF

SMF is responsible for maintaining information about the resources used by the users in the system. It collects information about job names, programs in execution, how long they were in execution, how much storage they used, how many input/output operations they did, how many lines they printed, whether they were running in WLM Compatibility Mode or Goal Mode, information for RACF, and information about the use of z/OS UNIX System Services resources. We could continue this list for many more pages.

This information is recorded in well over 100 types (and subtypes) of SMF records. Each SMF record type and subtype contains a number of fields for collecting information. Various z/OS components write these records to SMF datasets. Almost all z/OS installations collect and save this data for further analysis.

The data is most frequently used for accounting purposes. If you bill users for their use of data processing resources, the source of this billing information almost always comes from the analysis of SMF records. However, this SMF data is also used for several other purposes. It can be used to generate RMF performance and capacity planning reports. Auditing reports are sometimes generated from this data. It can also be used to record information about Lotus Domino, CICS, DB2, and RACF. Needless to say, it has many uses.

Several SMF record types collect data that is important to z/OS UNIX System Services:

- SMF Type 30 contains information about jobs and job steps. The OpenMVS Process Section has information specific to z/OS UNIX System Services.
- SMF Type 34 contains information about address space creation for UNIX.
- SMF Type 35 contains information about UNIX process termination.
- SMF Type 42, Subtype 6 contains information about HFS dataset access.
- SMF Type 72 (several subtypes) contains information about workload activity. Some of this information comes from WLM.
- SMF Type 74, Subtype 3 contains information about UNIX kernel activity.
- SMF Type 80 contains RACF auditing information about the use of UNIX resources.

The standard reference for SMF is *OS/390 MVS System Management Facilities,* GC28-1783.

SMF record types in the range 70 through 79 are used to generate RMF reports. As was noted above, SMF record types 72 and 74 are of particular value in a z/OS UNIX System Services environment. We should look at RMF.

1.2.3 RMF

Like SMF, RMF collects information about the use of z/OS resources. The main concern of SMF is to collect data for accounting and billing purposes. On the other hand, RMF data is also used to identify performance bottlenecks, provide data for system tuning, and predict how long each system resource is likely to last.

Needless to say, if you do not tune the system, it will result in more frequent hardware upgrades. If you do not monitor system capacity, you may not have sufficient lead time to order and install new hardware or software. Either way, it costs you in real dollars. RMF aids you in tuning and capacity planning.

RMF collects data about the use of z/OS resources and then presents this data in a complete and comprehensive set of RMF reports. This RMF information is collected by three different RMF monitors:

- RMF Monitor I
- RMF Monitor II
- RMF Monitor III

The reports generated by these monitors provide *extremely* valuable, useful, and comprehensive information about the system. We now look at each of these three monitors in more detail.

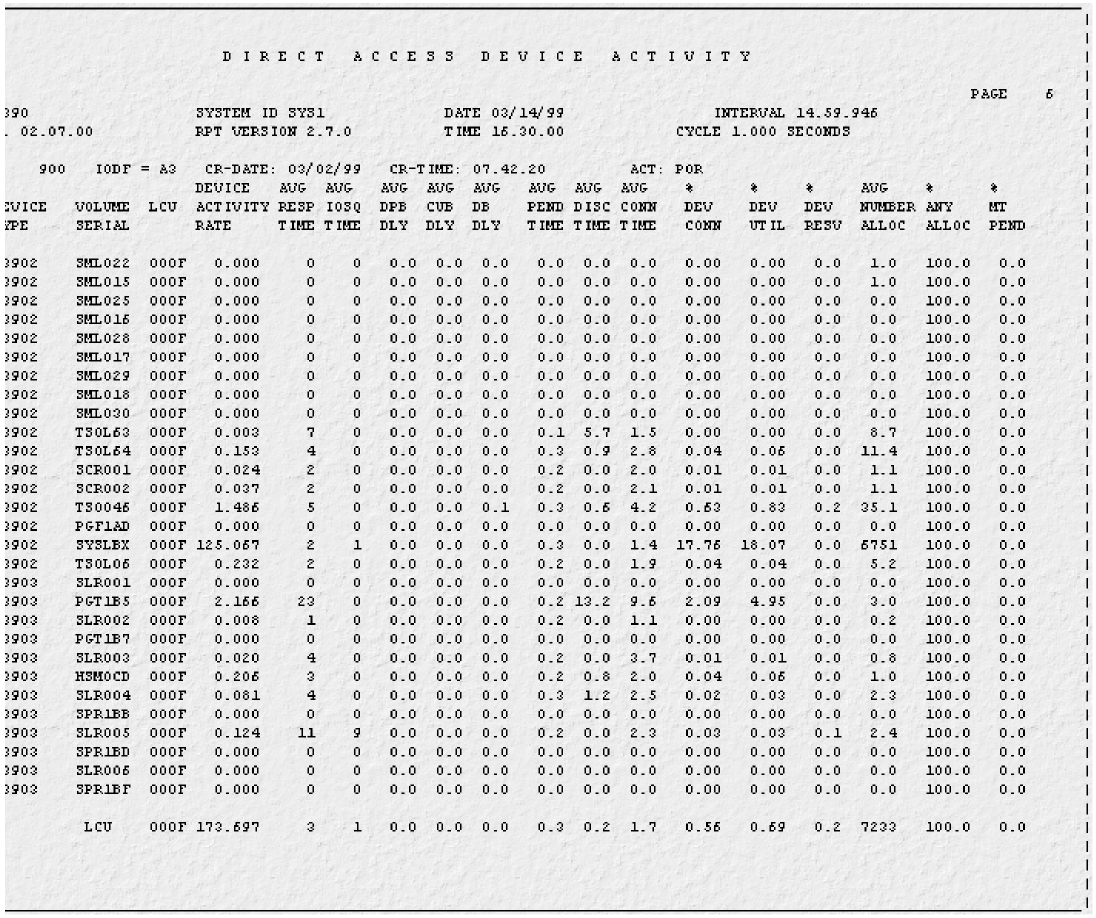
RMF Monitor I

An RMF session is usually started from a z/OS console as a started task (a JCL procedure). By default, Monitor I is started at the same time. If an installation is using RMF, the Monitor 1 session normally runs continuously over long periods of time and samples system resources during that time. The frequency of sampling and a number of other RMF Monitor I options are determined by the installation.

Information about which system resources the Monitor I session is supposed to sample are normally specified in the SYS1.PARMLIB member ERBRMF00. Monitor I information can be collected about the use of the CPU, DASD (disk), tape, channels, paging, swapping, workload activity, and several other system resources. This information is stored in SMF records which are written to the SMF datasets. The SMF records that contain the RMF data are later extracted, sorted by data and time, and then fed through a program called the (RMF) Post Processor.

The DASD Activity Report

The Post Processor generates numerous Monitor I reports. Typically these are printed reports. The numbers in the Monitor I reports are usually averages based on sampling, but exact counts sometimes appear. An example of a Monitor I report that shows performance information for DASD (disk) is shown in Figure 1-4 on page 12.



```
                    D I R E C T   A C C E S S   D E V I C E   A C T I V I T Y

                                                                                                PAGE    6
390                SYSTEM ID SYS1          DATE 03/14/99             INTERVAL 14.59.946
. 02.07.00         RPT VERSION 2.7.0       TIME 16.30.00         CYCLE 1.000 SECONDS

   900   IODF = A3   CR-DATE: 03/02/99   CR-TIME: 07.42.20        ACT: POR
                   DEVICE   AVG  AVG   AVG  AVG  AVG   AVG  AVG  AVG    %      %     %    AVG    %      %
VICE    VOLUME  LCU  ACTIVITY RESP IOSQ  DPB  CUB  DB    PEND DISC CONN   DEV    DEV   DEV  NUMBER ANY    MT
PE      SERIAL       RATE     TIME TIME  DLY  DLY  DLY   TIME TIME TIME   CONN   UTIL  RESV ALLOC  ALLOC  PEND

3902    SML022  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   1.0  100.0  0.0
3902    SML015  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   1.0  100.0  0.0
3902    SML025  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    SML016  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    SML028  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    SML017  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    SML029  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    SML018  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    SML030  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    TSOL63  000F   0.003     7    0  0.0  0.0  0.0   0.1  5.7  1.5   0.00   0.00  0.0   8.7  100.0  0.0
3902    TSOL64  000F   0.153     4    0  0.0  0.0  0.0   0.3  0.9  2.8   0.04   0.06  0.0  11.4  100.0  0.0
3902    SCR001  000F   0.024     2    0  0.0  0.0  0.0   0.2  0.0  2.0   0.01   0.01  0.0   1.1  100.0  0.0
3902    SCR002  000F   0.037     2    0  0.0  0.0  0.0   0.2  0.0  2.1   0.01   0.01  0.0   1.1  100.0  0.0
3902    TSO046  000F   1.486     5    0  0.0  0.0  0.1   0.3  0.6  4.2   0.63   0.83  0.2  35.1  100.0  0.0
3902    PGF1AD  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3902    SYSLBX  000F 125.067     2    1  0.0  0.0  0.0   0.3  0.0  1.4  17.76  18.07  0.0  6751  100.0  0.0
3902    TSOL06  000F   0.232     2    0  0.0  0.0  0.0   0.2  0.0  1.9   0.04   0.04  0.0   5.2  100.0  0.0
3903    SLR001  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3903    PGT1B5  000F   2.166    23    0  0.0  0.0  0.0   0.2 13.2  9.6   2.09   4.95  0.0   3.0  100.0  0.0
3903    SLR002  000F   0.008     1    0  0.0  0.0  0.0   0.2  0.0  1.1   0.00   0.00  0.0   0.2  100.0  0.0
3903    PGT1B7  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3903    SLR003  000F   0.020     4    0  0.0  0.0  0.0   0.2  0.0  3.7   0.01   0.01  0.0   0.8  100.0  0.0
3903    HSM0CD  000F   0.206     3    0  0.0  0.0  0.0   0.2  0.8  2.0   0.04   0.06  0.0   1.0  100.0  0.0
3903    SLR004  000F   0.081     4    0  0.0  0.0  0.0   0.3  1.2  2.5   0.02   0.03  0.0   2.3  100.0  0.0
3903    SPR1BB  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3903    SLR005  000F   0.124    11    9  0.0  0.0  0.0   0.2  0.0  2.3   0.03   0.03  0.1   2.4  100.0  0.0
3903    SPR1BD  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3903    SLR006  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0
3903    SPR1BF  000F   0.000     0    0  0.0  0.0  0.0   0.0  0.0  0.0   0.00   0.00  0.0   0.0  100.0  0.0

         LCU    000F 173.697     3    1  0.0  0.0  0.0   0.3  0.2  1.7   0.56   0.69  0.2  7233  100.0  0.0
```

Figure 1-4 RMF Monitor I DASD (Disk) Activity Report

This DASD Activity Report displays the device number (location) of each disk device, its volume serial number, the device activity rate (I/O [input/output] requests per second), the average response time (time required to complete an I/O request in milliseconds), the device utilization (the percent of time the device is busy), and lots more. Many additional types of Monitor I reports can be requested.

The Kernel Activity Report

A number of the z/OS UNIX System Services parameters are set in a SYS1.PARMLIB member, BPXPRMxx. Some of these parameters are not intuitive, and it is difficult to determine what their values should be. Another RMF Monitor I report, the Kernel Activity Report, is particularly useful in monitoring these parameter values and changing them if they need to be changed. This report helps to determine when these parameter values need to be changed and what the parameter value should be.

Now let’s move to Monitor II reports.

RMF Monitor II

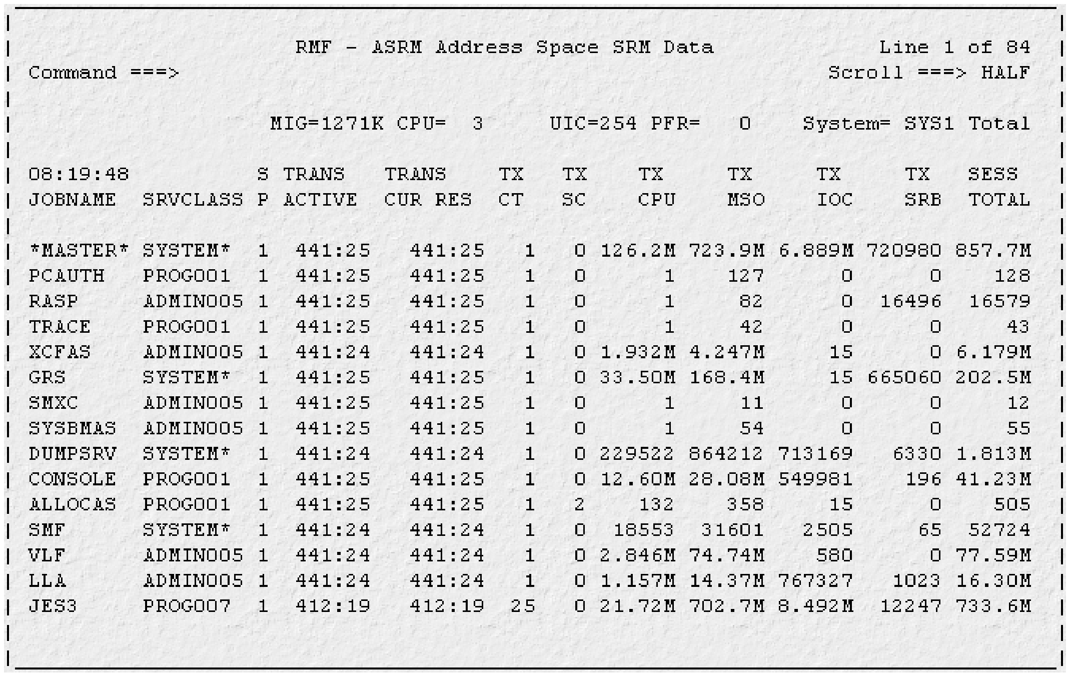
The Monitor II session provides a real time snapshot of z/OS system activity, or the activity of an individual address or job. It is even possible to continuously monitor a specific job. Up to 32 Monitor II sessions can run concurrently. Monitor II sessions can also write SMF Type 79 records if the session is running in the background.

The Monitor II session is started by using the modify (F) system command to modify the existing RMF started task to start a Monitor II session. The SYS1.PARMLIB member ERBRMF01 specifies the type of information that Monitor II is to collect. Monitor II information can be collected about central and expanded storage requirements, page and swap activity, channel activity, I/O activity, HFS dataset activity, enqueue activity, and several more areas.

Monitor II reports

Monitor II reports are either written back to the terminal screen in real time mode, or equivalent printed versions of the reports can be generated. To examine a Monitor II report, do the following:

1. Go to native TSO/E, and issue the command **`RMFMON`**. This will generate the RMF Display Menu (you can also issue **`RMF`** under native TSO and indicate that you want to examine a Monitor II report).
2. Specify the type of report you want. For example. if you issue the command **`ASRM`** in the command entry area at the upper left of the screen, you should get a report similar to the ASRM RMF Monitor II report shown in Figure 1-5.



```
                       RMF - ASRM Address Space SRM Data              Line 1 of 84
 Command ===>                                                        Scroll ===> HALF

                     MIG=1271K CPU=  3     UIC=254 PFR=   0     System= SYS1 Total

 08:19:48            S  TRANS    TRANS     TX    TX     TX      TX       TX      TX    SESS
 JOBNAME   SRVCLASS  P  ACTIVE   CUR RES   CT    SC     CPU     MSO      IOC     SRB   TOTAL

 *MASTER*  SYSTEM*   1  441:25   441:25     1     0 126.2M  723.9M  6.889M  720980  857.7M
 PCAUTH    PROG001   1  441:25   441:25     1     0      1     127       0       0     128
 RASP      ADMIN005  1  441:25   441:25     1     0      1      82       0   16496   16579
 TRACE     PROG001   1  441:25   441:25     1     0      1      42       0       0      43
 XCFAS     ADMIN005  1  441:24   441:24     1     0 1.932M  4.247M      15       0  6.179M
 GRS       SYSTEM*   1  441:25   441:25     1     0 33.50M  168.4M      15  665060  202.5M
 SMXC      ADMIN005  1  441:25   441:25     1     0      1      11       0       0      12
 SYSBMAS   ADMIN005  1  441:25   441:25     1     0      1      54       0       0      55
 DUMPSRV   SYSTEM*   1  441:24   441:24     1     0 229522  864212  713169    6330  1.813M
 CONSOLE   PROG001   1  441:25   441:25     1     0 12.60M  28.08M  549981     196  41.23M
 ALLOCAS   PROG001   1  441:25   441:25     1     2    132     358      15       0     505
 SMF       SYSTEM*   1  441:24   441:24     1     0  18553   31601    2505      65   52724
 VLF       ADMIN005  1  441:24   441:24     1     0 2.846M  74.74M     580       0  77.59M
 LLA       ADMIN005  1  441:24   441:24     1     0 1.157M  14.37M  767327    1023  16.30M
 JES3      PROG007   1  412:19   412:19    25     0 21.72M  702.7M  8.492M   12247  733.6M
```

Figure 1-5 RMF Monitor II ASRM Report

The ASRM Report is a row report. Each row represents a different address space (table reports, which repeatedly examine the same information at different times, are also available). This ASRM report identifies the jobname, the WLM Goal Mode service class for the job, the count of transactions, the amount of TCB service units, storage services units, I/O service units, SRB service units, and a few other bits of information.

Another RMF Monitor II report, the Hierarchical File System Statistics Report, provides useful information about HFS datasets in the z/OS UNIX System Services environment. A separate report can be requested for each HFS dataset. It tells you the date and time when the dataset was mounted, its size, and the number of types of I/O requests to the dataset.

We will now look at the Monitor III reports.

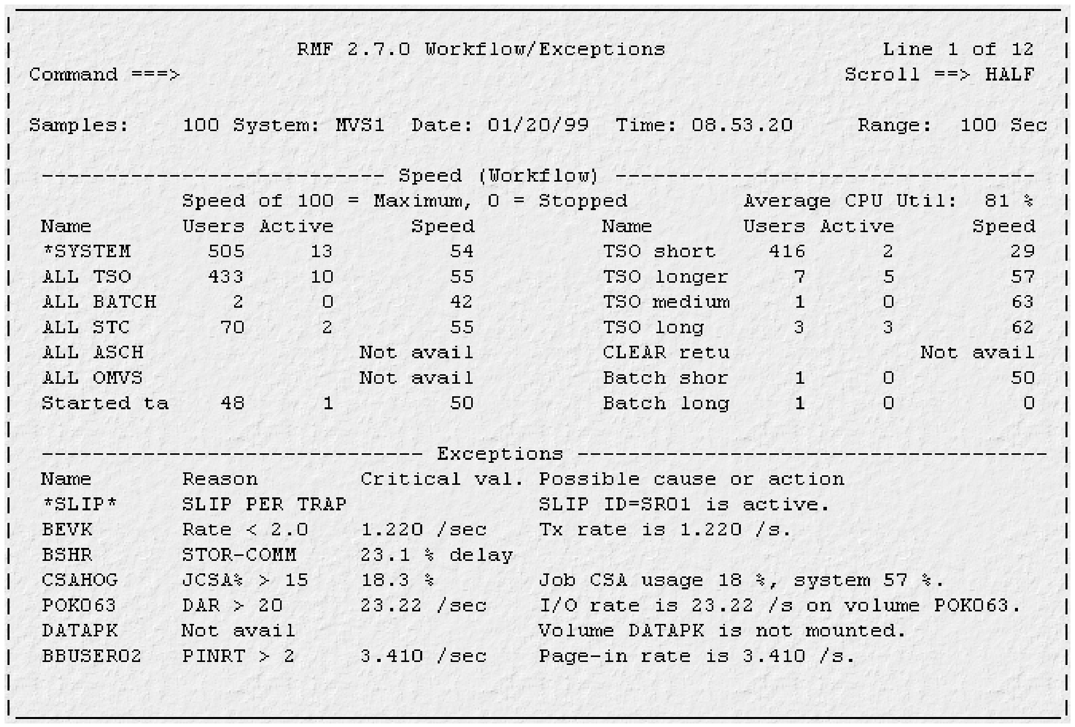
RMF Monitor III

The Monitor III session is an interactive session which collects its data by sampling. It typically takes a sample once per second, and then summarizes every 100 seconds. It provides a wide variety of reports about an entire Parallel Sysplex, or an individual system. It provides information indicating whether you are meeting your WLM-specified goals in a sysplex. It gives you a response time distribution for the work you are running. It identifies potential bottlenecks in a Coupling Facility or structures in a Coupling Facility. It tells you which jobs and users are running slowly and the likely reason for their slow progress through the system. The main focus of Monitor III sessions is to identify where you have delays and bottlenecks.

To start a Monitor III session, go to a z/OS console. Issue the modify (F) command, and indicate that a Monitor III session is to be started. The SYS1.PARMLIB member ERBRMF04 specifies the kind of data you want to gather for analysis. You can also gather the data by creating SMF records or by writing records to Virtual Storage Access Method (VSAM) datasets on disk.

But suppose you want to look at a specific Monitor III report. To request Monitor III reports, go to native TSO/E, and issue the command RMFWDM.

This will generate the RMF Monitor III Primary Menu (you can also enter RMF under native TSO and select a Monitor III report). Now you can specify the type of report you want. For example, if you issue the command WFEX on the Selection line, you should get a report similar to the WFEX RMF Monitor III report shown in Figure 1-6.



```
                      RMF 2.7.0 Workflow/Exceptions                Line 1 of 12
 Command ===>                                                      Scroll ==> HALF

 Samples:    100 System: MVS1  Date: 01/20/99  Time: 08.53.20     Range:  100 Sec

  --------------------------- Speed (Workflow) --------------------------------
             Speed of 100 = Maximum, 0 = Stopped         Average CPU Util:  81 %
  Name       Users Active       Speed         Name       Users Active      Speed
  *SYSTEM      505     13          54         TSO short    416      2         29
  ALL TSO      433     10          55         TSO longer     7      5         57
  ALL BATCH      2      0          42         TSO medium     1      0         63
  ALL STC       70      2          55         TSO long       3      3         62
  ALL ASCH                  Not avail         CLEAR retu               Not avail
  ALL OMVS                  Not avail         Batch shor     1      0         50
  Started ta    48      1          50         Batch long     1      0          0

  ------------------------------ Exceptions -------------------------------------
  Name       Reason         Critical val. Possible cause or action
  *SLIP*     SLIP PER TRAP                SLIP ID=SR01 is active.
  BEVK       Rate < 2.0     1.220 /sec    Tx rate is 1.220 /s.
  BSHR       STOR-COMM      23.1 % delay
  CSAHOG     JCSA% > 15     18.3 %        Job CSA usage 18 %, system 57 %.
  POK063     DAR > 20       23.22 /sec    I/O rate is 23.22 /s on volume POK063.
  DATAPK     Not avail                    Volume DATAPK is not mounted.
  BBUSER02   PINRT > 2      3.410 /sec    Page-in rate is 3.410 /s.
```

Figure 1-6 RMF Monitor III Workflow Exceptions Report

This report is divided into two main sections. The top half provides information about the various categories of work (TSO, BATCH, Started Tasks, OMVS, ...) and shows how rapidly they are moving through the system in terms of speed (a speed of 100 means they are moving at the fastest possible rate, while 0 means they are moving at the slowest possible rate).

The bottom half identifies the worst-performing jobs and users in the system. For example, we can see that job DATAPK is not doing well. Apparently it is due to the fact that the volume, DATAPK, is not mounted.

There are two especially useful RMF references: *OS/390 Resource Measurement Facility (RMF) User's Guide*, SC28-1949 documents how to start the monitors, what data to collect, and how to generate reports, while *OS/390 Resource Measurement Facility (RMF) Report Analysis*, SC28-1950 provides various sample reports and interprets the report information.

We mentioned the RMF Monitor II report, the Hierarchical File System Statistics Report earlier. But we did not discuss HFS datasets because we had not discussed Storage Management Subsystem (SMS). It is time to remedy that situation.

1.2.4 SMS

SMS is a disk and tape management subsystem for z/OS installations. Therefore, it is identified as such in member IEFSSNxx of the system dataset, SYS1.PARMLIB. It is part of the required DFSMS/MVS data management component of z/OS.

Why SMS

SMS was introduced into the large systems environment in the mid-1980s for several reasons. One reason was to simplify JCL for the end user. At that time more emphasis was being placed on user friendliness in data processing, and JCL was a barrier to entering the z/OS world.

Data management

Another reason for the introduction of SMS dealt with the complexity and time required to manage disk and tape datasets in the environment. Even in the PC world, data management is becoming an issue.

For example, before the appearance of Windows, there were disk compression software packages which "magically" found more space on the disk. Everyone always wants the latest software; however, the latest software requires more disk space, and more disk space requires bigger drives. It is sometimes difficult to remember whether a long-forgotten file needs to be kept. These data management issues are not unique to the PC world.

The large systems world has been dealing with these problems for years. Due to the larger numbers of datasets and disk and tape drives in the large systems world, this is an even bigger problem for z/OS installations. IBM has been addressing these data management problems over the years by introducing a number of products to simplify and automate the management of disk and tape datasets.

The Data Facility Storage Management Subsystem/MVS (DFSMS/MVS) incorporates a number of different components which are used to manage disk and tape storage. It includes SMS, which simplifies JCL and automatically controls dataset placement during dataset creation.

DFSMS/MVS also includes a backup utility, DFSMSdss, which can take full volume or dataset backups and then restore the volumes or datasets. In addition, it can do standalone restores if the operating system is not up.

A DFSMShsm component automates backups and migrates infrequently-used datasets to less expensive media. A DFSMSrmm component uses a tape robot to automate the retrieval, mount, unmount, and filing of tape cartridges back into the stacks. There are a few other products, such as DFSORT, which belong to this Systems Managed Storage family. All of these products taken together provide valuable assistance in automating the management of disk and tape datasets in the z/OS environment.

SMS configuration issues

Before SMS can be used in an installation, an SMS configuration must be created by the storage administrator. For example, the storage administrator must decide which datasets and volumes are to be managed by SMS.

Most datasets are eligible to be placed under the control of SMS. In fact, *most* application and user datasets probably *should be* controlled by SMS, while some datasets, such as PDSEs and VSAM RLS datasets, *must* be SMS-managed. Initially, the z/OS UNIX System Services HFS dataset had to be SMS-managed, but in an OS/390 V2R7 environment with DFSMS/MVS V1R5, this requirement was relaxed.

Note: Some system datasets, such as SYS1.NUCLEUS, cannot be managed by SMS.

Keep in mind that SMS-managed datasets reside on SMS-managed volumes, and non-SMS-managed datasets reside on non-SMS-managed volumes.

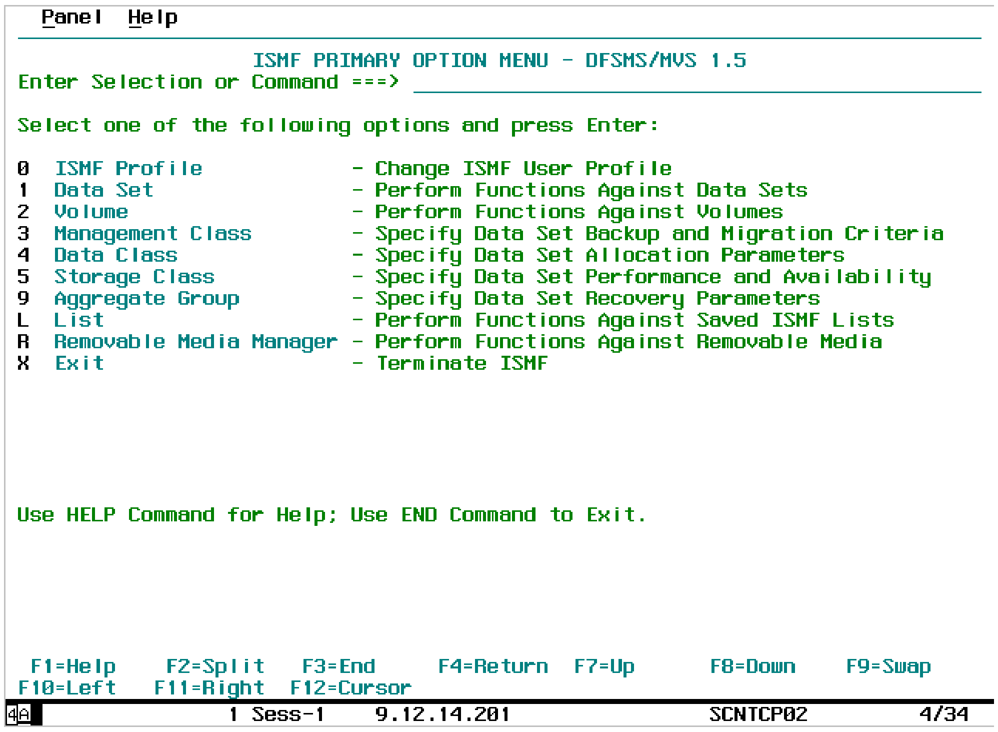
Data set management parameter defaults must be specified. These are coded in SMS constructs. The storage administrator must supply the Automatic Class Selection (ACS) routines which are used by SMS to assign constructs during dataset creation. These ACS routines are not supplied by IBM, since installations differ so widely in their dataset management requirements. Instead, IBM provides a script-like language which is used by the storage administrator to code these ACS routines.

Five types of constructs can be defined:

- The Data Class Construct is used to specify parameters which are used in creating new datasets. This construct simplifies and reduces the number of DD statement parameters which must be coded. It can be assigned during the creation of a dataset regardless of whether the dataset is to be SMS-managed or not.
- The Storage Class Construct allows you to indicate the desired level of performance and the availability you want when accessing an SMS-managed dataset. This information determines which SMS-managed volume the dataset is to be placed on.
- The Management Class Construct specifies dataset backup, migration, and retention information. It allows you to indicate how frequently a dataset should be backed up or how long it should be retained by the system before it is deleted.
- The Storage Group Construct allows you to manage a group of disk or tape volumes as a single volume. When you create an SMS-managed dataset, a dataset will be placed on a volume from the assigned storage group. Volume serial numbers can still be coded on DD statements during dataset creation. However, if the dataset is to be SMS-managed, the volume will be ignored, and SMS will select a volume.
- The Aggregate Group Construct allows you to define groups of datasets which can be backed up or recovered in a single operation.

Additional information on SMS constructs can be found in *DFSMS/MVS DFSMSdfp Storage Administration Reference*, SC26-4920.

The starting point for defining these constructs is a panel-driven application, Interactive Storage Management Facility (ISMF). Many installations will define it so that it is accessible from ISPF/PDF. In that case, select it from the ISPF/PDF menu, and you will get the ISMF Primary Option Menu shown in Figure 1-7.



```
  Panel  Help
-------------------------------------------------------------------------------
                   ISMF PRIMARY OPTION MENU - DFSMS/MVS 1.5
Enter Selection or Command ===> _____________________________________________

Select one of the following options and press Enter:

0  ISMF Profile               - Change ISMF User Profile
1  Data Set                   - Perform Functions Against Data Sets
2  Volume                     - Perform Functions Against Volumes
3  Management Class           - Specify Data Set Backup and Migration Criteria
4  Data Class                 - Specify Data Set Allocation Parameters
5  Storage Class              - Specify Data Set Performance and Availability
9  Aggregate Group            - Specify Data Set Recovery Parameters
L  List                       - Perform Functions Against Saved ISMF Lists
R  Removable Media Manager - Perform Functions Against Removable Media
X  Exit                       - Terminate ISMF





Use HELP Command for Help; Use END Command to Exit.






 F1=Help     F2=Split    F3=End      F4=Return   F7=Up       F8=Down     F9=Swap
F10=Left    F11=Right   F12=Cursor
4A                1 Sess-1     9.12.14.201                SCNTCP02           4/34
```

Figure 1-7   ISMF Primary Option Menu

Additional information on the ISMF application can be found in *DFSMS/MVS Using the ISMF*, SC26-4911.

HFS datasets

Directories and files in the z/OS UNIX System Services file system must be placed in HFS datasets. We mentioned that HFS datasets were initially required to be SMS-managed, and initially they were single volume datasets. With the introduction of OS/390 V2R7 and DFSMS/MVS V1R5, however, both requirements have been eliminated.

Thus the HFS datasets can be SMS- or non-SMS-managed. They can now be multivolume datasets. But the multivolume support still requires that they be SMS-managed. Therefore, if they are non-SMS-managed, they must be single volume datasets. In addition, you lose the capability of coding the JCL PATH parameters on DD statements if they are non-SMS managed.

2

UNIX and C/C++ basics in z/OS

In recent years, UNIX has become a big factor in computing. The rise in the popularity of the Internet has been a major driving force behind UNIX. IBM introduced UNIX System Services to OS/390 in MVS/ESA SP4.3 in March 1994.

In September 1996, OS/390 was branded XPG4.2 (UNIX 95)-compliant—the first operating system from a non-UNIX heritage to be UNIX-branded. In OS/390 Version 2, Release 7, IBM began to add UNIX 98 functions to OS/390.

Thus many C/C++ applications which formerly had to run on UNIX workstations can now run under z/OS and OS/390 along with transaction manager, database, batch, and other related applications. This redbook addresses the issue of porting UNIX based C/C++ applications to z/OS.

The word *port* traditionally refers to the process of moving (and usually modifying) the C/C++ source code to run on another platform. Before we discuss C/C++ or porting, we should first talk about the interactive interfaces to z/OS UNIX System Services.

2.1 The shell, ISHELL and OMVS interfaces

z/OS supports the standard shell (/bin/sh) and an enhanced C shell in OS/390 V2R9 or later (/bin/tcsh). The standard shell is based on the Korn shell and is upward-compatible with the Bourne shell. Additionally, there are other shells such as the KornShell (ksh93) and Bourne Again Shell (bash) that can be added to z/OS UNIX. See 3.8, “Shells and command languages” on page 48 for an additional discussion.

The z/OS UNIX Shell is a POSIX.2-compliant shell and was the first shell supported in OS/390. The tcsh shell is an enhanced but completely compatible version of the Berkeley C shell, csh. In addition, TSO provides an interactive interface, the ISPF/PDF Shell (the ISHELL), which allows access to UNIX System Services. The POSIX and tcsh shells are widely used in UNIX environments. It should be noted that the ISHELL is really a TSO interface and not a true UNIX shell.

Similar capabilities are available under the z/OS UNIX Shell, the tcsh shell, and to a lesser extent, the ISHELL. The interface you use is a matter of personal preference. If you have a background in UNIX, DOS, or OS/2, you will probably feel more comfortable with the command line interface provided with the z/OS UNIX Shell or tcsh shell. On the other hand, if you are familiar with ISPF/PDF, you might prefer the panel dialog interface provided with the ISHELL.

The exact set of capabilities and commands available to you depends on which interface you choose and how you log in. At this point we will discuss methods of creating a shell to access z/OS UNIX System Services.

2.1.1 Accessing z/OS UNIX shells

Any of the following five methods can be used to access either the z/OS UNIX Shell or tcsh shell. The RACF user profile and UNIX file customization will determine which shell you get and its characteristics.

For example, if the PROGRAM parameter of the OMVS segment of the RACF user profile specifies PROGRAM(‘/bin/sh’), you get a z/OS UNIX Shell during login. However, if the PROGRAM parameter specifies PROGRAM(‘/bin/tcsh’), then you get a tcsh shell during login.

In this section we assume that the OMVS segment contains PROGRAM(‘/bin/sh’). Therefore, the user will be accessing a z/OS UNIX Shell. Figure 2-1 on page 21 summarizes these five methods of accessing a shell. As you read left to right across the figure, the five methods use:

1. The OMVS command from a terminal
2. The OMVS command from a workstation
3. The `rlogin` command from a workstation
4. The `telnet` command from a workstation
5. Shell access from an Outboard Communication Server

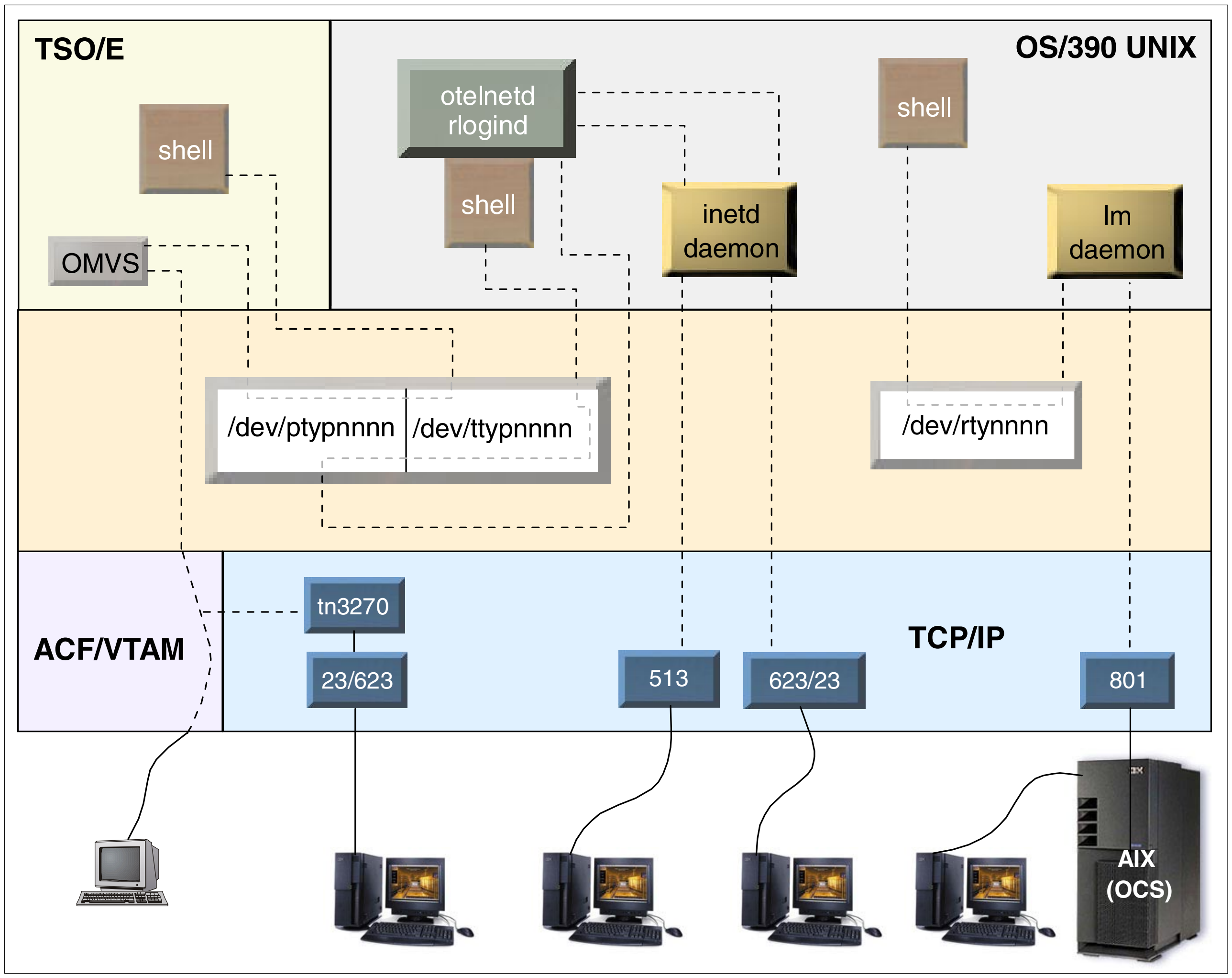


Figure 2-1 Five ways of accessing a z/OS UNIX Shell

Let's look at each of these methods in more detail.

The OMVS command from a terminal

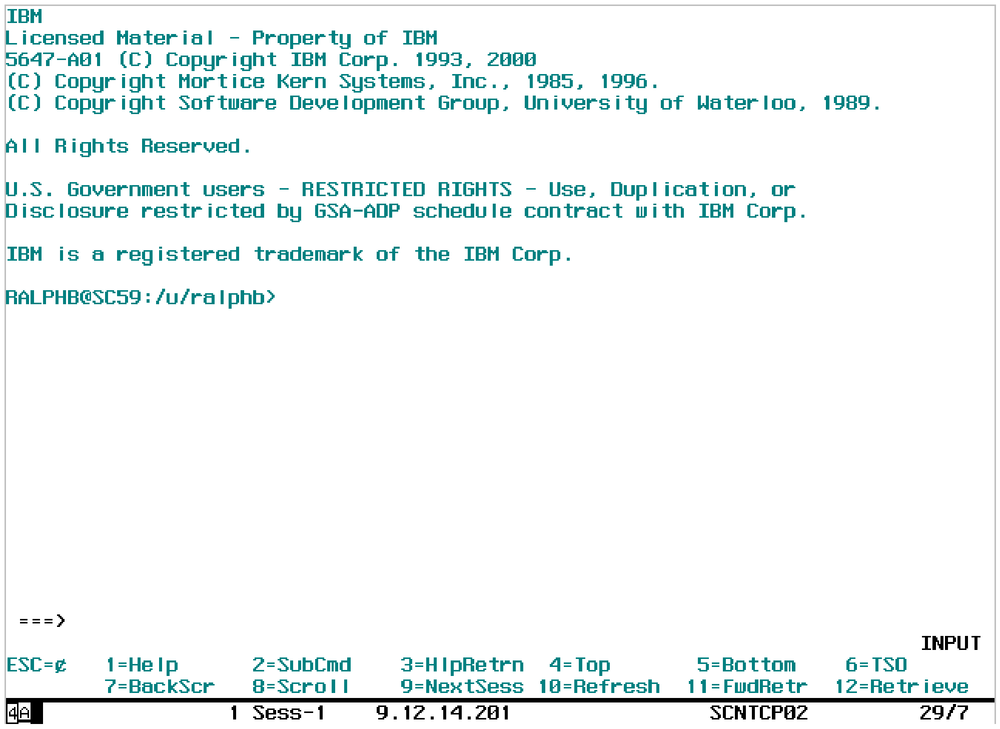
You can access z/OS UNIX System Services from a non-programmable (dumb) terminal that is SNA-attached to the zSeries or S/390 host. Log on to TSO, and then issue the OMVS TSO/E command. This will put you in the shell, where it is possible to invoke commands and utilities.

In this case you can issue the `exit` shell command to leave the shell and return to TSO. Thus you have full TSO access if you used the OMVS command to build a shell. In particular, you can also access the ISHELL and ISPF/PDF editor. (However, you cannot access the UNIX vi editor.)

You can use the SESSIONS parameter on the OMVS command to establish up to four shell sessions. If you ask for more than one session, z/OS ensures that all are created in the same TSO address space. You are always in block or line mode if you use the TSO OMVS command.

The OMVS command from a workstation

You can access z/OS UNIX System Services from a programmable workstation that is TCP/IP-attached to the host. In this case, if you want to issue the OMVS command, you must use telnet **`tn3270`** emulation or an equivalent 3270 emulator to access TSO. Log on to TSO, type the OMVS command after the TSO READY prompt, and you should get the OMVS shell shown in Figure 2-2.



```
IBM
Licensed Material - Property of IBM
5647-A01 (C) Copyright IBM Corp. 1993, 2000
(C) Copyright Mortice Kern Systems, Inc., 1985, 1996.
(C) Copyright Software Development Group, University of Waterloo, 1989.

All Rights Reserved.

U.S. Government users - RESTRICTED RIGHTS - Use, Duplication, or
Disclosure restricted by GSA-ADP schedule contract with IBM Corp.

IBM is a registered trademark of the IBM Corp.

RALPHB@SC59:/u/ralphb>









 ===>
                                                                   INPUT
ESC=¢    1=Help      2=SubCmd    3=HlpRetrn  4=Top       5=Bottom    6=TSO
         7=BackScr   8=Scroll    9=NextSess 10=Refresh  11=FwdRetr  12=Retrieve
4A                 1 Sess-1    9.12.14.201                SCNTCP02         29/7
```

Figure 2-2 The OMVS shell

The characteristics of this shell are the same as the shell you might build by issuing the OMVS command from a terminal. Therefore, you have full TSO access and can create an ISHELL. The ISPF/PDF editor is available, but the vi editor is not. The SESSIONS parameter allows up to four shell sessions to be built in your TSO address space, and you are always in block or line mode.

The rlogin command from a workstation

Go to a workstation that is TCP/IP-attached to the z/OS host, and use the **`rlogin`** command to perform a remote login. You will need to specify a hostname or IP address during the login. Typical **`rlogin`** syntax looks like this:

```
$ rlogin wtsc59oe -l ralphb
```

where `wtsc59oe` is the hostname and `ralphb` is the user ID. You should then be prompted for a password. A pair of host-based daemons, **`inetd`** and **`rlogind`**, will cooperate to build a shell.

This shell has a slightly different look and feel than either of the shells you created through the OMVS command. You may have seen PF keys displayed at the bottom on the shell screens built via the OMVS command, but you cannot customize this **`rlogin`** shell screen to display PF keys. You will also have to enter the shell command or utility immediately after the shell prompt, instead of on the command line. To delete a character, use the backspace key.

This shell does not allow you to exit back to TSO. Therefore, you cannot get to the ISHELL. The tso shell command allows limited access to TSO/E commands, but some environmental variable customization and TSO logon proc customization is required. The **tsocmd** command provides access to a larger subset of TSO/E commands. For example, **tsocmd** allows you to issue RACF commands, whereas **tso** does not. The **tsocmd** command is available from the Tools and Toys page at the following Web site:

```
http://www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1toy.html
```

The **rlogin** command does not provide access to the ISPF/PDF editor. However, the vi editor *is* available. Thus here is another difference between the rlogin- and OMVS-created shells. A few shell commands that are available in the OMVS shell are not available in the rlogin shell, and vice versa.

For example, the commands oedit and obrowse are available under the OMVS shell, but not the rlogin shell. We previously mentioned that the vi editor is available under the rlogin shell, but not the OMVS shell. However, such commands are the exception rather than the rule; the vast majority of shell commands are available under any of the shells discussed here.

You can establish multiple shell sessions by issuing rlogin repeatedly to create multiple shells. In this case, each shell will be created in a different address space. The **rlogin** command puts you in block mode by default, but certain shell commands like **vi** operate in character or raw mode. Therefore, this shell can cause greater overhead for z/OS than the OMVS shell.

The telnet command from a workstation

You can also use the **telnet** command from a workstation that is TCP/IP-attached to the z/OS host. You will need to specify a hostname or TCP/IP address to indicate the host you are accessing. Typical **telnet** syntax looks like this:

```
telnet wtsc59oe 623
```

where `wtsc59oe` is the hostname and `623` is the port. The port may be the default telnet port of 23, in which case the command can be simplified to:

```
telnet wtsc59oe
```

There is another possibility: you could build a Windows icon to invoke a telnet client. The Tera Term Pro or Hummingbird telnet clients are good candidates to be configured in this way. The IBM Redbook *Networked Applications on OS/390 UNIX*, SG24-5447 has a valuable discussion of these clients, as well as the xterm client, in Chapter 3.

Regardless of how you invoke telnet, you will be prompted for a user ID and password.

Two host-based daemons, **inetd** and **otelnetd**, will create a shell. These telnet commands communicate with a different telnet daemon than the telnet **tn3270** daemon previously mentioned. In this case, no 3270 emulation is involved.

Both **rlogin** and **telnet** provide a similar interface. Both produce a slightly greater overhead for z/OS than the OMVS shell for at least two reasons.

The first reason deals with the implementation of multiple shell session support. As an example, if you build three shells via telnet or rlogin, it will require three address spaces. However, three OMVS shells would require only one address space. In general, the larger the number of address spaces, the greater the overhead.

The second reason for increased overhead deals with character mode support in the shell. We mentioned that commands like vi use character mode. This means that if your rlogin or telnet workstation is directly attached to the z/OS host, then a character is transmitted to the host every time you press a key on the keyboard (though telnet can operate in line mode). Each character transmission produces an interrupt on the host.

To avoid the resulting performance degradation on the host, IBM provides a fifth method of logging in to z/OS. This involves using an intermediate AIX platform, which must be configured in an Outboard Communication Server (OCS) configuration.

Shell access from an Outboard Communication Server

This method is provided to minimize the overhead associated with character mode support. The OCS platform requires that rlogin workstations, telnet workstations, or ASCII terminals be directly attached to a RS/6000 running AIX 4.1.3 or above. These workstations or terminals are attached to the RS/6000 by means of a serial port.

The RS/6000 can either be attached to the zSeries or S/390 host via a LAN or channel-to-channel attachment. An OCS configuration is required on both AIX and z/OS. Terminal descriptions, IP addresses, etc. must be specified in the OCS configurations on *both* operating systems. Support for this OCS configuration is included as part of a base element in z/OS.

Users log into the z/OS host as if they were directly attached. The shell they see appears to be the same as the shell they saw when they were directly attached. However, when they are in a character mode session using a utility like vi or talk, the character transmissions will be *trapped* by the AIX platform. It will use these individual characters to create blocks, and then send blocks to z/OS. z/OS treats these users as if they were in block mode.

Table 2-1 summarizes the characteristics of the shells created by the five methods.

Table 2-1 Characteristics of the five methods of building a z/OS UNIX Shell

	OMVS command at terminal	**OMVS command at workstation**	**rlogin access from workstation**	**telnet access from workstation**	**Terminal or workstation on OCS host**
How attached	ACF/VTAM	TCP/IP	TCP/IP	TCP/IP	TCP/IP
TCP/IP port	None	23/623	513	623/23	801
TSO access	Full	Full	Limited	Limited	Limited
ISHELL access	Yes	Yes	No	No	No
File editors	ISPF/PDF	ISPF/PDF	vi, ed	vi, ed	vi, ed
# sessions	Multiple	Multiple	Multiple	Multiple	Multiple
Shell mode	Block	Block	Block/Char	Block/Char	Block/Char
PF key use	Yes	Yes	No	No	No

2.1.2 Accessing the tcsh shell

Any of the five methods discussed in 2.1, “The shell, ISHELL and OMVS interfaces” on page 20 can be used to access a tcsh shell as well as an z/OS Shell. If the OMVS segment contains the parameter PROGRAM(‘/bin/tcsh’), you will get a tcsh shell when you log in. Therefore Table 2-1 also summarizes the five ways to create a tcsh shell.

On the other hand, if the OMVS segment contains PROGRAM('/bin/sh'), then you can still create a tcsh shell by entering the command **`tcsh`** on the command line of the z/OS Shell.

2.1.3 Accessing an alternate shell

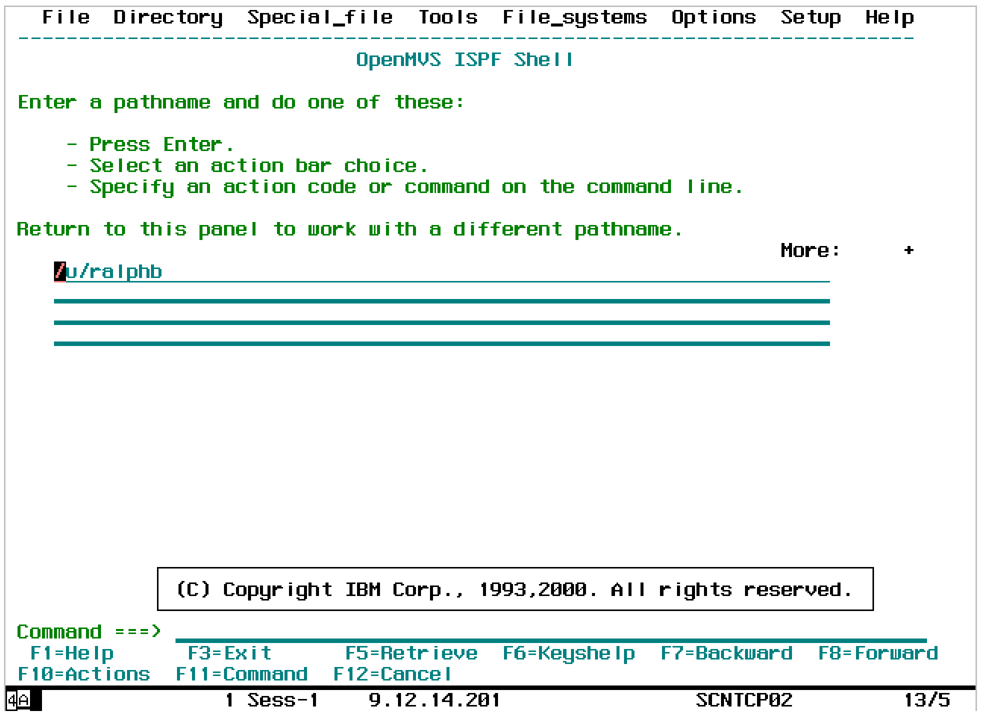
An installation can also install and use an alternate shell. The shell can be accessed directly from the HFS. But, for optimal performance, perform the following steps:

- Place the alternate shell in the Link Pack Area.
- Turn on the sticky bit in the file that contains this shell.
- Specify the path name of this shell in the PROGRAM parameter of the OMVS segment of the RACF user profile.
- Specify the path name of this shell via the sh option in the /etc/init.options file.
- Further customize the /etc/init.options file to ensure that the initialization parameters are consistent with this new shell.

2.1.4 The ISHELL

We want to set the record straight at the outset. The ISHELL is not a shell; it is a TSO/E command that provides shell services from an ISPF/PDF panel dialog interface similar to that provided for other ISPF/PDF applications. Some people (perhaps those familiar with TSO and ISPF) claim that UNIX is not user-friendly.

In part to address this issue, IBM introduced the ISHELL or ISPF/PDF Shell. The ISHELL uses an IPSF/PDF interface which has become the de facto standard for most users who are accessing TSO. It provides a user-friendly (or at least somewhat familiar) interface for these users. It can be invoked by going to TSO or ISPF/PDF option 6 and typing **`ISHELL`**. (You can also enter **`ISH`**, which is the abbreviation for ISHELL.) This yields the OpenMVS ISPF Shell primary menu shown in Example 2-3.



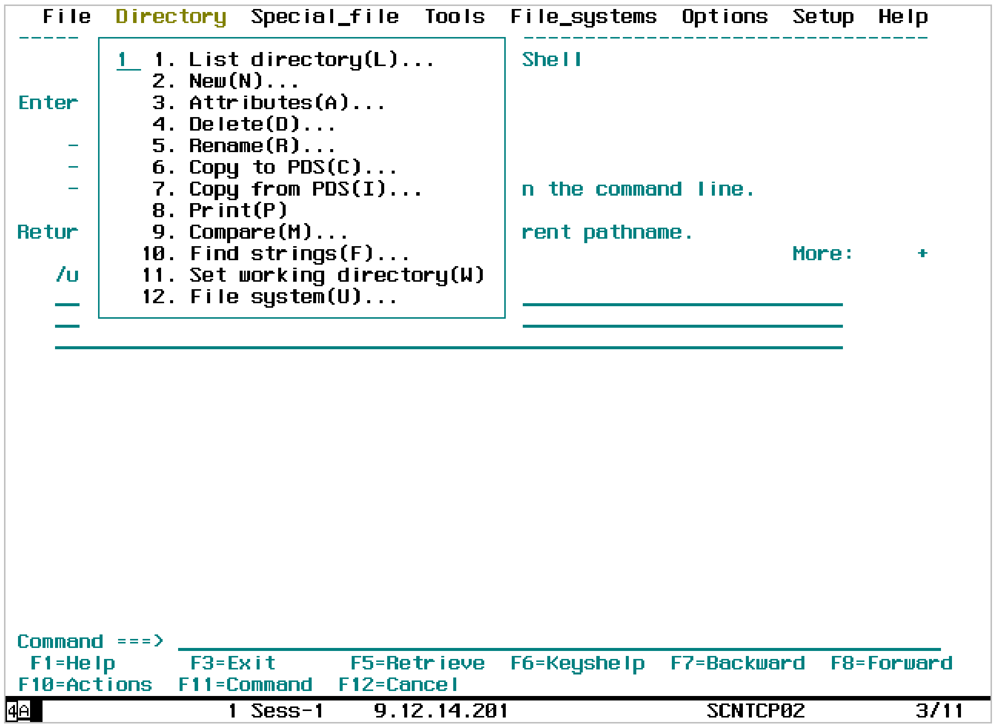
```
   File  Directory  Special_file  Tools  File_systems  Options  Setup  Help
 --------------------------------------------------------------------------
                              OpenMVS ISPF Shell

 Enter a pathname and do one of these:

     - Press Enter.
     - Select an action bar choice.
     - Specify an action code or command on the command line.

 Return to this panel to work with a different pathname.
                                                               More:     +
    /u/ralphb




             (C) Copyright IBM Corp., 1993,2000. All rights reserved.

 Command ===>
  F1=Help      F3=Exit      F5=Retrieve  F6=Keyshelp  F7=Backward  F8=Forward
 F10=Actions  F11=Command  F12=Cancel
 4A                  1 Sess-1     9.12.14.201                 SCNTCP02        13/5
```

Figure 2-3 The OpenMVS ISPF Shell primary menu

The ISHELL provides most of the services that are available to anyone using a shell like the z/OS or tcsh shell. It provides the standard UNIX directory and file services. It provides shell command execution services for most z/OS UNIX Shell commands.

You start with the OpenMVS ISPF Shell primary menu shown in Figure 2-3. For example, if you want to list the entries in directory /u/ralphb, then position the cursor under the Directory Option on the action bar of the OpenMVS ISPF Shell primary menu and press Enter.

Then type **1** on the blank line in the upper left corner of the pull-down menu, as shown in Figure 2-4 on page 26. If you press Enter again, you'll see a display of the entries in /u/ralphb.



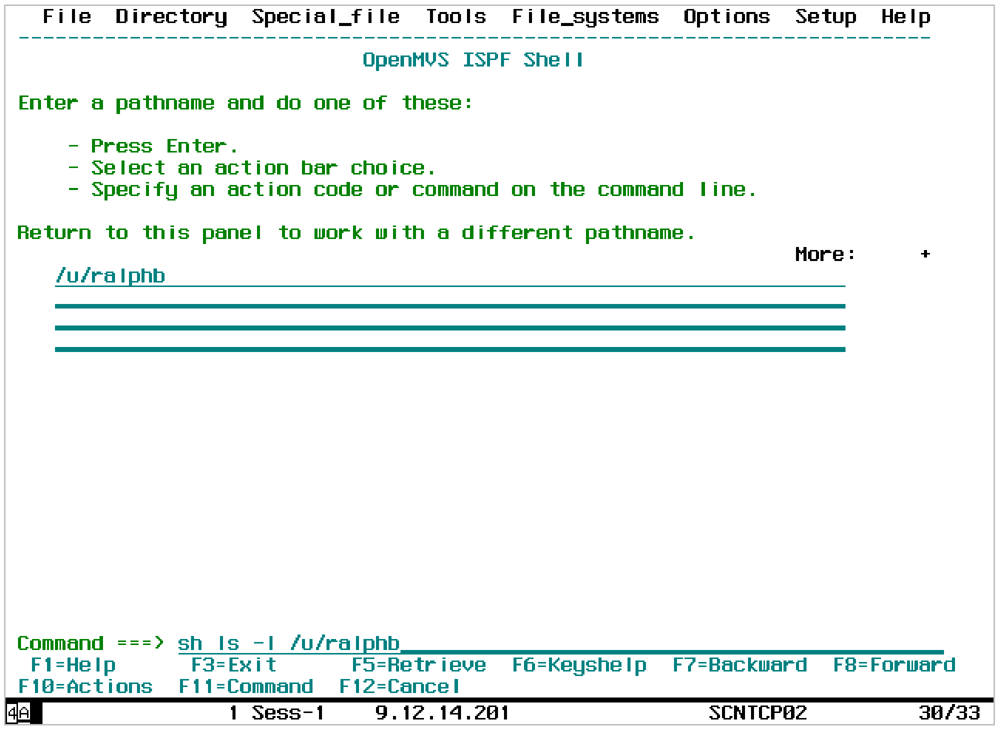
```
  File  Directory  Special_file  Tools  File_systems  Options  Setup  Help
 -----  ┌────────────────────────────────┐ ---------------------------------
        │ 1_  1. List directory(L)...    │ Shell
        │     2. New(N)...               │
 Enter  │     3. Attributes(A)...        │
        │     4. Delete(D)...            │
     -  │     5. Rename(R)...            │
     -  │     6. Copy to PDS(C)...       │
     -  │     7. Copy from PDS(I)...     │ n the command line.
        │     8. Print(P)                │
 Retur  │     9. Compare(M)...           │ rent pathname.
        │    10. Find strings(F)...      │                     More:     +
    /u  │    11. Set working directory(W)│
    __  │    12. File system(U)...       │ __________________________
    __  └────────────────────────────────┘ __________________________
    ______________________________________________________________


 Command ===> ______________________________________________________________
  F1=Help      F3=Exit      F5=Retrieve  F6=Keyshelp  F7=Backward  F8=Forward
 F10=Actions  F11=Command  F12=Cancel
 4A                1 Sess-1    9.12.14.201              SCNTCP02         3/11
```

Figure 2-4 Using the ISHELL Directory pull-down menu

File services could be requested in a similar way. In this case, you would position the cursor under the File Option on the action bar of the ISHELL Primary Option Menu and press Enter. Then select the appropriate option from this pull-down menu.

The ISHELL also allows you to issue most shell commands using the usual shell command syntax. Just select the **`Tools`** Option, and select option **2** on the pull-down menu.

Equivalently, you can prefix the shell command by sh, and then enter the resulting prefixed command on the ISHELL command line. For example, if you type in **`sh ls -l /u/ralphb`** on the command line of the screen in Figure 2-5 on page 27 and press Enter, you'll get a list of directory and file entries in the directory /u/ralphb.



```
  File  Directory  Special_file  Tools  File_systems  Options  Setup  Help
 --------------------------------------------------------------------------
                             OpenMVS ISPF Shell

 Enter a pathname and do one of these:

     - Press Enter.
     - Select an action bar choice.
     - Specify an action code or command on the command line.

 Return to this panel to work with a different pathname.
                                                              More:     +
    /u/ralphb







 Command ===> sh ls -l /u/ralphb
  F1=Help       F3=Exit       F5=Retrieve   F6=Keyshelp   F7=Backward   F8=Forward
 F10=Actions   F11=Command   F12=Cancel
                  1 Sess-1     9.12.14.201                SCNTCP02          30/33
```

Figure 2-5 Entering shell commands from the ISHELL

The ISHELL provides several services which are not normally available to shell users in most UNIX environments. For example, it allows you to create HFS datasets and mount and unmount HFS file systems.

2.2 UNIX standards

The focus of this redbook is on porting C/C++ applications from other UNIX platforms to z/OS. The C language is important on UNIX due to its historical background. C++ has become popular in recent years because of its heritage from C and its object-oriented capabilities. Both of these languages are now standardized under the International Standards Organization (ISO).

Due to the market force on operating systems, significant effort has also been put in by major software vendors to standardize UNIX. The hallmark of the early effort was POSIX created by IEEE (POSIX stands for Portable Operating System Interface - uniX).

This was followed by the X/Open Portability Guide (XPG) from the Open Group; the Open Group was a consortium of software vendors that came together to push for an open standard in UNIX. These efforts merged in 1995 and a single standard was created known as *the Single Unix Specification*, or UNIX95 for short. This was subsequently updated and enhanced in 1998 to UNIX98.

IBM has been an active member in all these efforts, and has been investing in product developments to push for conformance and openness. z/OS conforms to UNIX95; effort is underway to update the conformance to UNIX98.

2.3 An overview of compile and bind operations

Except for some minor differences in terminology, the compile and bind (or link) operations are the same in z/OS as for other UNIX environments. However, a salient z/OS optimization feature, the Interprocedural Analyzer (IPA), performs aggressive cross-file global optimizations.

There are several ways to compile C/C++ source and bind executable programs (executables) under z/OS:

- The **c89** utility invokes the Standard C (or ANSI C) compiler and/or binder. It is most commonly invoked under the shell.
- The **cc** utility invokes the Common Usage C compiler and/or binder. It is also usually invoked under the shell.
- The **c++** utility invokes the C++ compiler and/or binder. It is normally invoked via the shell interface.
- z/OS also provides a number of JCL procedures (procs) which can be used for compile and/or bind operations.
- Several TSO REXX EXECs are available for compile and bind operations in z/OS.
- ISPF/PDF provides interactive panels which can be used to compile and bind in z/OS.

This list describes several ways of invoking the C compiler, but in this redbook we only describe the first three methods in detail. (The method involving the use of JCL procs is more commonly used in the traditional z/OS environment, and the last two methods of invoking the compiler are not relevant to our subject.)

The C++ compiler is designed to compile C++ code; and since C++ is a superset of C, it can be used to compile C code as well. In fact, if you can compile a C program with the C++ compiler, and if the C program contains a large number of small functions, the executable may execute more efficiently. (The C++ compiler provides superior facilities for handling of function calls, but C++ references global data less efficiently than C.)

Compile and bind operations using the shell command **c89** and z/OS JCL procs look different, but produce equivalent results. We can illustrate this with a sample file, types.c:

```
$ cat types.c
#include <stdio.h>
 void main()
  {
   printf("sizeof(void *) = %d\n", sizeof(void *));
   printf("sizeof(char) = %d\n", sizeof(char));
   printf("sizeof(short) = %d\n", sizeof(short));
   printf("sizeof(int) = %d\n", sizeof(int));
   printf("sizeof(long) = %d\n", sizeof(long));
   printf("sizeof(float) = %d\n", sizeof(float));
   printf("sizeof(double) = %d\n", sizeof(double));
   printf("sizeof(long double) = %d\n", sizeof(long double));
  }
```

We also assume the member TYPES, in the PDSE, RALPHB.SOURCE.C, contains source code identical to what is shown above. We will now look at how you might compile and bind using **c89** and z/OS procs.

2.3.1 The c89 utility

We first consider the **c89** utility.

Compile examples

The compiler reads C/C++ source code and compiles it. As the following two examples illustrate, **c89** can read this source either from a file or a dataset.

- Issue the following command to compile input in the file types.c:

  ```
  $ c89 -c types.c
  ```

- Issue the following command to compile input from the PDS RALPHB.SOURCE.C(TYPES):

  ```
  $ c89 -c "//'RALPHB.SOURCE.C(TYPES)'"
  ```

Compile and bind examples

The compiler compiles the C/C++ source, and the binder produces an executable version of the program from this C/C++ source. The C/C++ source can be read from a file or dataset, and the executable program can be written to a file or dataset. Therefore, **c89** allows you to combine input and output in four possible ways. Figure 2-6 illustrates these combinations.

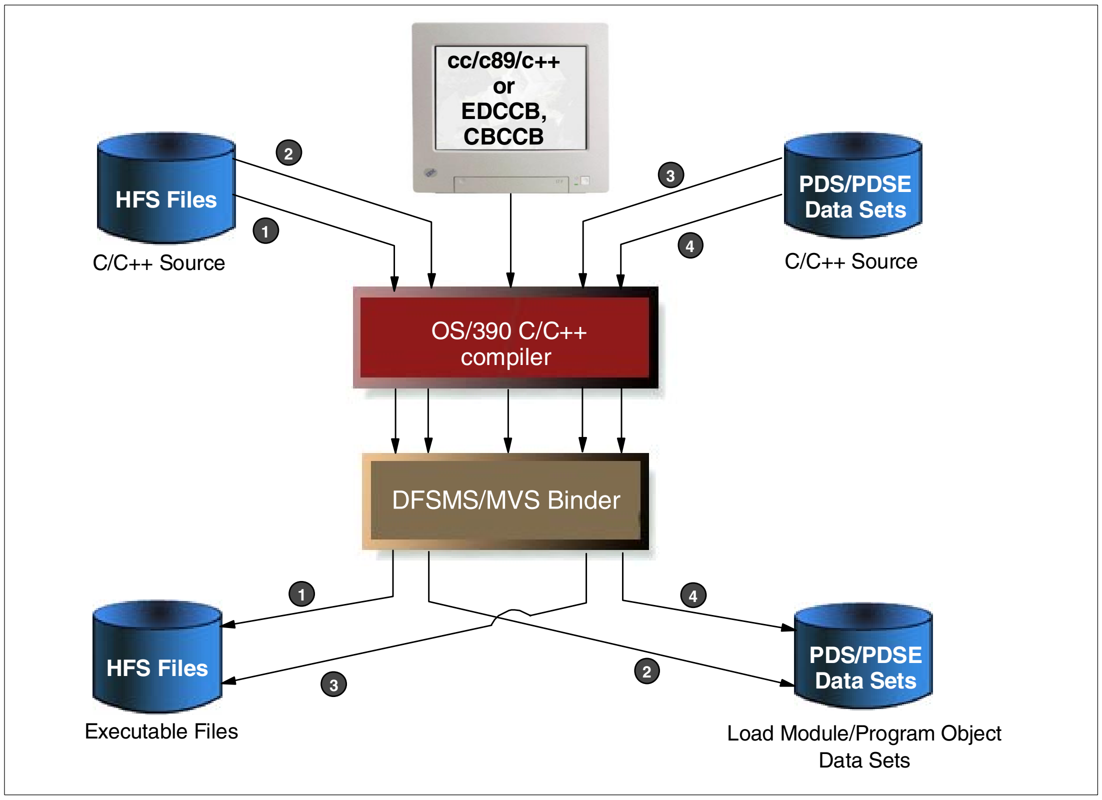


Figure 2-6 Compile and bind via utilities(c89, cc, or c++) or procedures (EDCCB or CBCCB)

Here are the four ways we could invoke the **c89** utility:

1. Issue the following command to compile input from the file, types.c, and write output to the executable types:

   ```
   $ c89 -o types types.c
   ```

2. Issue the following command to compile input from the file, types.c, and write the member TYPES to the dataset RALPHB.LIB.OUT:

   ```
   $ c89 -o "//'RALPHB.LIB.OUT(TYPES)'" types.c
   ```

3. Issue the following command to compile input from member TYPES of the dataset RALPHB.SOURCE.C, and write output to the file, types:

```
$ c89 -o types "//'RALPHB.SOURCE.C(TYPES)'"
```

4. Issue the following command to compile input from the member TYPES of dataset RALPH.SOURCE.C, and write output to the member TYPES of the dataset RALPHB.LIB.OUT:

```
$ c89 -o "//'RALPHB.LIB.OUT(TYPES)'" "//'RALPHB.SOURCE.C(TYPES)'"
```

In all of the preceding examples, the options and operands can be interchanged if you set the environmental variable `_C89_CCMODE=1`. Thus, instead of writing `c89 -o types types.c`, we could have written `c89 types.c -o types`.

2.3.2 JCL procedures for compile and bind

z/OS has traditionally provided JCL procedures for compile and bind operations for the programming languages it supports. The JCL procedures for a C compile and C compile and bind are EDCC and EDCCB, respectively. The JCL procedures for a C++ compile and C++ compile and bind are CBCC and CBCCB, respectively.

In the following section, we go through examples similar to those preceding; however, this time we'll use JCL procedures.

Compile examples

We go through two compile jobs using the EDCC procedure (proc).

- Submit the following job to compile input from the file, types.c, and write output to the file, types.o:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//STEP1    EXEC EDCC
//COMPILE.SYSIN  DD PATH='/u/ralphb/types.c'
//COMPILE.SYSLIN DD PATH='/u/ralphb/types.o',
//       PATHMODE=(SIRWXU),PATHOPTS=(ORDWR,OCREAT,OTRUNC)
```

- Submit the following job to compile input from the member TYPES of the dataset RALPHB.SOURCE.C, and write output to the file, types.o:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//STEP1    EXEC EDCC
//COMPILE.SYSIN  DD DSN=RALPHB.SOURCE.C(TYPES),DISP=SHR
//COMPILE.SYSLIN DD PATH='/u/ralphb/types.o',
//       PATHMODE=(SIRWXU),PATHOPTS=(ORDWR,OCREAT,OTRUNC)
```

Compile and bind examples

Procs provide the same versatility as the **c89** utility. Procs will read the C/C++ source code from a file or dataset and write the executable version of the program to a file or dataset. Therefore, as Figure 2-6 on page 29 illustrates, procs allow you to combine input and output in four possible ways.

Here are four ways to invoke the EDCCB procedure.

1. Submit the following job to compile input from the file, types.c, and write an executable to an existing file, types:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//STEP1    EXEC EDCCB
//COMPILE.SYSIN  DD PATH='/u/ralphb/types.c'
//BIND.SYSLMOD DD PATH='/u/ralphb/types',PATHMODE=(SIRWXU),
//       PATHOPTS=(ORDWR,OCREAT,OTRUNC)
```

2. Submit the following job to compile input from the file, types.c, and write output to member TYPES of the existing dataset RALPHB.LIB.OUT:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//STEP1    EXEC EDCCB
//COMPILE.SYSIN  DD PATH='/u/ralphb/types.c'
//BIND.SYSLMOD DD DSN=RALPHB.LIB.OUT(TYPES),DISP=OLD
```

3. Submit the following job to compile input from member TYPES of the dataset RALPHB.SOURCE.C, and write output to an existing file, types:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//STEP1    EXEC EDCCB
//COMPILE.SYSIN  DD DSN=RALPHB.SOURCE.C(TYPES),DISP=SHR
//BIND.SYSLMOD DD PATH='/u/ralphb/types',PATHMODE=(SIRWXU),
//       PATHOPTS=(ORDWR,OCREAT,OTRUNC)
```

4. Submit the following job to compile input from member TYPES in the dataset RALPHB.SOURCE.C and write it to member TYPES in the existing dataset RALPHB.LIB.OUT:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//STEP1    EXEC EDCCB
//COMPILE.SYSIN  DD DSN=RALPHB.SOURCE.C(TYPES),DISP=SHR
//BIND.SYSLMOD DD DSN=RALPHB.LIB.OUT(TYPES),DISP=OLD
```

2.4 Invoking an executable

In the previous section we provided a sample source file, types.c, and then provided a number of examples showing how an executable might be created from that file.

In this section, we discuss methods that can be used to invoke that executable.

The examples demonstrated that a compile and bind operation can store an executable in a file or dataset. Regardless of where the executable is stored, it can be invoked from the shell, TSO, batch, or another C/C++ executable program. Figure 2-7 on page 32 illustrates these four methods of invoking an executable.

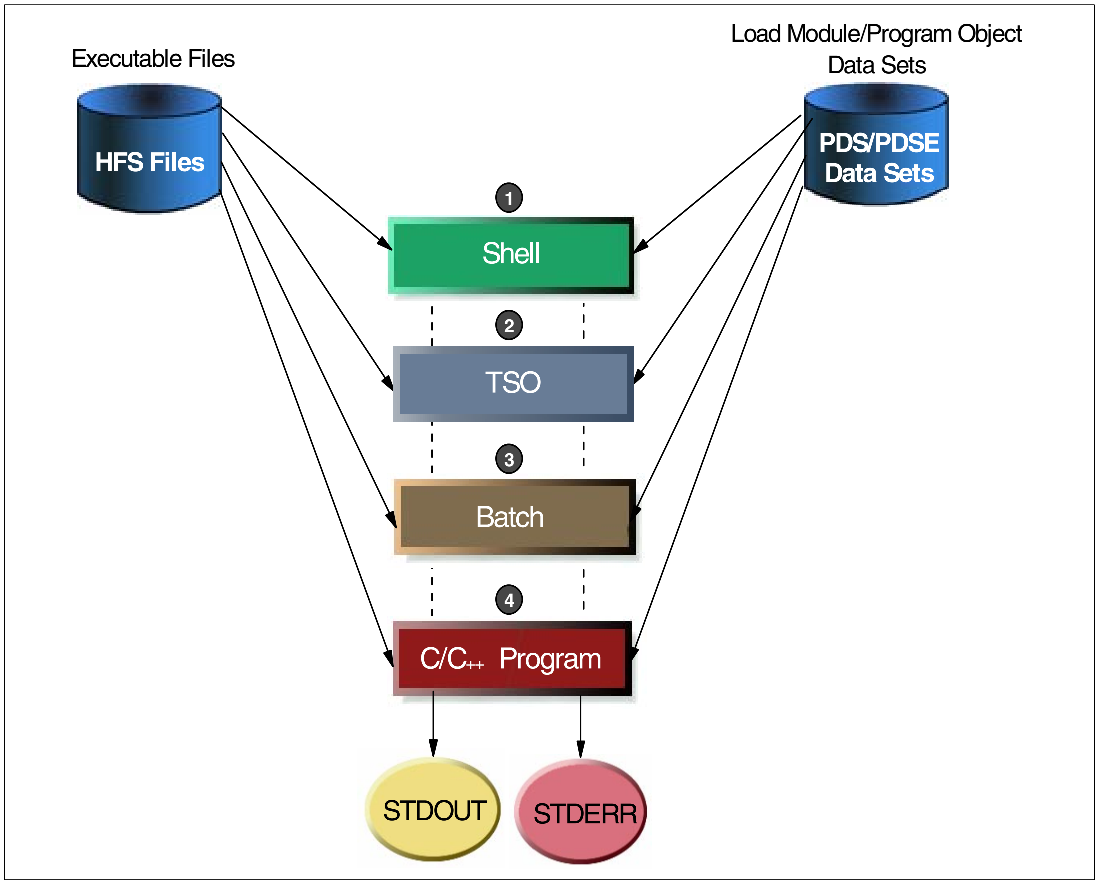


Figure 2-7 Four ways of invoking an executable

Let's look at specific examples.

2.4.1 Execution from a shell

From a shell session, an executable named types can be invoked from the directory in which it resides by entering:

```
$ ./types
```

In this case, the standard output (stdout) and any possible standard error (stderr) output will be written to the shell.

The executable TYPES in dataset RALPHB.LIB.OUT can be invoked by entering:

```
$ tso "CALL 'RALPHB.LIB.OUT(TYPES)'"
```

In this case, the stdout and stderr will be written to the terminal.

2.4.2 Execution from TSO

From TSO, first allocate two files for stdout and stderr messages via the TSO ALLOCATE command:

```
ALLOCATE FILE(STDOUT) PATH('/u/ralphb/std.out') PATHMODE(SIRWXU)
         PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHDISP(KEEP,DELETE)

ALLOCATE FILE(STDERR) PATH('/u/ralphb/std.err') PATHMODE(SIRWXU)
```

```
          PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHDISP(KEEP,DELETE)
```

Then an executable in a file, types, can be invoked by entering the command:

```
BPXBATCH PGM /u/ralphb/types
```

In this case, the stdout and stderr will be written to the files /u/ralphb/std.out and /u/ralphb/std.err, respectively.

First allocate the input dataset, RALPHB.LIB.OUT, via the TSO ALLOCATE command:

```
ALLOCATE DA('RALPHB.LIB.OUT') OLD
```

Then the executable TYPES in dataset RALPHB.LIB.OUT can be invoked by entering:

```
CALL 'RALPHB.LIB.OUT(TYPES)'
```

In this case, the stdout and stderr will be written to the terminal screen.

2.4.3 Execution via batch

- An executable in a file, types, can be invoked by submitting the following job:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
 //    CLASS=A,MSGCLASS=H,TIME=1
 //STEP1    EXEC PGM=BPXBATCH,PARM='PGM /u/ralphb/types'
 //STDOUT   DD PATH='/u/ralphb/std.out',PATHMODE=SIRWXU,
 //            PATHOPTS=(OWRONLY,OCREAT,OTRUNC)
 //STDERR   DD PATH='/u/ralphb/std.err',PATHMODE=SIRWXU,
 //            PATHOPTS=(OWRONLY,OCREAT,OTRUNC)
```

  In this case, the stdout and stderr will be written to the files /u/ralphb/std.out and /u/ralphb/std.err, respectively.

- An executable in dataset RALPHB.LIB.OUT can be invoked by submitting the following job:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
 //    CLASS=A,MSGCLASS=H,TIME=1
 //STEP1    EXEC PGM=TYPES
 //STEPLIB  DD DSN=RALPHB.LIB.OUT,DISP=OLD
```

  In this case, the stdout and stderr will be written to the SPOOL.

2.5 CICS

The Customer Information Control System (CICS) is the IBM general purpose online transaction processing (OLTP) software. It is a powerful application server that runs on a range of operating systems, from the smallest desktop to the largest mainframe.

It is flexible enough to meet your transaction processing needs, whether you have thousands of terminals or a client/server environment with workstations and LANs exploiting modern technology such as graphical interfaces or multimedia.

CICS takes care of the security and integrity of your data, while also looking after resource scheduling, thus making the most effective use of your resources. CICS seamlessly integrates all the basic software services required by OLTP applications, and provides a business application server to meet your information processing needs of today and the future.

Typical transaction processing applications include:

- Retail distribution systems
- Finance—banking, insurance, and stockbroking systems
- Order entry and processing systems
- General ledger systems
- Payroll systems
- Automatic teller machines
- Airline reservation systems
- Process control systems

2.6 DB2

DB2 is IBM's relational database management system (RDBMS). The traditional mission of DB2 is to deliver the power of the SQL language for decision support applications and the performance required for online transaction processing.

DB2 stores and provides access to a high proportion of the world's commercial data. In addition to being available under z/OS, DB2 is also available on a number of platforms including AIX, HP-UX, OS2, SCO Unixware, Windows 95/98/NT.

2.6.1 Background on a precompile operation

This section briefly outlines a methodology for porting a database application from a UNIX platform to DB2 under z/OS UNIX System Services. We concentrate on those applications which use statically embedded SQL statements. In this section, we specifically examine what must be done to port those applications to DB2 under z/OS.

Since we are assuming our source code contains embedded SQL statements, it needs to be *precompiled with the DB2 precompiler* before compiling and binding it. The precompiler reads this source code from an HFS file and then uses OCOPY to copy the source to a temporary MVS dataset. The precompiler uses the source to generate two output datasets: the *modified (C) source module* and the *database request module* (DBRM):

- In the modified source module, the EXEC SQL statements are replaced by calls to the DB2 language interface.
- The DBRM contains all the EXEC SQL statements that were extracted from the source program.

The modified source code is written to a second MVS dataset by the precompiler, and then the precompiler uses OCOPY to copy it to an HFS file. A standard C compile and bind can now be used to produce an executable from this modified source.

However, the precompile can optionally either continue by binding a plan, or it can stop after it creates the DBRM file. Figure 2-8 on page 35 illustrates this precompile process.

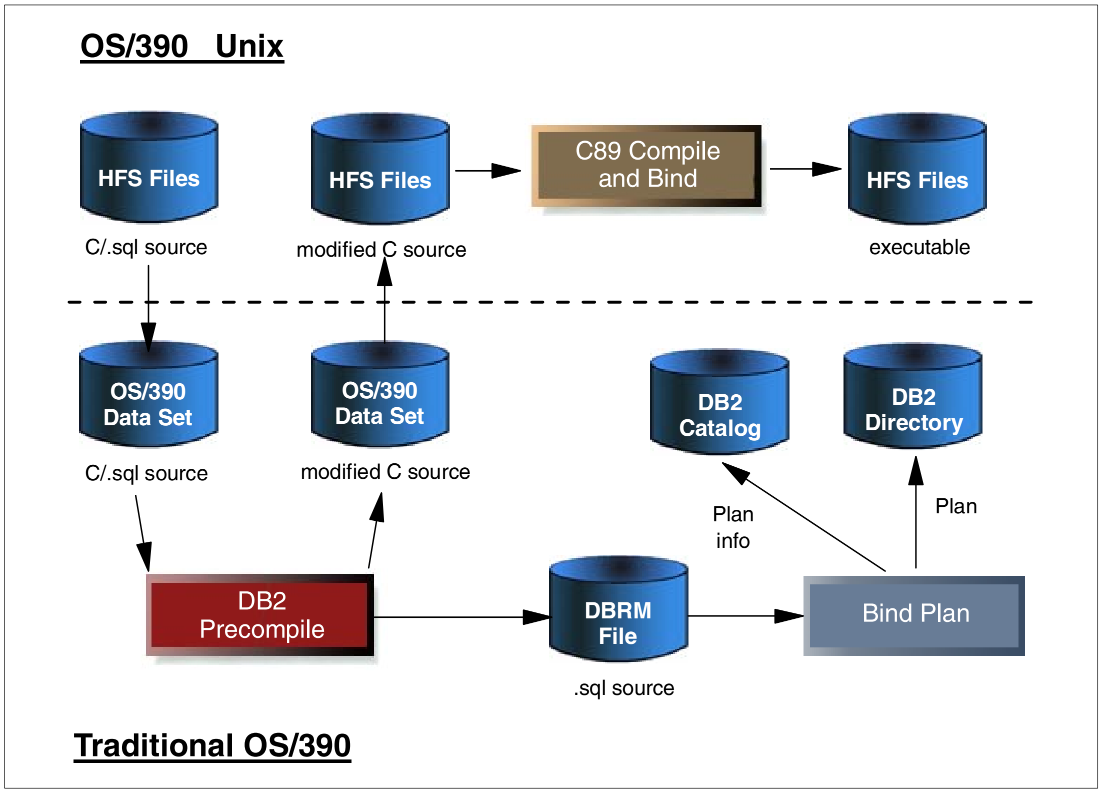


Figure 2-8 Precompile flow

Since the original source program is separated into two datasets, DB2 inserts a *consistency token* into both datasets. This consistency token is used at run time to insure that the two datasets originated from the same source program. This token is passed from the modified source to the load module during the compile and bind process.

The bind process transforms the DBRM into DB2 interpretable code. The output of bind is called a *plan*. A consistency token is copied from the DBRM to the plan at that time. This consistency token is used by DB2 to check against the consistency token of the load module at execution time; the user does not have to do any special setup to associate the load module with the plan prior to execution time.

In the bind/plan process, multiple DBRMs can be included in one plan. Since one DBRM consists of one program, a plan can contain information on SQL statements for multiple programs. When a program is a common routine shared by multiple applications, one DBRM is bound into multiple plans.

The IBM Redbook *Porting Applications to the OpenEdition MVS Platform*, GG24-4473 contains a detailed discussion on using the DB2 precompiler to port source code containing embedded SQL statements to the z/OS platform.

Before invoking the precompiler, several parameters and REXX EXEC variables must be properly set:

The precompiler requires two input parameters:

- *in* is the HFS input file to the DB2 precompiler.
- *ou* is the HFS output file from the DB2 precompiler.

It is also possible to code two optional parameters to the precompiler:

- *pa* is the path for both the input and output files.
- *pl* is the name of the plan to be created.

Two REXX EXEC variables must also be set in the precompiler EXEC before it is invoked:

- The DB2 subsystem name defined in the SYS1.PARMLIB member, IEFSSNxx.
- The high level qualifier of the DB2 datasets.

As we noted, both pa and pl are optional. If you specify the fully qualified path name when you code in and ou, pa should be omitted. If you want to bind a plan when you invoke the EXEC, code pl. If you omit pl, the EXEC will not bind a plan.

For example, suppose file db2.i contains a C source program with embedded EXEC SQL statements. Assume you want to invoke the precompiler from an OMVS shell. Assume further that the DB2 subsystem name and dataset high level qualifier have been coded in a precompiler EXEC named precomp.

- To run the precompiler and bind a plan, issue:

  ```
  tso precomp pa=/u/ralphb in=db2.i ou=db2.c pl=db2cplan
  ```

- To run the precompiler but not bind a plan, issue:

  ```
  tso precomp pa=/u/ralphb in=db2.i ou=db2.c
  ```

Now we can use our knowledge of the precompiler to port a database application.

2.6.2 DB2 precompile example

The prior section provides sufficient background to enable us to look at an example. But we need sample source module containing embedded SQL statements.

We select a database application, a “Modified Version of DB2 IVP Program for testing CAF,” written in C for this purpose. It was previously used as part of the Installation Verification Procedure (IVP) for DB2 on z/OS. “C code with embedded SQL” on page 146 contains a copy of the source code for this program.

A copy of the precompiler REXX EXEC which we used in porting this application is given in “Precompile REXX EXEC” on page 154. Note that we used the SY system variable at the beginning of the EXEC to define the name, DBS3, of our DB2 subsystem. The db2hlq system variable was also used to define the name of the DB2 high level qualifier, DSN610, to the EXEC.

The initial C source code containing the EXEC SQL statements was in the file, db2cpgm2.i. We decided to name the modified C source file, db2cpgm2.c, and we elected to build a plan with the name, db2cplan.

Thus, to invoke the precompiler, we went to an OMVS shell and entered:

```
tso precomp pa=/u/ralphb in=db2cpgm2.i ou=db2cpgm2.c pl=db2cplan
```

To perform a compile and bind on the modified source file, db2cpgm2.c, we used the JCL jobstream:

```
//RALPHBA  JOB (999,POK),'BEST, R',MSGLEVEL=(1,1),REGION=6M,
//    CLASS=A,MSGCLASS=H,TIME=1
//STEP1    EXEC EDCCB,CPARM=’DEF(_ALL_SOURCE)’
//COMPILE.SYSIN  DD PATH='/u/ralphb/db2cpgm2.c'
//BIND.SYSLMOD DD PATH='/u/ralphb/db2cpgm2',PATHMODE=(SIRWXU),
```

```
//       PATHOPTS=(ORDWR,OCREAT,OTRUNC)
//BIND.SYSLIB  DD
//           DD
//           DD DSN=DSN610.ADSNLOAD,DISP=SHR
//           DD DSN=DSN610.SDSNLOAD,DISP=SHR
```

2.6.3 Other DB2 considerations

Following are other considerations for building DB2 executables.

The db2pb Script

The precompiler can also be invoked by using a shell script, **db2pb**, which is available via download from the Tools and Toys page on the Web site:

http://www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1toy.html

This has one small advantage over the approach we used here; it allows the precompiler to read input directly from an HFS file and write output directly to an HFS file. It eliminates the need for the creation of two MVS datasets.

The Call Level Interface (CLI)

There is another method of designing database applications which does not require the use of a precompile and bind operation during the port process. The X/Open Company and the SQL Access Group jointly developed a specification for a callable SQL interface referred to as the x/Open Call Level Interface.

The goal of this interface is to increase the portability of applications by enabling them to become independent of any one database product vendor's programming interface. The CLI interface is designed to use *function calls* rather than embedded SQL statements.

Therefore, a DB2 CLI application does not have to be precompiled or bound but instead, uses a standard set of functions to execute SQL statements and related services at run time.

Access to DB2 using Net.Data

One of the main ways of accessing DB2 from a UNIX platform is through the DB2 World Wide Web gateway. This gateway uses a Common Gateway Interface (CGI) programming model to provide an interface from a Web Server to DB2.

This approach uses a CGI runtime engine which processes the input from HTML forms on the Web and sends SQL commands to a DB2 system specified in a Net.Data Application. Net.Data provides database connectivity for Internet and intranet applications.

This application consists of a macro file containing HTML input and report form definitions, SQL commands, and variable definitions.

2.7 IMS

The Information Management System (IMS), along with its database manager (IMS DB) and its Transaction Manager (IMS TM), makes accurate, consistent, timely, and critical information available to end users. IMS TM provides high-volume, high-performance, high-capacity, low-cost transactions for both IMS DB and DB2 databases. It supports many terminal sessions at extremely high transaction volumes. IMS should be used when immediate access to mission-critical information is imperative.

3

z/OS UNIX development tools

This chapter addresses application development from a task point of view. The focus of this section is on UNIX shell interaction and the C/C++ programming languages, but is not limited to these. The aspects addressed are:

- Archiving, compression, and text translation
- Compilers and associated tools
- Editors
- Make tools
- Source code control
- Code analysis
- Debugging
- Shells, scripting tools, and command languages
- Performance analysis and tuning

Besides the standard z/OS, open source, and priced tools described in this chapter, there are many other z/OS development tools.

3.1 Archiving, compression and text translation

There are many tools in the computer industry today that can archive and compress files. *Archiving* is the process of packaging directories and files into a single, larger file. *Compression* is the process of *squeezing* a single file into a smaller space, reducing both disk space and the time needed to move a large file. Often archiving and compression are used in tandem, but the difference between the two should be considered.

Typically, archives and compressed files are moved between ASCII systems, so when they are *wound* and *unwound*, no text conversion need take place; the text is represented in the same encoding scheme. With z/OS, text files are encoded in EBCDIC, so an additional factor must be considered when moving archives with text from ASCII-based systems.

3.1.1 Archive formats

The most common archive formats are:

- tar - This format is the most common on UNIX systems. There are actually two tar formats: the original UNIX tar, and the newer USTAR format defined by the POSIX 1003.1 standard. However, the two are relatively compatible, so the difference is usually not noted by the user.
- zip - This is the most common format on the PC.
- cpio - This is the legacy UNIX format, but is deprecated in favor of the USTAR format.

When an archive is created, it is sometimes described as being *tarred* or *wound*. When it is extracted, it is often described as being *untarred* or *unwound*.

3.1.2 Compression formats

The most common compression formats are:

- Lempel-Ziv (or LZW) - This format is used with the UNIX **compress** and **uncompress** commands.
- gzip - This format is used with GNU's **gzip** and **gunzip** commands.
- zip - This format is used with PC's **zip** and **unzip** commands, and the many popular GUI-driven zip manipulation tools.

3.1.3 Text translation

When text files are transferred between systems that store readable text differently, the text must be translated to the appropriate encoding. On z/OS, text data is typically stored in EBCDIC. On the PC and UNIX, text is typically in ASCII. The command **iconv** is a standalone text conversion tool that can be used to operate on *individual files*, not archives or compressed files.

The **pax -o** flag allows text conversion to be done in conjunction with archiving and compression. For example, if you have obtained a tar file with the text in ASCII (assuming the standard ASCII code page ISO 8859-1), and you want to unwind it on z/OS in EBCDIC (assuming the standard EBCDIC code page IBM-1047), you would issue the following command:

```
$ pax -rzf myASCIIpkg.tar.Z -o from=ISO8859-1,to=IBM-1047
```

Conversely, if you wanted to create a package on z/OS destined for an ASCII system, you would use the command:

```
$ pax -wzf pkg4ASCII.tar.Z -o from=IBM-1047,to=ISO8859-1 directoryToTar
```

3.1.4 Archive, compression and text translation commands

The addition of the **-z** flag to **tar** allows archives to be wound/compressed or unwound/uncompressed in the same step. The **pax** command is a new POSIX standard to do all that **tar** and **cpio** do and more.

The most common compression format for open source software is **gzip**. The **zip** and **unzip** commands are the de facto archive and compression format on the PC, though it is not common to find this format on z/OS. See Table 3-1 for a summary of archive and compression commands and formats.

Table 3-1 Summary of archive and compress commands

Command	Archive format	Compression format	File name convention
tar	tar	LZW (-z flag)	*.tar - just archive *.tar.Z - compressed archive
cpio	cpio	none	*.cpio (not common)
pax	tar or cpio	LZW (-z flag)	*.tar - archive *.tar.Z - compressed archive
gzip/gunzip	none	gzip	*.gz
zip/unzip	zip	zip	*.zip
compress/ uncompress	none	LZW	*.Z

To avoid having to uncompress and unwind in two steps, some **tar** and **pax** commands have new flags available to do both steps. However, the compression formats can differ.

On z/OS, the **tar** and **pax** commands accept the -z flag and either compress or uncompress using the Lempel-Ziv algorithm (and preclude the need for the **compress** or **uncompress** commands). On Linux, the -z flag to the **tar** command will perform gzip compression or uncompression.

To perform gzip decompression and unwind an archive in one step, the UNIX pipe command can be used. For example, to unwind the ASCII archive *example.tar.gz* which is also gzip-compressed, the following command can be used:

```
$ gzip -c -d example.tar.gz | pax -o from=ISO8859-1,to=IBM-1047 -r
```

The commands **tar**, **cpio**, **pax**, **compress** and **uncompress** are standard with z/OS. GNU's **gzip** and the PC's **zip** and **unzip** (Info-ZIP is the freeware version) must be obtained and installed on z/OS. GNU's **gzip** can be downloaded from the Web starting at:

http://www.gnu.org/software/gzip/gzip.html

Info-ZIP, which consists of the **zip** and **unzip** commands, can be obtained starting at the Tools and Toys "Ported Tools" Web site:

http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html

Also, the redbook *Open Source Software on z/OS and OS/390 UNIX*, SG24-5994, includes a CD with gzip, Info-ZIP and many other open source software packages on it. See 3.10, "Open source software" on page 53.

Sometimes an archive contains a mix of text in ASCII and binary data. Unwinding these properly is a problem because it cannot be done in one step. The script *ext* was written to handle files like this. It first extracts all files in text, thereby doing an ASCII-to-EBCDIC translation. It then re-extracts all binary files with no conversion. Again, see 3.10, “Open source software” on page 53, for more information.

3.2 Compilers and associated tools

By far, the most commonly used language on UNIX is C. C++ is an extension to C that adds many object-oriented features.

Java is also important; refer to the redbook *Experiences Moving a Java Application to OS/390*, SG24-5620 for more information. That publication was developed in a similar fashion to this redbook, except it focuses on Java instead of C++. The PDF is on the Web at:

`http://www.redbooks.ibm.com/abstracts/sg245620.html`

The C and C++ compilers available for OS390 UNIX are:

- z/OS C/C++ compiler
- SAS/C Cross-Platform Compiler and the SAS/C Cross-Platform C++ Development System
- GNU’s gcc
- Systems/C from Dignus; see:

  `http://www.dignus.com/`

Other tools often used in conjunction with the C/C++ compiler are **`lex`** and **`yacc`**, or the open source counterparts, **`flex`** and **`bison`**.

3.2.1 z/OS C/C++ compiler

Because the native C/C++ compiler is crucial to porting UNIX applications, we dedicate an entire chapter to it; refer to Chapter 4, “z/OS C/C++ compiler” on page 57.

3.2.2 SAS/C and C++ Cross-Platform Compiler

We did not have access to the SAS compiler, but it has been observed on the MVS-OE list server that running this cross-compiler on other systems (often the PC) can result in improved compile times. This is especially true if the S/390 hardware you are running on is low end (P/390, for example).

An overview of this system was found on the Web at:

`http://www.sas.com/products/sasc/`

3.2.3 GNU’s gcc and g++

A port of GNU’s gcc was done by David Pitts of the Cox corporation. The executables and source code are on the Web at:

`ftp://www.cozx.com/pub/gnu`

No port of g++ to z/OS is known.

3.2.4 Lex and yacc

The tools **lex** and **yacc** were originally written to do compiler development, but they can be used for many types of applications that have to process a language, or some other type of application-specific configuration file.

The **lex** tool does lexical analysis, which is the first stage of a compiler. It takes a source file, typically with the suffix .l, and generates the C code for the function yylex() that will convert the source language into tokens. Therefore, the process of using yylex() to generate lexical tokens from an input file is sometimes called *tokenizing*. If an application just needs to tokenize a source file, it calls yylex() in the appropriate place.

The acronym "yacc" stands for Yet Another Compiler Compiler. It allows syntactic analysis to be done, which is the second stage of a compiler. The **yacc** command takes as input a source file specific to yacc, typically with the suffix .y, which is the *grammar* for a language. From this grammar, it generates C code for the function yyparse(). This function itself calls yylex(), so yacc is almost always used in conjunction with lex, and the tokens returned by yylex() are used as input to the yacc input file.

3.2.5 Flex and bison

The tools **flex** and **bison** are GNU's replacement for **lex** and **yacc**. The term **flex** stands for Fast LEX. It performs the same function as **lex** and is purported to be much faster. Similarly, the tool **bison** is a replacement for **yacc** (so we assume a bison is faster than a yak).

These tools are available in the open source software distribution on the IBM Tools and Toys Web site at:

`http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html#opensrc`

They are also available on the open source software CD; see 3.10, "Open source software" on page 53.

3.3 Editors

The following editors are available on z/OS:

- UNIX editors - vi, ed
- The MVS ISPF editor
- GNU's Emacs
- Nedit

3.3.1 UNIX editors - vi, ed

The traditional editors on UNIX are ed and vi. While ed has been described as a line editor, vi has been described as a full-screen line editor.

3.3.2 The MVS ISPF editor

The traditional editor used on MVS is the ISPF editor. This has been made available to z/OS programmers via an OMVS session and is named OEDIT. However, this editor is *not* available via a UNIX telnet or rlogin session since it uses the 3270 data stream. The command OEDIT is described in *OS/390 UNIX System Services User's Guide*, "Using ISPF to Edit an HFS File."

3.3.3 GNU's Emacs

IBM and MKS worked together to port Emacs (Editor MACroS) to z/OS UNIX. Emacs is a popular alternative to vi and ed because it is often considered much easier to use. In Emacs you don't have to worry whether you are in command mode or text mode, as you do with vi. For more information on how to use Emacs, see *Learning GNU Emacs, 2nd Edition* by Debra Cameron, Bill Rosenblatt and Eric Raymond.

Emacs is available in the open source software distribution on the IBM Tools and Toys Web site at:

```
http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html#opensrc
```

3.3.4 Nedit

Nedit is a standard GUI-type editor for programs and normal text files. It has been written to make users of MacOS and Windows editors feel as much at home as possible. It provides all the components of a full-screen editor, such as standard buttons, menus, and mouse support.

It is based on the X Window system, and therefore requires an X server to be running on your desktop. If you have a UNIX desktop, you probably have an X server running. If you have a Windows desktop, Hummingbird's Exceed is often used (it is a priced product).

There is also a "try and buy" X server available from MicroImages. Click **MI/X 2.0 for Windows** at:

```
http://www.microimages.com/freestuf/
```

Once the X server is running, you must use the DISPLAY environment variable. For example:

```
$ export DISPLAY=10.0.0.1:0
```

Then you can run **`nedit`**, or any other X client, and display the output on your desktop.

Nedit is available on the z/OS UNIX Tools and Toys page (ported tools) at:

```
http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html
```

3.4 Make tools

The tool make is used by developers to build programs. It reads a specific input file, typically named makefile, to know how to build the program. z/OS ships the command **`make`** with the operating system, and the GNU version of make (sometimes called **`gmake`**) has been ported to z/OS UNIX.

Make has always been a mainstay in the UNIX development toolbox. It is invoked via the command **`make`**, followed by zero or more targets which are the "things" you want to build. Usually the programmer puts the main executable that the code in the current directory is to build as the first rule. Therefore, the command **`make`** with no arguments will build the program whose source code is in the current directory. Then there are ancillary make rules that are common, such as **`make clean`** to clean up extraneous files that can be rebuilt, or **`make depend`** to build dependencies from the source code.

A list of some of the differences between **`gmake`** and **`make`** is included in Appendix B, "Comparison of z/OS and GNU compilers and make tools" on page 159.

3.4.1 z/OS make

The **`make`** command is documented in detail in *OS/390 UNIX Systems Services Command Reference*, SC28-1892.

3.4.2 GNU make

GNU's version of make, formerly named **`gmake`**, but now commonly named **`make`**, has features not found in standard **`make`**. It is often required to port UNIX applications to z/OS because the application's makefiles require it and reworking them to work with z/OS's **`make`** would not be feasible. An excellent description of GNU make is on the Web at:

```
http://apache.btv.ibm.com/gnu/make.html
```

`gmake` is available in the open source software distribution on the IBM Tools and Toys Web site at:

```
http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html#opensrc
```

3.4.3 Makedepend

The tool **`makedepend`** works in conjunction with **`make`** and makefiles to dynamically create a set of header file dependencies at the bottom of a makefile. In order to properly build the program, make must have a complete, current set of rules that are composed of dependencies and recipes. **`makedepend`** now ships with the z/OS C/C++ compiler as of V1R2.

Additionally, a copy of **`makedepend`** is included with itools, which is available on the open source software CD; see 3.10, "Open source software" on page 53.

3.4.4 Autoconf, m4 and configure

The commands **`Autoconf`**, **`m4`** and the resulting script **`configure`** are tools for writing portable software packages. The following description of **`Autoconf`** was taken from the GNU Web site:

```
http://www.gnu.org/software/autoconf/autoconf.html
```

Autoconf is an extensible package of m4 macros that produce shell scripts to automatically configure software source code packages. These scripts can adapt the packages to many kinds of UNIX-like systems without manual user intervention. Autoconf creates a configuration script for a package from a template file that lists the operating system features that the package can use, in the form of m4 macro calls.

Producing configuration scripts using Autoconf requires GNU m4. You must install GNU m4 (version 1.1 or later, preferably 1.3 or later for better performance) before configuring Autoconf, so that Autoconf's configuration script can find it. The configuration scripts produced by Autoconf are self-contained, so their users do not need to have Autoconf (or GNU m4).

Also, some optional utilities that come with Autoconf use Perl, TCL, and the TCL packages Expect and DejaGNU. However, none of these are required in order to use the main Autoconf program.

The autoconf and m4 packages are available in the open source software distribution on the IBM Tools and Toys Web site at:

```
http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html#opensrc
```

The **`configure`** script created by **`autoconf`** is normally shipped with the software package. Example 3-1 shows a diagram of the entire process.

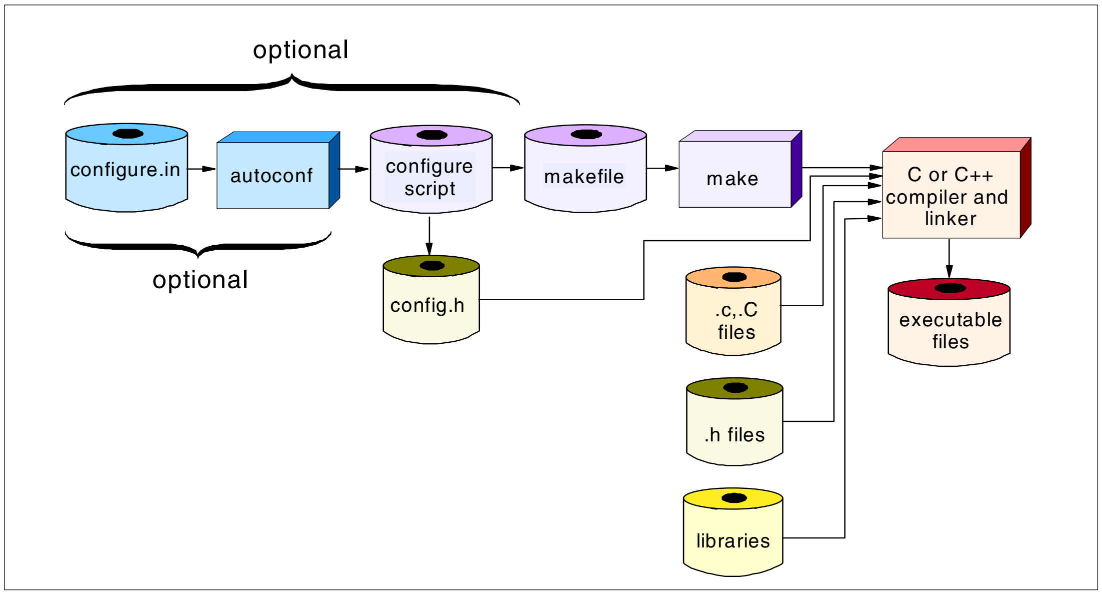


Figure 3-1 Overall build process

3.5 Source code control tools

When a single programmer is working on a project, it is *usually* easy to maintain the set of source files that comprise the code. When two or more programmers are working on the same project, however, it is prudent to use a source code control system so only one programmer has read/write access to any given file.

The following source code control tools are commonly used on UNIX:

- SCCS - the "standard" UNIX source code control system
- RCS - GNU's revision control system
- CVS - a client/server source code control system

None of these systems is standard with z/OS UNIX, but RCS and CVS (local and client mode) are available on the open source software CD; see 3.10, "Open source software" on page 53.

3.5.1 The UNIX Source Code Control System (SCCS)

Though this system is not supported by z/OS, SCCS is worth mentioning briefly since it is standard on many UNIX systems. There is a set of commands that comprises the SCCS suite. These commands are used to maintain source code as a set of original files and changes or "deltas" to each file. The commands that are commonly used are **`sccs`**, **`get`**, **`delta`**, **`prs`** and **`what`**.

3.5.2 GNU's Revision Control System (RCS)

RCS is a freely available file revision control system that was developed by Walter Tichy in 1982 at Purdue University. It is now distributed by the Free Software Foundation and maintained by Paul Eggert. RCS can be used by a single developer or, more typically, to coordinate groups of people working on common source files.

RCS consists of the four frequently used shell commands **`ci`**, **`co`**, **`rlog`**, and **`rcsdiff`**, as well as the less common commands **`srcs`**, **`ident`**, **`rcsclean`**, and **`rcsmerge`**.

3.5.3 CVS: A client/server source code control system

The Concurrent Version System (CVS) is a free client/server source code control package that has been ported to many systems. It is the state-of-the-art source code control system for developing open source software. There is information on this package on the Web at:

`http://www.cyclic.com/cvs/info.html`

Also, a good description of the value of this tool for developing software is in the paper *Open Source as a Business Strategy* by Brian Behlendorf, on the Web at:

`http://www.oreilly.com/catalog/opensources/book/brian.html`

It is also available on the open source software CD; see 3.10, “Open source software” on page 53. However, only local and client mode work on this port. Server mode is not operational.

3.6 Code analysis

Code analyzers are tools that read through source code and report on certain aspects of the code, but do not compile it into object code. Depending on the analyzer, many aspects of the code get reported on. The tool lint is a simple C code analyzer, or “C verifier”, that is standard with most UNIXes. This tool does *not* ship with z/OS, though it does not seem to be missed.

3.7 Debugging

A debugger is a tool that allows the developer to step through his code to find where problems are arising. Two debuggers are available on z/OS: dbx and the “Debug Tool.”

3.7.1 Dbx

The conventional command line UNIX debugger is dbx. It is standard with z/OS UNIX. There is a Web site describing the z/OS version at:

`http://www.s390.ibm.com/oe/dbx/dbx.html`

This site has many resources useful in working with dbx. A more direct reference to the dbx commands for various levels of the OS/390 operating system is on the Web at:

`http://www.s390.ibm.com/oe/dbx/dbxincmds.html`

It is also documented in *OS/390 UNIX Systems Services Command Reference*, SC28-1892.

3.7.2 The Debug Tool

The Debug Tool provides the common tools you would expect from a source-level debugger: the ability to step through programs, list variables and values, and set breakpoints so that the application can be run at normal speed until a specific event occurs. It allows you to debug from a graphical user interface on a Windows desktop.

The Debug Tool is available as a product feature of the z/OS C/C++ compiler. It is described on the Web at:

http://www.s390.ibm.com/dt/

3.7.3 Other debugging tools

Other third-party debugging tools are available. See the Web site:

http://www.s390.ibm.com/products/adtools/toolkit/index.html

Search for the string “Maintenance, Testing and Debugging.”

3.8 Shells and command languages

The shells that are available on z/OS are:

- The standard shell /bin/sh
- An enhanced C shell - tcsh
- The Korn shell - ksh
- The Bourne again shell - bash

The shells **sh** and **tcsh** are standard with z/OS UNIX. The **ksh** and **bash** shells must be added to z/OS UNIX. Also see 2.1, “The shell, ISHELL and OMVS interfaces” on page 20 for a more detailed discussion of each z/OS access method.

The standard shell - /bin/sh

The z/OS UNIX shell, **/bin/sh**, is based on the Korn shell and is upwardly compatible with the Bourne shell. **/bin/sh** is documented in *UNIX Systems Services Command Reference* in the section “sh - Invoke a shell.” This section is on the Web at:

http://www.s390.ibm.com/bookmgr-cgi/bookmgr.cmd/BOOKS/BPXA505/SH

An enhanced C shell - tcsh

The shell **tcsh** is an enhanced but completely compatible version of the Berkeley UNIX C shell, **csh**. It is a shell designed to have syntax and statements similar to the C Programming language. This shell has been available on OS/390 since V2R9 and is available on some previous releases as a no-priced feature.

It includes many features not found in the standard C shell including a command-line editor, interactive word completion and listing, and an enhanced history mechanism.

The Korn shell - ksh

The Korn shell was ported to z/OS by the author, David Korn. Directions for getting the Korn shell are on the z/OS Tools and Toys Web page at:

http://www.s390.ibm.com/products/oe/bpxa1toy.html

The Bourne again shell - bash

The **bash** shell is commonly the default on Linux systems, therefore, it is widely used. It is an sh-compatible shell that incorporates features from the Korn shell (**ksh**) and C shell (**csh**). It offers functional improvements over sh for both programming and interactive use. Most **sh** scripts can be run by **bash** without modification. Improvements include:

- Command line editing
- Unlimited size command history
- Job control
- Shell functions and aliases
- Indexed arrays of unlimited size

The **`bash`** shell is available on the open source software CD; see 3.10, “Open source software” on page 53.

3.8.1 Shell techniques

Some aspects of shell interaction that can increase the productivity of UNIX systems administrators or programmers are:

- Profiles
- Aliases
- Command history editing
- Command or file completion

Profiles

A profile is a program or “script” that gets executed when you log in. It should contain commands commonly used to set up your shell environment. For the shell /bin/sh, the profile is the file $HOME/.profile. If a file name is the value of the ENV environment variable, that file is run every time a shell starts.

Aliases

Aliases are alternate names for a given command, and they are set via the **`alias`** command. For example, the command **`alias dir=ls`** has the effect of “creating” a command **`dir`**, which actually just does the command **`ls`**.

Command history editing

Command history editing allows the user to not only retrieve commands recently used, but also to edit any of those commands using a mode that emulates either the vi or Emacs editors. The commands **`set -o vi`** (often set in the user’s .profil), or **`set -o emacs`**, allow editing of commands in a fashion similar to either the vi or Emacs editors.

Command or file completion

The term “command completion” is used to describe an aspect of modern shells that allows the shell to complete the command that is currently being typed. If you type part of a file name which uniquely identifies the file and press the command completion key, the rest of the file name is typed for you. If no match is found, the terminal bell rings. If the word is already complete, a forward slashmark (/) or space is added to the end if it isn't already there.

The subtleties of command completion vary with different shells, but the concept is similar. This command completion keys for the available z/OS shells are as follows:

/bin/sh	Esc - Esc (escape twice) in emacs editing mode
	ESC - \ (escape then backslash) in vi editing mode
/bin/tcsh	Tab
bash	Tab

3.8.2 Scripting tools and command languages

On UNIX, a “script” is a command file that can be executed. Scripts are typically files that are in plain text, have one or more executable bits set, and are interpreted. The following scripting tools and command languages are available on z/OS:

- Shells
- REXX
- PERL

- awk and sed

Before we discuss each of these, we need to address the subject of support of the “magic value” convention in the first line of a script file. Often a script will start with a line similar to the following:

```
#! /usr/local/bin/perl
```

This is the magic value notation for using Perl to execute the script. Beginning in V2R8 of OS/390, the magic value convention has been supported. The format of the header is as follows:

```
#! Path String
```

where #! is the file magic number. The magic number indicates that the first line is a special header that contains the name of the program to be executed and any argument data to be supplied to it.

When the environment variable `_BPX_SPAWN_SCRIPT=yes` is set, spawn will first recognize the file magic number and process the file accordingly. If the file magic number is not found in the file's first line, spawn will treat the specified file as a shell script and invoke the shell to run the shell script.

The Perl scripting language

Perl is a highly popular language. It has been described as the “duct tape of the Internet.” See the Perl Information Web site at:

http://www.perl.org/perl.html

Perl 5.6.1 should build cleanly on z/OS UNIX. Prior to that version, Perl 5.005p3 was available in the open source software distribution on the IBM Tools and Toys Web site at:

http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html#opensrc

Prior to OS/390 V2R8, the *magic value* was not supported. For Perl scripts, the following code is necessary to accomplish the same:

```
# explicitly call perl since #! does not yet work
eval 'exec /usr/local/bin/perl -S $0 "$@"'
        if 0;
```

The REXX command language

REXX is an IBM command language originally written for VM/CMS. It has been ported to just about all IBM operating systems, including z/OS.

The publication *Using REXX and OS/390 UNIX System Services*, SC28-1905, specifically describes REXX, and is available on the Web at:

http://www.s390.ibm.com/bookmgr-cgi/bookmgr.cmd/BOOKS/BPXB603

REXX programs always start with a C-style comment (`/* comment */`). An example of a simple REXX program follows, which prints the contents of the root directory to standard output:

```
/*   rexx   */
call syscalls 'ON'
address syscall
'readdir / root.'
do i=1 to root.0
 say root.i
end
```

Awk and sed

Awk is a programming language that lets you work with information stored in files. With awk programs, you can:

- Display all the information in a file, or selected pieces of information
- Perform calculations with numeric information from a file
- Prepare reports based on information from a file
- Analyze text for spelling, frequency of words or letters, and so forth

An example of using awk is as follows:

```
binaryFiles=`pax -f foo.tar | awk '/.Z$|.gz$|.tgz$/ {print $0}'`
```

This command puts all of the files in the archive foo.tar with names ending in .Z, .gz and .tzg into the variable `binaryFiles`.

The tool sed is a “stream editor” and is often used as a script language programming tool, not as an interactive editor. An example of using the **sed** command is as follows:

```
sed -e 's/FLAGS1 = -O/FLAGS1 = -O -Wl,EDIT=NO/g' Makefile > Makefile.temp
```

This command adds another compiler flag to the line `FLAGS1 = -O` in a makefile and redirects the output to a new file, Makefile.temp.

3.9 Performance analysis and tuning tools

There are Web sites, documents and tools for tuning z/OS UNIX applications. Following are just some of the resources available:

- The z/OS UNIX “Tune up and Perform” Web site at:

  `http://www.s390.ibm.com/oe/bpxa1tun.html`

- The IBM Redbook *Tuning Large C/C++ Applications on OS/390 UNIX System Services*, SG24-5606
- The *OS/390 UNIX Systems Services Porting Guide*, chapter entitled, “After the port, focus on performance.” This document is on the Web in PDF format at:

  `http://www.s390.ibm.com/ftp/os390/oe/docs/port.pdf`

- STROBE
- The C/C++ Performance Analyzer

3.9.1 z/OS C/C++ Performance Analyzer

The Performance Analyzer is a profiling tool for C/C++ applications that can provide detailed information about where an application is spending most of its time. Close examination can reveal questionable programming practices. You can further optimize high usage routines to improve performance.

The following description was taken from the Performance Analyzer Web pages:

> The Performance Analyzer helps you understand and improve the performance of your C/C++ programs. It traces the execution of a program and creates a trace file which contains data that can be examined in several diagrams at the workstation. With the Performance Analyzer you can also generate the trace report, which is a text file that you view on the host. Sometimes this tracing is also referred to as profiling. With the trace information, you can improve the performance of a program, examine a sequence of calls leading up to an exception, and understand the execution flow when a program runs.

Performance Analyzer function trace

Function trace enables the compiler to generate hooks in the code at the following points:

- Function entry
- Function exit
- Before a function call
- After a function call

For each of these points, a time stamp is recorded. The difference between function entry and exit allows the function trace tool to profile your C/C++ code.

Using the Performance Analyzer

Here are some examples of z/OS UNIX shell commands used to compile and execute a program and turn on the Performance Analyzer for function tracing. Use the **c89** command to compile and bind a sample C source file, test.c:

```
c89 -o ./test -O -Wc,"TEST(HOOK),NOGONUMBER" test.c
```

Use the **c++** command to compile and bind the sample C++ source file, test.cxx:

```
c++ -o ./test -O -Wc,"TEST(HOOK)" test.cxx
```

Use the **export** command for setting run-time options and environment variables, for example:

```
export _CEE_RUNOPTS="PROFILE(ON,'FUNCTION,REAL')"
export __PROF_FILE_NAME=./test.trc
```

Set the STEPLIB environment variable so that Language Environment can find the Performance Analyzer module (only required if it is not included in the Link Pack Area):

```
export STEPLIB=CBC.SCTVMOD:$STEPLIB
```

Start your program as follows, and the Performance Analyzer tracing will begin at the same time:

```
test
```

Tracing can be turned off after running your program by setting the _CEE_RUNOPTS environment variable as follows:

```
export _CEE_RUNOPTS="PROFILE(OFF)
```

Alternatively, tracing can be turned off by unsetting the appropriate environment variables. A couple of sample scripts, **debon [1]**and **deboff [2]**, were written to easily turn on and off the Performance Analyzer tracing. When using these scripts, remember to use the "**.**" command **[3]** to set the environment variables in the current shell.

```
$ cat debon [1]
export _CEE_RUNOPTS="PROFILE(ON,'FUNCTION,REAL')"
export __PROF_FILE_NAME=./test.trc
$ cat deboff [2]
unset _CEE_RUNOPTS
unset __PROF_FILE_NAME
$ . debon [3]
```

After generating the trace files on z/OS, download them in binary to your workstation. Appendix F, "Performance analyzer output" on page 179 shows some sample screen shots of the Performance Analyzer.

3.10 Open source software

Open source software (sometimes called freeware) is continuing to become more important for UNIX development. Much open source software is available for z/OS UNIX, primarily from the following places:

- z/OS UNIX Tools and Toys
- GNU Web site
- The redbook *Open Source Software on z/OS and OS/390 UNIX*, SG24-5944

z/OS UNIX Tools and Toys

This site contains many open source software tools, though the source code is not available for all of them. It is divided into the following sections:

- Ported tools
- z/OS UNIX tools
- Code samples
- Other sources
- Performance tools

It is on the Web at:

`http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html`

GNU Web site

Of course, the most freeware is available on the GNU Web site. Due to EBCDIC encoding of text on z/OS, many of the packages will not build or work cleanly. However, it is still a valuable resource. The list of software packages is on the Web at:

`http://www.gnu.org/software/software.html`

Open source redbook

The redbook *Open Source Software on z/OS and OS/390 UNIX*, SG24-5944 was written to summarize the majority of freeware available for z/OS. It addresses Info-ZIP and OpenSSH, for which the software is not included. It also addresses the following packages for which the source code and binaries are available:

- apache 1.3.19 and version 2
- autoconf 2.13
- automake 1.4
- bash 2.03
- bison 1.25
- cvs 1.11.05
- diffutils 2.7
- emacs 19.34
- findutils 4.1
- flex 2.5.4a
- gmake 3.76.1
- gnats 3.113
- GNU grep 2.4.2
- groff 1.17
- gzip 1.2.4
- id-utils 3.2d
- itools 6.3
- less 358
- lynx 2.8.3
- m4 1.4

- nedit 5.1.1
- OpenSSH 2.9p1
- OpenSSL 0.9.6a
- patch 2.5.4
- Perl 5.005p3 and 5.6.1
- php 4.0.5
- RCS 5.7
- rlogin client
- rsh client
- Samba 1.9.18p1 and 2.2.1a
- sntp 1.5
- THE 3.0
- vim 5.6
- xterm 3.9.18
- zlib 1.1.3

In addition to the code, there are scripts written for most packages to make installation easier. The book is on the Web at:

http://www.redbooks.ibm.com/abstracts/sg245944.html

The software (source and binaries) is on the Web at:

http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html#opensrc

3.11 Development environments

In preceding chapters, we describe the traditional UNIX *command line-driven* environment. For most UNIX programmers, this is more than adequate. On the PC however, *GUI-driven* integrated development environments (IDEs) are the norm. For z/OS, there are a couple of third party, priced products that allow software development to be done with IDEs:

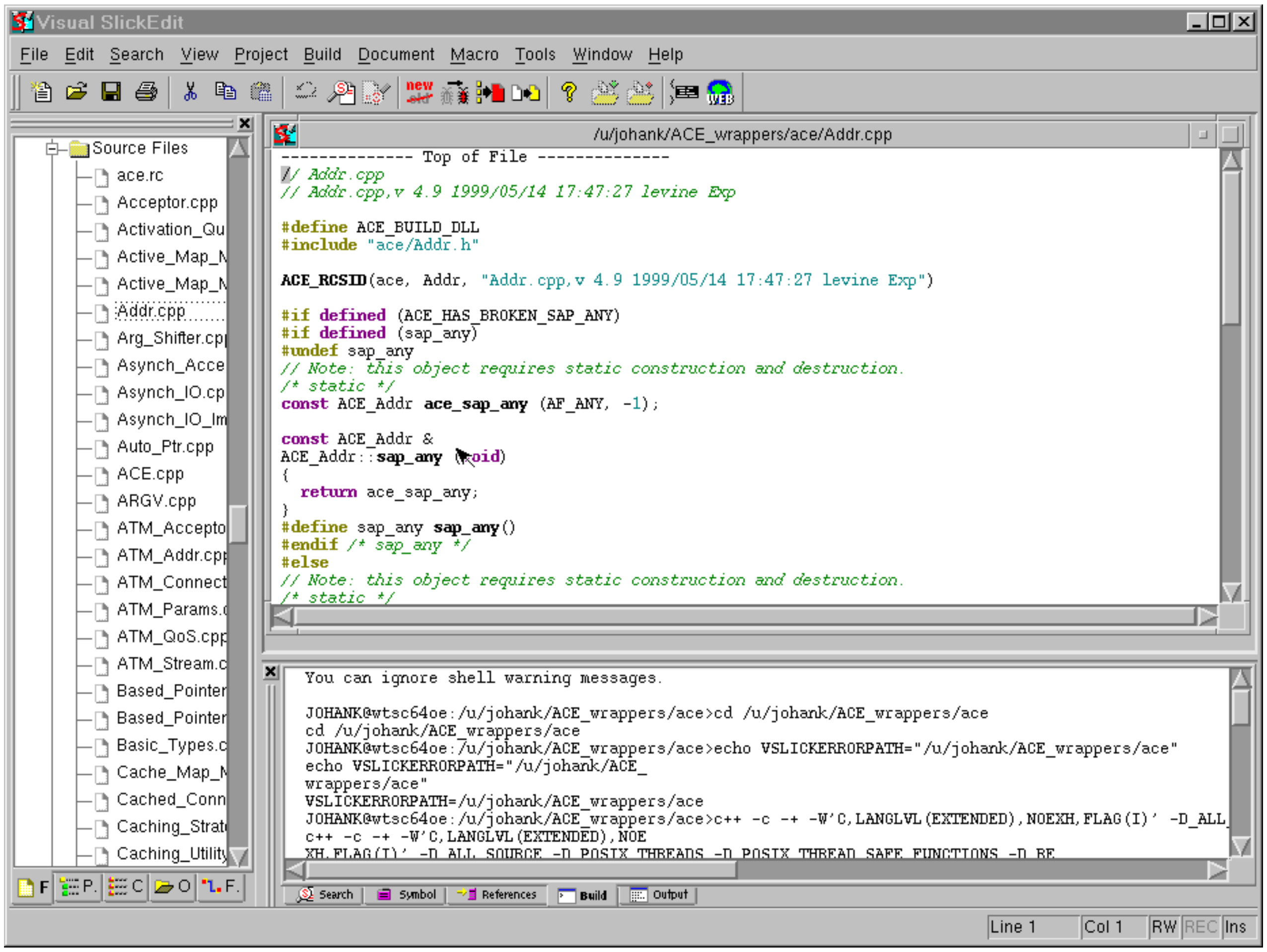
- MicroEdge's Visual SlickEdit
- Bristol's Tributary

The two use significantly different models, though in the end, both use the z/OS C/C++ compiler to create native executables.

3.11.1 MicroEdge's Visual SlickEdit

Visual SlickEdit runs as a z/OS X client and displays back to the user's desktop running an X server. We installed SlickEdit on OS/390 2.10 and exported the display to a Windows-NT desktop running Hummingbird Exceed X-server. Installation was straightforward, and the product presented a modern GUI.

A screen shot of Visual SlickEdit is shown in Figure 3-2 on page 55.



```
-------------- Top of File --------------
// Addr.cpp
// Addr.cpp,v 4.9 1999/05/14 17:47:27 levine Exp

#define ACE_BUILD_DLL
#include "ace/Addr.h"

ACE_RCSID(ace, Addr, "Addr.cpp,v 4.9 1999/05/14 17:47:27 levine Exp")

#if defined (ACE_HAS_BROKEN_SAP_ANY)
#if defined (sap_any)
#undef sap_any
// Note: this object requires static construction and destruction.
/* static */
const ACE_Addr ace_sap_any (AF_ANY, -1);

const ACE_Addr &
ACE_Addr::sap_any (void)
{
  return ace_sap_any;
}
#define sap_any sap_any()
#endif /* sap_any */
#else
// Note: this object requires static construction and destruction.
/* static */
```

```
You can ignore shell warning messages.

JOHANK@wtsc64oe:/u/johank/ACE_wrappers/ace>cd /u/johank/ACE_wrappers/ace
cd /u/johank/ACE_wrappers/ace
JOHANK@wtsc64oe:/u/johank/ACE_wrappers/ace>echo VSLICKERRORPATH="/u/johank/ACE_wrappers/ace"
echo VSLICKERRORPATH="/u/johank/ACE_
wrappers/ace"
VSLICKERRORPATH=/u/johank/ACE_wrappers/ace
JOHANK@wtsc64oe:/u/johank/ACE_wrappers/ace>c++ -c -+ -W'C,LANGLVL(EXTENDED),NOEXH,FLAG(I)' -D_ALL_
c++ -c -+ -W'C,LANGLVL(EXTENDED),NOE
XH,FLAG(I)' -D_ALL_SOURCE -D_POSIX_THREADS -D_POSIX_THREAD_SAFE_FUNCTIONS -D_RE
```

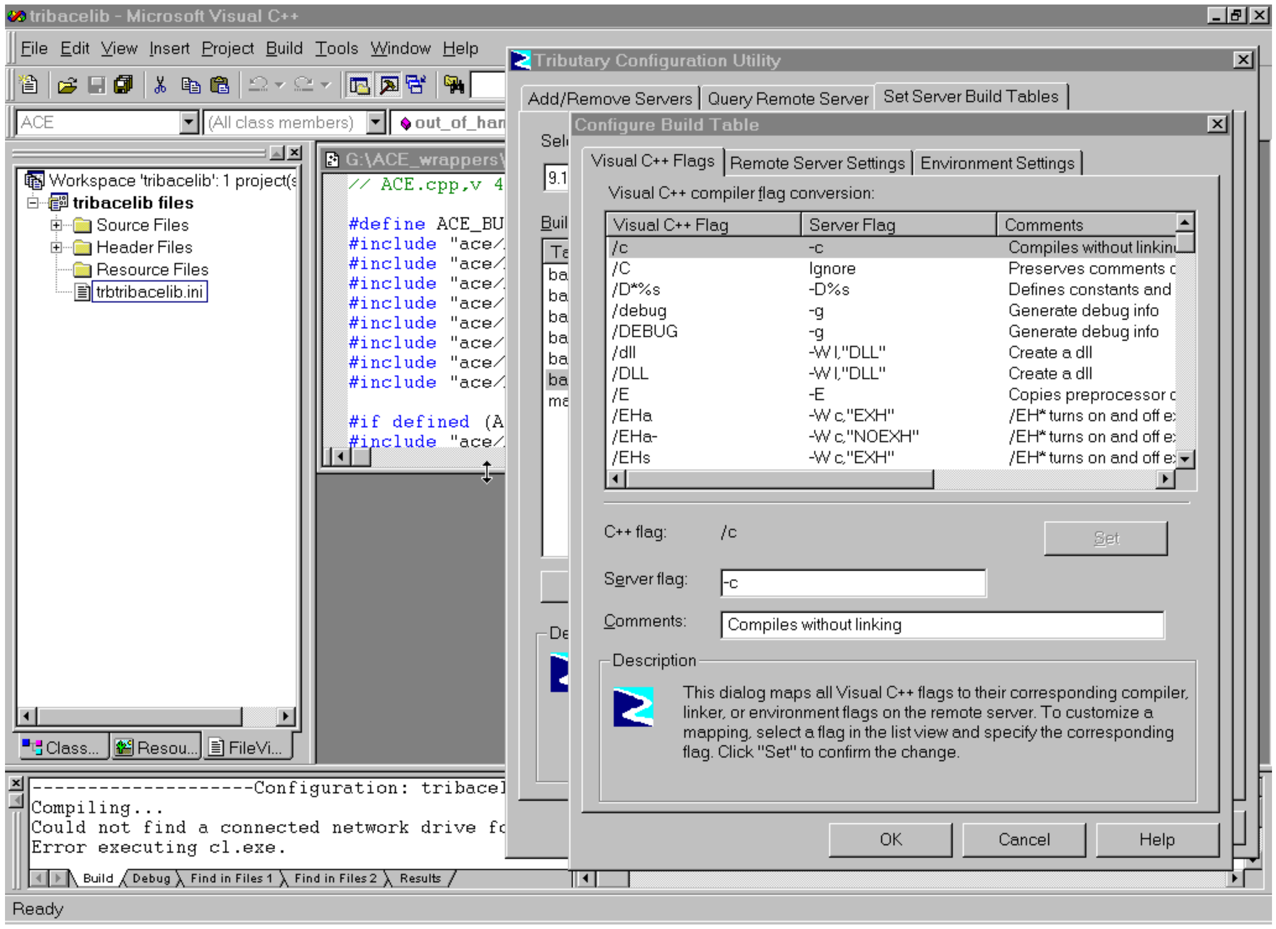
Figure 3-2 Visual SlickEdit

3.11.2 Bristol's Tributary

This product is a plug-in to MicroSoft's Visual C++ IDE. Development is done as it always is on the PC. Tributary uses a server running on z/OS to issue compile and make commands on behalf of the Visual C++ client. The Tributary client uses an NFS or Samba to access source code on the z/OS machine. Tributary translates Visual C++ flags to corresponding z/OS C++ flags.

We installed an evaluation copy of Tributary with the server on OS/390 V2R9 and the client on Windows-NT. We used Samba for file sharing between the systems. Installation was easy, though configuration is a somewhat complex operation and not intuitive for a person who is not familiar with Visual C++.

Tributary is useful for developers accustomed to the Microsoft Visual C++ environment. It is highly dependent on the response time of the NFS or Samba server on OS/390. A screen shot of Bristol's Tributary is shown in Figure 3-3 on page 56.



Configure Build Table

Visual C++ compiler flag conversion:

Visual C++ Flag	Server Flag	Comments
/c	-c	Compiles without linkin
/C	Ignore	Preserves comments c
/D*%s	-D%s	Defines constants and
/debug	-g	Generate debug info
/DEBUG	-g	Generate debug info
/dll	-W l,"DLL"	Create a dll
/DLL	-W l,"DLL"	Create a dll
/E	-E	Copies preprocessor c
/EHa	-W c,"EXH"	/EH* turns on and off e:
/EHa-	-W c,"NOEXH"	/EH* turns on and off e:
/EHs	-W c,"EXH"	/EH* turns on and off e:

C++ flag: /c

Server flag: -c

Comments: Compiles without linking

Description

This dialog maps all Visual C++ flags to their corresponding compiler, linker, or environment flags on the remote server. To customize a mapping, select a flag in the list view and specify the corresponding flag. Click "Set" to confirm the change.

```
--------------------Configuration: tribacel
Compiling...
Could not find a connected network drive fc
Error executing cl.exe.
```

Figure 3-3 Bristol Tributary

z/OS C/C++ compiler

This chapter describes the C/C++ compiler on z/OS from a UNIX point of view. It highlights some of the key differences you may encounter between z/OS and other UNIX compilers.

Note that this redbook is not focused only on doing a direct port of an existing application. Because a porting exercise tends to have more stringent requirements in terms of compiler differences, we gear our discussion here towards direct porting. Readers can easily make the adjustments necessary to fit their own situation.

You can invoke the compilers from within the z/OS UNIX shell, using MVS JCL or TSO commands (see 2.3, "An overview of compile and bind operations" on page 28). In this chapter, we focus the discussions from an z/OS UNIX perspective. All the tasks for compiling a program can also be done by using MVS Job Control Language (JCL).

At the end of this chapter, we provide examples of the compilation tasks described here. If you are not familiar with JCL, you can use these examples as a starting point to experiment with the additional capabilities provided. We do not go into detail on TSO commands. For more information, refer to *OS/390 C/C++ User's Guide*, SC09-2361.

The ways to invoke the C and C++ compilers are the same, except for the name of the command used to identify the compiler. Also note that not all compiler options apply to both C and C++.

In the following sections, we first describe the C compiler, and then discuss the C++ compiler.

4.1 The C compiler

You can invoke the C compiler from z/OS UNIX like any other UNIX platform. For example, this command compiles and binds the hello.c file in the current directory and produces an a.out executable in the current directory:

```
$ c89 hello.c
```

The following command does the same and instructs the compiler to search for header files under the ./include directory. The -O (upper caseO) option invokes the optimizer:

```
$ c89 -I./include -O hello.c
```

z/OS UNIX shell utilities conform to the Single UNIX Specification (the latest release of z/OS UNIX System Services conforms to UNIX98). As a shell utility, **c89** supports all the options described in that standard.

4.1.1 z/OS-specific options

In addition to the usual UNIX options, **c89** also supports z/OS specific options. The command syntax is as follows:

```
$ c89 -Wc,"<option>,..." hello.c
```

The `-W` indicates the beginning of the z/OS options. The letter following `-W` indicates the *phase*; `c` signifies the compiler phase, `l` (lower case L) signifies the bind or link phase, and `I` (upper case I) signifies the IPA phase. The double quote (or a single quote) is a shell meta character to avoid the shell being confused by the option syntax. The double quote is optional if there are no meta characters, like open/close parentheses, inside the option specifications.

`<option>` is the z/OS option; the specification is not case-sensitive. More than one option can be specified, separated by commas.

For example, the following command invokes the compiler with the extended language level, which provides additional features on top of Standard C:

```
$ c89 -Wc,"langlvl(extended)" -I./include hello.c
```

The `-W` specification can appear more than once, and can be used with other UNIX like options. In the preceding example, `-I` is also specified to specify an include path.

Refer to the *OS/390 C/C++ User's Guide*, SC09-2361; the chapter entitled “Options” provides a full list of z/OS options.

4.1.2 K&R C

The **c89** command provides full support for Standard C (the ANSI C89 standard). Before the C language was standardized in 1989, the C compilers in UNIX environments loosely followed a de facto standard known as K&R C (named after the original C authors, Brian Kernighan and Dennis Ritchie). To help port these older programs, z/OS supports K&R C via the **cc** command:

```
$ cc hello.c
```

Alternatively, you can use the "langlvl(commonc)" option of **c89** as follows:

```
$ c89 -Wc,"langlvl(commonc)" hello.c
```

Note that K&R was an evolving standard during the ten years leading up to the standardization of C; therefore, the `cc` command may behave differently on different platforms. To achieve portability, use **c89** for all new programs. Refer to 4.6.6, “Key differences between cc and c89” on page 74 for further details.

The same option syntax applies to all compiler shell utilities (**c89** or **cc**). According to the Single UNIX Specification, all options for shell utilities *precede* all operands on the command line. You can change this behavior and intermix options and file names by setting the environment variable _<xxx>__CCMODE to 1 (where <xxx> can be C89, CC or CXX). For example:

```
$ export _CC_CCMODE=1
$ cc -g hello.c -I./include
```

4.1.3 Compilation phases

If you look under the cover, the compilation is actually comprised of a number of phases. The simplest one consists of a *compile* phase and a *bind* phase. The compile phase produces the .o files, and the bind phase links them together to produce the executable. These phases are invoked by **c89** (or **cc**) automatically and do not usually concern the programmer, unless you need to use z/OS-specific options (the `-W` options) targeted to a particular phase.

The examples provided so far direct the `-W` options to the compiler phase, hence the letter `c` following the `-W`. To specify a bind option, use the letter `l`. For example:

```
$ c89 -Wl,”list” hello.c
```

The `LIST` option produces a binder listing directed to stdout.

Another phase is the IPA phase. IPA stands for Interprocedural Analysis, which does *whole program optimization* as opposed to *single source file optimization*; the latter is specified by the `-O` option. We discuss IPA in 4.4, “C/C++ Compiler Interprocedural analysis (IPA)” on page 68.

Note: Optimization within a source file, the `-O` option, is considered part of the compiler phase. IPA pushes optimization one step further and pulls together information from all source files (all compilation units) and does optimization across the whole program.

4.1.4 Commonly used z/OS C/C++ options

Following is a short list of useful z/OS-specific options:

ANSIALIAS This is a compile phase option that is optimization-related. When building an application for the first time on z/OS, turn this option off (that is, specify NOANSIALIAS); this is the safe setting. Read Chapter 7, “Performance and tuning techniques” on page 93 to determine whether you could turn this on. For example:

```
$ c89 -Wc,"noansialias" hello.c
```

Note: The default is ANSIALIAS for **c89** and **cxx**.

DLL This option is both a compile and bind phase option to build a dynamic link library. (They are really two options having the same name). For example, to create a DLL, invoke:

```
$ c89 -o mydll -Wc,"DLL" -Wl,"DLL" mydll.c
```

This command produces a dynamic link library (DLL) called mydll. Also see 4.2.3, “Commonly used z/OS C++ options” on page 65 for a discussion of how the DLL option differs with C++ code.

EDIT=NO This is a bind phase option. The binder is very flexible and can reprocess (rebind) its own output. It does this by inserting extra information into the program object. This option asks the binder not to do this and can reduce the size of the program.
However, you will not be able to run utilities that return information about the program object, and you will not be able to do rebinding. Also, you cannot use STROBE to measure the application’s use of system resources. Try this if you find the executable size large. For example:

```
c89 -Wl,”EDIT=NO” hello.c
```

GONUMBER A compile phase option, this inserts line number information into the executable; the line number will show up in the traceback. This is useful when debugging an abend. For example:

```
c89 -Wl,”GONUM” hello.c
```

LANGLVL A compile phase option, it takes a suboption specified within parentheses. The commonly used suboptions are: ANSI (ANSI C language features), EXTENDED (ANSI plus z/OS extended features) and COMMONC (K&R C). For example:

```
c89 -Wc,”langlvl(extended)“ hello.c
```

Note that the “long long” data type (64-bit integer) is not an ANSI C feature. If you need long long support, you need to specify `langlvl(extended)` explicitly. Also see 4.7.1, “Changes to string and wide character function prototypes” on page 81 for a discussion of `langlvl(oldstr)`, which affects string functions.

LIST This option is both a compile and bind phase option. In the compile phase, this produces an pseudo assembler listing directed to stdout. It is usually used with the SOURCE option. For example:

```
c89 -Wc,”LIST,SOURCE” hello.c
```

In the bind phase, this produces the binder listing. For example:

```
c89 -Wl,”LIST” hello.c
```

SOURCE Compile phase option,it produces a source listing directed to stdout. Example:

```
c89 -Wl,”SOURCE” hello.c
```

Two other useful bind options are `DLL` and `EDIT=NO`. Refer to 4.2.3, “Commonly used z/OS C++ options” on page 65 for details.

4.2 The C++ compiler

Starting with z/OS V1R2, the C++ compiler implementation is based on the ISO C++ 1998 Standard. The previous OS/390 V2R10 compiler is also shipped with the z/OS V1R2 compiler to aid in maintaining backward compatibility. This older compiler is only recommended if your C++ code uses the old style templates.

Up to OS/390 V2R10, the C++ compiler implementation was based on Draft Proposal X3J16/92-00091. The standardization process hsd been evolving since that draft and the C++ language was standardized in 1998. Major C++ compilers conform to the standard to various degrees. The application you are porting from may therefore require various features of the ISO C++ 98 Standard.

To help you plan a porting exercise, a comparison between the OS/390 C++ implementation and the 98 Standard is provided in 4.6.7, “Comparison between OS/390 V2R10 C++ and z/OS V1R2 C++ compilers” on page 77.

The C++ compiler can be invoked from the shell using either of the commands **cxx** or **c++**. For example:

```
$ c++ hello.C
```

c++ and **cxx** are the same utility. The command line syntax is the same as for **c89**/**cc**.

cxx/c++ recognize the .C suffix (upper case C) as C++ source file by default. If your C++ files have a different suffix, you can change this by setting the environment variable _CXX_CXXSUFFIX. For example:

```
$ export _CXX_CXXSUFFIX=cpp
$ cxx hello.cpp
```

Alternatively, you can use the + option to force the compiler to accept any suffix as C++. For example:

```
$ cxx -+ hello.cpp
```

Only source suffixes are recognized and affected by `-+`. Source suffixes are .c, .C and .cpp. By default, a lower case c (`.c`) suffix signifies a C source file. If you use a .c file in a **cxx/c++** command, the compiler will automatically switch to **c89** and process the file as a C file.

4.2.1 ISO C++ 1998 features

The z/OS V1R2 compiler is in compliance with the ISO C++ 1998 standard. The following are major changes between it and the previous C++ compilers.

Namespaces

A namespace allows the grouping of all declarations, for example (classes, templates, functions, enums and variable names). Before the namespace construct was added to C++, all symbols were added to the “global” namespace. The namespace concept helps reduce symbol name collision between user applications and libraries.

By default, all symbols not declared in a namespace are in the global namespace. This means you can reference the symbol without fully qualifying it. All the symbols in the C library are in the global namespace for backward compatibilities and are imported into the std namespace via header files such as <cstdio>, <cstring> etc. For new projects, we recommend using the fully qualified names because the code will be more portable.

For porting applications that use the old iostream library (iostream.h) to the C++ Standard iostream (iostream), we recommend the `using` directive, which sets the default namespace for a compilation unit. For example:

```
using namespace std;
```

This reduces the number of code changes required. The new iostream library has all of its symbol in the namespace std. With the `using` directive, the user does not need to fully qualify the namespace of the symbols. For example, instead of:

```
std::cout << “Hello world” << std::endl;
```

the following is sufficient:

```
using namespace std;
cout << “Hello world” << endl;
```

Run-time type identification

RTTI is a mechanism for determining the class of an object at run time and allows type conversions that are type checked at run time.

This is controlled by the compile time option RTTI. In order to use the new `typeid` and `dynamic_cast` operators, the RTTI information must be available. There is no execution speed penalty for compiling with RTTI. However, extra space is needed to store the type information. This causes the object files and executables to be larger. The amount of extra space needed is related to the complexity of the class inheritance tree. Example 4-1shows an example of using RTTI.

Example 4-1   rtti_example.C

```
#include <cstdio>
#include <typeinfo>


class A {
public:
   virtual ~A(){}
};

class B {
public:
   virtual ~B(){}
};

class C : public A, public B {
public:
   virtual ~C(){}
};


int main()
{
    {
       /*
          Part 1.1 - cross cast:

          Using the dynamic_cast operator to safely
          convert the type of an object across an
          inheritance tree.
       */

       A* a1 = new C;
       B* b1;

       // the static type of a1 is A *, convert it to B *
       b1 = dynamic_cast<B *>(a1);

       std::printf("address of b1 is %p\n", b1);

       // clean up
       delete a1;
    }

    {
       /*
          Part 1.2 - cross cast:

          Using the dynamic_cast operator to safely
          convert the type of an object across an
          inheritance tree.  In this case, the dynamic
          type of the object referenced by a1 cannot
          be cast to B, so this conversion will fail.
       */

       A* a1 = new A;
       B* b1;

       // the static type of a1 is A *, convert it to B *
       // this conversion will fail
       b1 = dynamic_cast<B *>(a1);

       // the value of b1 should be 0, because the conversion failed
       std::printf("address of b1 is %p\n", b1);

       // clean up
       delete a1;
   }

   {
       /*
          Part 2.1 - down cast:

          Using the dynamic_cast operator to safely
          convert the type of an object to a more-
          derived type.
       */

       A * a2 = new C;

       // the static type of a2 is A *, convert it to C *
       C * c2 = dynamic_cast<C *>(a2);

       std::printf("address of c2 is %p\n", c2);

       // clean up
       delete a2;
   }

   {
       /*
          Part 3.1 - typeid:

          Using the typeid operator to obtain the
          type_info structure for a polymorphic
          type.
       */

       A * a3 = new C;

       // the name of *a3's type is "C"
       std::printf("name of a3 is %s\n", typeid(*a3).name());
```

```
        // clean up
        delete a3;
    }
}
```

This example can be built with the command:

```
$ c++ -Wc,RTTI rtti_example.C
```

Boolean support

The C++ standard now defines `true`, `false` and `bool` as reserved keywords. It was common practice to `typdef` or `#define` these names in C and C++ code. If this is the case, the code must be modified to remove such definitions.

Be careful with functions that return boolean or have one or more boolean arguments. These functions now have a different *mangled name*. So if a library with such functions has been built with a previous version of the compiler, and an application built with z/OS V1R2 compiler tries to link to it, there will be unresolved symbols. If the source of the library is available, it is recommended that the library be rebuilt with the latest compiler. However, if the library is from a third party and is unavailable, it is possible to disable these boolean related keywords by using the NOKEYWORD compiler option.

Example 4-2 bool_example.C

```
typedef int bool;

const bool true=1;
const bool false=0;

int main() {
        bool a;

        a=true;
        return 0;
}
```

Build command:

```
#c++ -Wc,"NOKEYWORD(bool),NOKEYWORD(false),NOKEYWORD(true)" bool_example.C
```

Note: What is name mangling?

On z/OS, name mangling is only done for C++ symbols. It is a way of capturing the parameter type information in the symbol name. This is most important for class and function overloading.

For example, a function getnum under class B in the namespace mangling_example:

```
mangling_example::B::getnum()
```

is mangled to:

```
getnum__Q2_16mangling_example1BFv
```

Friend access declaration

Due to changes in the ISO C++ 1998 Standard, friend declaration no longer prototypes the function, it only modifies the access of that function. Therefore, the function must be declared before using it. Example 4-3 on page 65 illustrates the changes required.

Example 4-3 FriendAcess_example.C

```
/* uncomment to build on z/OS V1R2*/
/* int foo(); */

class X {
  public:
  X() { a=1; }
  int test();
  friend int foo();

  private:
  int a;
};

int X::test() {
  return foo();
}

int foo() { return 0; }
```

Digraphs

The C++ standard now defines the following keywords: and, bitor, or, xor, compl, bitand, and_eq, or_eq, xor_eq, not and not_eq. If you try to define these as symbols, it will cause a compile time error. See Chapter 15 of *z/OS C/C++ Compiler and Run-Time Migration Guide,* GC09-4913 for more details.

4.2.2 ObjectModel (IBM)

In the z/OS V1R2 C++ compiler, a new object model is introduced to z/OS. The object files produced will perform better than those produced using the previous object model, especially when the inheritance tree is deep. However, the size of the object files will be larger.

Note that C++ classes compiled with the new object model cannot inherit from classes compiled with the older object model. Therefore, all libraries shipped with the new compiler are compiled using the `objectmodel(compat)` flag, which allows inheritance from classes compiled with the old object model.

4.2.3 Commonly used z/OS C++ options

In addition to the options described in 4.1.4, “Commonly used z/OS C/C++ options” on page 59, following is a short list of useful C++ options:

ATTRIBUTE A compile phase option, it produces an attribute listing for the identifiers used in your program. To avoid excessive information, a variable has to be used in the code before it would show up in this listing (unless the `full` suboption is used). Example:

```
$ c++ -Wc,”attribute” hello.C
```

DLL This is a bind phase option. Unlike the C counterpart, the compile phase DLL option is not needed in C++. The C++ compiler always produces DLL-enabled .o files (objects). For example:

```
$ c++ -Wl,DLL -o mydll mydll.C
```

TEMPINC This a compile phase option. The compiler instantiates templates using template instantiation files. By default, these files are created under the directory ./tempinc. You can change this using this option. For example:

```
$ c++ -Wc,”tempinc(./temp)” hello.C
```

4.3 Dynamic Link Libraries (DLLs)

The function of shared libraries on most UNIX platforms is provided on z/OS by dynamic link libraries. A DLL is a collection of one or more functions or variables in an executable module that is executable or accessible from a separate application module. In an application without DLLs, all external function and variable references are resolved statically at bind time. In a DLL application, external function and variable references are resolved dynamically at run-time.

Here’s a simple example of how to build and use a DLL. Suppose we have a function foo that we want to be in a DLL; we have a source file called mylib.c:

```
$ cat mylib.c
#include <math.h> /* need fabs */
double precision=1.0E-6;

double foo(double g) {
   double x = 1.0, y;
   if (g<0.0) return -1.0;/*error condition */

   while ((y=g/x, fabs(y-x) > precision))
      x = (x+y)/2.0;
   return x;
}
```

To build the DLL:

```
$ c89 -omylib -Wc,DLL,EXPORTALL -Wl,DLL mylib.c
```

The `-o` option names the DLL mylib. We need the DLL option in both the compile phase and the bind phase. The EXPORTALL option in the compile phase exports all external names defined in the source. Two files are produced: mylib, the DLL; and mylib.x, the sidedeck. The sidedeck is a text file. Mylib is the actual DLL used when you are running the program.

The sidedeck contains the names of all the exported variables and functions. If you use the **`cat`** command on the sidedeck, it produces the following output:

```
$ cat mylib.x
IMPORT CODE,mylib,'foo'
IMPORT DATA,mylib,'precision'
```

The exported names are `foo` and `precision`. The record length of this file is important. We recommend that you do not edit the file; but if you do edit it, you must keep the record length unchanged.

The sidedeck is used in the bind step of the program that uses the DLL. For example:

```
$ cat prog.c
extern double precision; /* imported from the DLL */
double foo(double); /* imported from the DLL */
void main() {
    printf(“sqrt 2.0 = %e +- %e\n”, foo(2.0), precision);
}
```

Compile prog.c as follows:

```
$ c89 -oprog -Wc,DLL prog.c mylib.x
```

You need the compile phase DLL option, but you do not need the bind phase DLL option. The latter is needed only when building the DLL itself. The sidedeck, mylib.x, is used to resolve the names exported from the DLL.

Note that the DLL user (prog.c, in our case) does not need to explicitly identify the imported variables and functions. The sidedeck mechanism will resolve them the same way as a static library.

When running the program, the system searches for DLLs in the path names specified in the environment variable LIBPATH. Make sure you add the path names of your own DLLs to this variable. For example:

```
$ export LIBPATH=./:$LIBPATH
$ prog
```

Finally, instead of using the EXPORTALL option, you can also use the #pragma export to export selected functions and/or variables. C++ also supports the _Export keyword. Refer to *OS/390 C/C++ Language Reference*, SC09-2360 for details.

4.3.1 Differences between C and C++

The preceding example shows a C DLL. You do not need to use the DLL option for C++ in the *compile phase*. The option is always turned on.

But you still need the DLL option in the bind phase. For example, the following command compiles a C++ DLL:

```
$ c++ -omylib -Wc,exportall -Wl,dll -+ mylib.c
```

All the options are the same as before in C, except that the compile phase DLL option, `-Wc,dll`, is not needed.

4.3.2 Mixing DLL and non-DLL

You can mix DLLs and non-DLLs. A C program compiled with `-Wc,dll` or a C++ program compiled with any options can use both DLLs and static libraries. On the other hand, a C program compiled with `-Wc,nodll` can only be statically linked.

If your library has both DLL and non-DLL versions, the step that determines which one to use is in the bind step of the program using the library. If the sidedeck (the .x file) is used in the bind, the DLL version is used. If the .o or the archive library is used, the non-DLL version is used.

There is one caveat, though, regarding the use of function pointers: function pointers in programs compiled with `-Wc,dll` cannot point to a non-DLL function, unless the `DLL(CALLBACKANY)` suboption is used. Even though the DLL compile option is always turned on for C++, the suboption `CALLBACKANY` is not. This suboption generates less efficient code; turn it on only if you need to use a DLL function pointer on non-DLL functions.

4.3.3 DLLRNAME utility

You can use the DLLRNAME utility to package and redistribute DLLs with your z/OS C/C++ applications.

With the DLLRNAME utility, you can modify an executable application or a DLL load module to change the names of any DLLs that are loaded at execution time. The DLLRNAME utility also provides a report which can be used to understand the DLL dependencies of your application.

For more details about using DLLs, refer to the chapter entitled “Building and Using Dynamic Link Libraries (DLLs)” in *OS/390 C/C++ Programming Guide,* SC09-2362.

4.4 C/C++ Compiler Interprocedural analysis (IPA)

Interprocedural analysis (IPA) is a compiler feature which extends the scope of code optimization from a single function or source file to an entire loadable program segment such as an executable module or dynamic link library. IPA plays two important roles in improving code optimization: it widens the scope of analysis for existing powerful optimizations in the compiler, and it performs many new analyses and transformations that would be impossible in a traditional optimizer.

The analyses and transformations performed by IPA include:

- Inlining across compilation units (source files)
- Context-sensitive program partitioning (mapping of code)
- Context-sensitive mapping of static data
- Elimination of unreachable or fully-inlined functions
- Interprocedural propagation of constants and sets
- Interprocedural context-sensitive pointer usage analysis
- Interprocedural analysis of function side effects
- Interprocedural elimination of redundant or unused stores and computations
- Intraprocedural computation propagation and simplification
- Function specialization or cloning

For more details of this tool, see the chapter “Optimizing Your C/C++ Code with Interprocedural Analysis” in *OS/390 C/C++ Programming Guide*, SC09-2362.

4.4.1 How IPA works

Traditional compilers perform optimization within a single source file, or more precisely a compilation unit (CU). However, more optimization opportunities are available if the compiler can draw on information from more than one compilation unit.

Interprocedural Analysis does just that—it pulls together information from all CUs and optimizes globally. The following simple example illustrates cross-inlining:

```
$ cat file1.c
int foo(int, int);
int main() {
   if (foo(1,3) > 2) {
      /* some error condition */
   }
   else {
      /* do something */
   }
}
```

```
$ cat file2.c
int foo(int x, int y) {
   return (x-y)*(x-y);
}
```

IPA sees information from both file1.c and file2.c, and determines that it can *inline* the function call foo into the conditional. The conditional is then transformed to:

```
if ((1-3)*(1-3) > 2)
```

Since the expression is now a constant, it can be evaluated during compile time; the conditional is always true. The false path of the if statement is removed from the program.

4.4.2 How to invoke IPA

A normal compilation consists of a compile phase and a bind phase. The compile phase produces .o files, and the bind phase links all the .o files together to resolve external references to produce the executable. The IPA phase lies somewhere in between and overlaps part of the compile phase and part of the bind phase. The compile phase deposits analysis information into the .o files when individual source files are processed. (This is just like a normal compile.)

Then an IPA link step is invoked to *ipa-link* all the .o files together; the interprocedural optimization is done at this time. The output is a temporary file. A normal bind phase is still needed to produce the final executable.

Refer to the sections entitled “Compiling with IPA” and “IPA Link Step” in *OS/390 C/C++ User's Guide*, SC09-2361 for a detailed description of the IPA compilation process. The full IPA Link functionality is powerful and flexible and can handle a different mixture of objects and libraries while maintaining binary backward compatibility. However, you do not have to use all the capabilities all at once. The following gives an example to illustrate the basics.

Let us use the sample program presented above, with the two source files file1.c and file2.c. A slight modification is made to file1.c to form a complete program so that we can compare the run time improvement after IPA optimization.

```
$ cat file1.c
int i=0;
int loop = 100 * 1000*1000; /* 100M */

int foo(int, int);
int main() {
   int j;

   for (j=0; j<loop; j++)
      if (foo(1,3) > 2) {
          i++;
      }
      else {
         printf(“dead code...\n”);
      }
}
```

Note that normal single file optimization cannot hoist the function call (in the conditional) out of the loop because of possible side effects of the function call.

An IPA compilation can be done simply as follows:

```
$ c89 -o prog -WI file1.c file2.c
```

The I (upper case I) following -W instructs the compiler to use IPA. The compile phase, IPA, and bind phase are invoked automatically to produce the executable prog.

Alternatively, you can do the IPA compile step and IPA link step separately as you would in a make-based build. For example, the following commands compile file1.c and file2.c separately to produce file1.o and file2.o.

```
$ c89 -c -WI file1.c
$ c89 -c -WI file2.c
```

This is similar to a normal compilation except analysis information is inserted into the .o files for use by the IPA Link step. The IPA Link step is invoked when the -WI option is used in the bind phase. For example, the following command invokes the IPA Link step to ipa-link file1.o and file2.o together.

```
$ c89 -o prog -WI file1.o file2.o
```

Interprocedural analysis is done at this point. After that, the binder is invoked automatically to produce the executable prog.

You can try out the same program without IPA to compare the run time:

```
$ c89 file1.c file2.c
```

and run the a.out.

4.4.3 Optimization levels in IPA

Started in OS/390 V2R10, IPA optimization comes in three levels (LEVEL 0, 1 and 2). IPA LEVEL(2) is the most aggressive IPA optimization available. It provides the fastest execution time improvement at the expense of application build time. However, your particular application may or may not benefit from using it because optimization is highly dependent on the type of computations done by your application. We recommend testing running your application at IPA LEVEL(0) (which only does inter-CU optimization, mostly used for debugging purposes), then progress to IPA LEVEL(1), and finally IPA LEVEL(2).

Each source file is considered as a CU. During normal optimization, each function by itself is being optimized. The benefit of using IPA is that optimization is done over function boundaries, giving the compiler more opportunity to optimize. At IPA LEVEL(0), this intra-function optimization is done at a per-CU level. At LEVEL(1) and (2), optimization is carried out in all functions within the program, which means there are more ways the compiler can optimize.

4.5 Extra Performance Linkage (XPLink)

The traditional z/OS linkage was not designed to handle a large number of function calls with a small function body; the overhead of function calling is relatively expensive. XPLink was designed to bring performance improvement opportunities to applications that are call-intensive with a small function body, typical of the newer workloads written in C and C++ being ported to z/OS. XPLink achieves this by changing register conventions and the layout of the stack. These changes allow for:

1. Faster detection of stack overflow
2. Faster saving of registers
3. Faster allocation of local storage for called functions
4. Opportunities for improved register allocation in function bodies

5. Greater use of registers for function arguments and return values, improving the performance of argument list construction, parameter use, return value construction, and return value use

Measurements of many specific code fragments changed by XPLink show a greater than 30% performance improvement. From a UNIX shell, XPLink applications can be created by using the `-W` option to specify additional options to the compile and link-edit phases. For example, to compile a simple C program as an XPLink application, you could use:

```
$ c89 -o HelloWorld -Wc,xplink -Wl,xplink HelloWorld.c
```

The IBM redbook *XPLink: OS/390 Extra Performance Linkage*, SG24-5991covers XPLink in detail. On the Web at:

```
http://www.redbooks.ibm.com/abstracts/sg245991.html
```

4.6 Language feature issues

This section describes language feature issues that you are likely to encounter during a porting exercise. In 4.6.6, “Key differences between cc and c89” on page 74, we elaborate on the differences between **c89** and **cc**, and in 4.6.7, “Comparison between OS/390 V2R10 C++ and z/OS V1R2 C++ compilers” on page 77, we detail the differences between ISO C++ 1998 and OS/390 C++; these will help you to plan your port.

Note that a port can be divided into three steps (assuming you are doing a direct port of an existing application running on another platform). The first step is to get a clean build. The second is to make it run. After it runs successfully, the third step is to do performance tuning. The following will help you go through the first step and part of the second step.

4.6.1 Options

Refer to Appendix B, “Comparison of z/OS and GNU compilers and make tools” on page 159 for a comparison between gcc options and z/OS options.

When building the application for the first time on z/OS, we recommend setting the options as follows:

langlvl(extended) If the application needs the long long data type, turn on this option. For example:

```
$ c89 -Wc,"langlvl(extended)"
```

However, the __STDC__ macro is not defined if this option is turned on. (Note: If the application requires long long and also depends on this macro, the application is not ANSI-conforming.)

-g Try not to use this option, as it makes the objects and load modules large—which may cause problems with the build environment in terms of disk space. It could also take a longer time to compile. During the initial stage, you may be spending time setting up make files and the source tree; long builds consuming large amount of storage makes this stage difficult. *Turn this on only after you get a clean build and start debugging the port.*

-O Do not turn on any optimization options, including IPA. Get a clean build and successful run first.

ANSIALIAS *Do not* turn this option on; that is, use NOANSIALIAS. Turn this on afterwards in the performance tuning phase of the porting exercise. Refer to 7.1.3, “ANSIALIAS compiler option” on page 95 for an explanation of this option. Example:

$ c89 -Wc,NOANSIALAS

4.6.2 Enum size

On most other UNIX platforms, the size of enum is constant and is usually the size of int. On z/OS, the size of enum depends on the range of the enumerators (enum constants). The compiler uses the size of the smallest integral type for the enum—the size could be 1, 2 or 4 bytes.

This usually does not present a problem unless the enum is used inside a structure. To force the enum to be of size int, insert an enumeration constant with value INT_MAX.

An option, ENUM, is provided in V2R9 via a PTF (and available starting in OS/390 V2R10) to control the enum size. However, this option must be used with care as it affects all enums including those defined in system headers. Enums defined in system headers must remain the same size, otherwise the behavior is undefined. Refer to the *OS/390 C/C++ Compiler User Guide*, Chapter 5, “Compiler Options.”

4.6.3 Extern “C”

The parameter passing convention for function calls (i.e., the linkage convention) is different between C and C++ on z/OS. If you are mixing C and C++ source code, make sure you use the extern “C” construct to specify C functions. For example:

```
extern “C” {
   void foo(int);
}
```

or

```
extern “C” void foo(int);
```

or

```
void __cdecl foo(int);
```

For function pointers, the syntax is a bit tricky. Keep in mind that a qualifier always qualifies the ‘*’ before it. For example:

```
void (* __cdecl F)(int);
```

Read the declaration starting from the name:

```
F is a __cdecl pointer
   to a function
      with an int parameter
         returning void.
```

You can use F to point to the function foo defined earlier:

```
   F=&foo;
```

Hints

You can write function pointer declarations using the following steps:

1. Ignore the pointer first. Just write down the function prototype; for example, void* foo (int, int*) is a function taking two parameters, int and int*, and returns a void*. In this case, foo is just a dummy name.

2. Put parentheses around the function name; for example, void* (foo) (int, int*).
3. Insert the asterisk (*) before the name and inside the parentheses. Change the name to the required pointer variable name if necessary; for example, void* (*F)(int, int*) declares F as the function pointer variable.
4. Also, if you add typedef at the beginning, you get a typedef; for example, typedef void* (*F)(int, int*). If you remove the name altogether, you get the type definition which can be used in type casts; for example, (void* (*)(int, int*)) is the cast operator.
5. Note that qualifiers apply to the '*' before it. It is true not just for __decl but for other qualifiers, like const, as well; for example, if it is a constant pointer, insert const after the asterisk * to get: void* (* const F)(int, int*).

You cannot use the extern "C" syntax on a member function or a static function. You must use the __cdecl syntax. For example:

```
struct X {
   void __cdecl member(int);
};

static __cdecl init(void);
```

A common difficulty here is with the use of function pointers. If you are passing function pointers across a C/C++ boundary, i.e., taking the address of a function on one side and then using it (call the function) on the other side, the function has to be in C linkage. This is because the C linkage is the only one that is understood by both sides.

This may cause a major problem in porting when the application uses a lot of function callbacks, as in the case of X applications where the X library expects function pointers to be in C linkage.

A related issue is the use of C signal handlers in C++. The handler has to be in C linkage. Be careful if the signal handling mechanism is wrapped inside a class.

Virtual member functions cannot be in C linkage. Therefore, they cannot be passed as callback into a C library.

4.6.4 Mixing signal handling and exceptions

You cannot throw an exception from within the signal handler, and you cannot mix setjmp/longjmp with exception handling. The C++ exception handling interferes with the other two mechanisms on the stack.

When an object is thrown, the runtime has to unwind the stack to get to the proper enclosing scope that catches the exception. All the local objects in the nested blocks are destructed in an orderly manner during the unwinding. The runtime puts marker on the stack so that it can backtrack itself.

Note that setjmp/longjmp also uses the stack to keep track of the jump. This is essentially a C mechanism and does not know anything about C++ exceptions, stack unwinding, and object destructions. It is an undefined behavior to throw an exception within a signal handler or to do a longjmp from within the execution of a try block.

4.6.5 Bit fields

Bit fields are platform-specific. You almost always run into a problem if the structures have bit fields and the size of the structures has to remain the same across all the platforms (for example, if the structure is a packet header in a communication protocol). Refer to Appendix G, “Bit field” on page 183 for some porting hints.

4.6.6 Key differences between cc and c89

On z/OS, **cc** supports support K&R language features, while **c89** conforms to Standard C (ANSI89 C). There are significant differences between the two. Note that K&R was the de facto standard on the UNIX environment before the C language was standardized in 1989. C compilers based on K&R may vary slightly, and programs written using K&R C may not be as portable as those using Standard C.

Note: On z/OS, **c89** implicitly turns on the compiler options LANGLVL(ANSI) and ANSIALIAS. **cc** implicitly turns on LANGLVL(COMMONC) and NOANSIALIAS.

The following highlights the major differences.

Mixing signed and unsigned types in an expression

This is probably the most serious difference between **cc** and **c89**. It may cause a totally different program behavior. At issue here is the resulting type of an expression when there are mixed signed and unsigned types.

(You can run the following code to check which way your existing compiler is behaving and choose the language level via the LANGLVL option accordingly.)

```
#include <stdio.h>
int main()
{
   int  i = -8;
   unsigned char j = 3;
   float g = i+j;

   /* Notice the different value of g in c89 and in cc. */
   printf(“g=%f\n”, g);

   /*
      The key is in the type of (i+j).
      If the value preserving rule is used, the type
      is signed int and the value is -5.
      If the unsignedness preserving rule is used, the
      the is unsigned int and the value is UINT_MAX - 5.
   */

   if ( g < 9)
      printf("i am in ansi mode\n");
   else
      printf("I AM IN CC MODE\n");
}
```

What is the type of `(i+j)`? **cc** uses *unsigned preserving*, meaning that in an expression with mixed signed and unsigned types, everything is converted to unsigned, and the result is unsigned. **c89** uses *value preserving*; roughly speaking, everything is converted to int if it can contain the value, or unsigned int otherwise (ANSI 3.2.1.5. arithmetic conversion.)

Since this is not an error as seen from either mode, there is no message for this.

In **c89**, you can force unsigned preserving by using the option `UPCONV`. This option has no effect on **cc**.

Macro substitution within quotes

cc does macro parameter substitution inside quotes. **c89** does not. There is no compiler message in either mode.

```
    #define  msg(parm)  "parm"
    #define  quoteMe(a) 'a'
    {
       printf("msg=%s\n",  msg(this_is_a_message));
       printf("quoteMe = %c\n", quoteMe(x));
    }
```

Compiling with **c89**, the above gives:

```
 msg = parm
 quoteMe = a
```

Compiling with **cc**:

```
 msg = this_is_a_message
 quoteMe = x
```

A way to get the result of **cc** in ANSI is:

```
 #define msg(parm) #parm
 #define quoteMe(a) (#a)[0]
    ...
```

There is no effective way of doing **quoteMe** under ANSI. The method above uses brute force; the macro expands into an object of type char. It may not be exactly the same as the corresponding **cc** version in all circumstances.

Macro substitution with concatenation /**/ vs

There could be a syntax message in **c89** depending on how the tokens are formed.

```
 #define cat(a,b)   a/**/b
 {
    int i1;
    printf("i1=%d\n",  cat(i,1));
   }
```

Under **cc**, `cat(i,1)` is replaced by `i1`. Under **c89**, all comments are replaced with a single space; so `cat(i,1)` becomes two tokens, `i 1`, with a space in between. Note that the above *will not compile* under **c89**.

ANSI uses ## for the above purpose:

```
 #define cat(a,b)  a##b
```

Casting on the left-hand side of an assignment

This is *not* allowed under **cc** or **c89**; but is allowed in some old C compilers (for example, gcc allows it).

There is an error message from **c89**:

```
 {
    char *p;
    float *f;
```

```
      (char *) f = p;
     }
```

cc allows using the size of on bit fields

ANSI does not allow it. There is a compiler message from **c89**:

```
   struct X {
      int b:1;
     } x;

     {

      printf("sizeof b = %d\n",  sizeof(x.b));
     }
```

Note that the size and alignment of bit fields are implementation-defined; the value you get may be different on different platforms. So even though **cc** allows it, you might still want to change the code to make it portable.

cc tolerates non-int bit fields

ANSI does not allow it. There is a compiler message from **c89**:

```
   struct X {
           char b:1;
   } x;
```

cc gives a warning and treats the bit field as unsigned int. **c89** gives an error. (Note that in this case, “error” means no object code is generated.)

cc allows anonymous struct and union

ANSI does not allow it. There is a compiler message from **c89**:

```
       struct X {
          int i;
          struct Y {
          float f;
          }; /* no variable name for this struct */
     } x;

       foo() {
          float g;
          g = x.f; /* ok for cc, error for c89 */
       }
```

Assigning an int to a pointer

```
   {
     float *v;
     int i;

     v = i;
    }
```

cc gives a warning (object code still generated). ANSI gives an error.

Using **c89** with LANGLVL(EXTENDED) also gives a warning. But note that there is a very high risk of violating the ANSIALIAS rules. Either use the NOANSIALIAS option, or be sure that the address really refers to the same object type as the pointer type when the pointer is dereferenced. **cc** also tolerates a `float` assignment to a pointer.

Function prototypes

K&R does not have function prototypes. ANSI introduces this into the language as a step towards type safety. However, since there are still many old programs that uses K&R-style function definition, ANSI accepts it; but this should be avoided in new programs.

Let's look at function definition first.

K&R:

```
int foo(a, b)
     float a;
     char b;
{
   /* do something */
}
```

ANSI

```
int foo(double a, int b)
{
   /* do something */
}
```

Note the types of the parameters under ANSI; they are double and int instead of float and char. K&R promotes the argument before passing into the function; char, short and int are promoted to int; float, double are promoted to double. ANSI passes the arguments as specified. So if you want to change a K&R style definition to an ANSI definition, remember the default promotions to keep the effects on the caller to a minimum.

The corresponding function declarations are as follows:

K&R:

```
int foo();
```

ANSI

```
int foo(double a, int b);
```

Because the K&R declaration does not specify parameter types, no type checking is possible by the compiler. A program can potentially call the same function from two different locations and pass different argument types.

You can use the ANSI prototype with the K&R definition. This allows you to add a common header file that contains the prototypes of all the functions in an application to take advantage of the better type checking, but still leave the function definitions alone without change. But keep in mind the default argument promotion rule when creating the prototypes. In fact, only promoted types are allowed in mixing K&R and ANSI function declarations/definitions.

4.6.7 Comparison between OS/390 V2R10 C++ and z/OS V1R2 C++ compilers

OS/390 (and z/OS V1R1) C++ follows the C++ standard draft proposal. The proposal had been evolving since that draft and there were a number of changes when the language was standardized in 1998. The following highlights the major differences between the OS/390 V2R10 C++ (which is the same in z/OS V1R1) and z/OS V1R2 C++ (ISO 98 C++ Standard compliance). At z/OS V1R1, the C++ compiler is the same as the one for OS/390 V2R10.

Attention: All of the differences between OS/390 C++ and ISO C++ 1998 have been addressed. In z/OS V1R2, the C++ compiler is fully ANSI compliant and the following differences no longer apply. However, if your code used to compile free of errors with the OS/390 C++ compiler, it may fail due to the stricter rules in the IOS C++ 1998 Standard.

Scope of variable in for-loop initializer

In ISO C++ 1998, the scope of the variable declared in the for-loop initializer declaration is to the end of the loop body. The scope of such variables in C++ for z/OS V1R1 and earlier is to the end of the lexical block containing the for-loop. For example:

Example 4-4 `forloop.C`

```
#include <iostream.h>

main()
 {
  for (int i=0;i<10;i++) {
    if (i==9)
      cout << "i is in scope" << endl;
   }
  if (i==10) // On z/OS V1R2, error: CCN5274 - i not in scope
   cout << "i is in scope" << endl;
 }
```

You may run into problems if there is more than one variable i visible at the point after the for-loop.

Friend class name

In ISO C++ 1998, the class name in a friend declaration is not visible until introduced into scope by another declaration. For example:

```
class C {
   friend class D;
};
D *p; // Accepted in OS/390; error in ISO C++ 1998.
// error message when compiled with z/OS V1R2
// CCN5040 (S) The text "*" is unexpected.  "D" may be undeclared or ambiguous.
```

This change is due to the stricter rule in the ISO C++ 98 Standard.

Exception handling

When an object is thrown in ISO C++ 1998, a temporary copy rather than the actual object itself is used.

Also, in ISO C++ 1998 the catch of const objects will catch both const and non-const throws. This is not true in OS/390 C++.

Namespaces

Namespaces are not yet supported in OS/390 C++ as of V2R10, but are fully supported in z/OS V1R2 C++.

When using the OS/390 C++ compiler that does not support namespaces (all versions of the OS/390 C++ and z/OS V1R1 C++ compilers), if your code requires the std namespace, try defining it as a macro to nothing:

```
#define std
```

```
#define using
#define namespace
```

The bool type

The bool type is not yet supported in OS/390 C++ as of V2R10, but is fully supported in z/OS V1R2 C++.

When using the OS/390 C++ compiler, you can define a class bool to work around this. However, additional code changes may be needed if the program expects the relational operators to return a bool type. For example:

```
class bool;
void foo(bool);
void foo(int); // overloaded
...
foo (a<b); // expect to call foo(bool), but called foo(int) instead
```

Mutable specifier

The mutable specifier is not yet supported in OS/390 C++ as of V2R10, but is fully supported in z/OS V1R2 C++.

When using the OS/390 C++ compiler, this keyword nullifies the const specifier in a class object. If your code uses this keyword, try defining it as a macro to nothing:

```
#define mutable
```

wchar_t

As of OS/390 V2R10, wchar_t is defined as a typedef. ISO C++ 1998 defines it as simple type.

Attention: In z/OS V1R2, wchar_t is defined as simple type.

explicit

The explicit keyword is not yet supported in OS/390 C++ as of V2R10, but is fully supported in z/OS V1R2 C++.

The purpose of this keyword is to make a conversion constructor a normal constructor. For example:

```
class C {
   explicit C(int);
};
C c(1); // ok
C d = 1; // error, no conversion constructor
```

Operator new and delete

In ISO C++ 1998, operator new throws std::bad_alloc if memory allocation fails. As of V2R10, OS390/C++ does not throw this exception.

As of V2R10, OS390/C++ does not support overloaded delete.

The above features are now fully supported in z/OS V1R2 C++.

C++ cast

In ISO C++ 1998, there are new cast operators: const_cast, dynamic_cast, reinterpret_cast and static_cast. These are not supported in OS/390 C++ as of V2R10.

Implicit int not valid

ISO C++ 1998 requires that declaration specifiers must have a least one simple type, and functions must have a return type. The following examples are invalid under ISO C++ 1998:

```
const i; // OS/390 C++ means const int i.
main() { } // OS/390C++ means returning int.
```

This should not be a problem for porting from ISO C++ 1998 to OS/390 C++ or z/OS V1R1 C++ compilers. The opposite direction of porting would require code changes to explicitly specifying the int such that declaration specifiers must have at least one simple type and functions must have a return type.

Templates

Templates have changed significantly in ISO C++ 1998. Following are the areas that have changed

Type names

This is to resolve the ambiguity whether a name is a class name (to be instantiated later) or a variable name.

Template keywords

The use of keywords to indicate templates in qualifiers. For example:

```
struct A {
  Template<class T> T f(T t) { return t;}
};
template<class T> class C {
  void g(T* a) {
  // The following would become ambiguous without
  // the keyword template
  int i = a->template f<int>(1);
  // this would become ambiguous
};
C<A> c;
```

Name searching

In ISO C++ 1998, template-dependent base classes are not searched during name resolution. For example:

```
int *t=0;
template <class T> struct Base {
   T t;
};
template<class T> class C : public Base<T> {
   T f() {
   return t; // refers to global int *t
   }
};
```

Template specialization

In ISO C++ 1998, template specializations must be preceded with the string `template`<>. For example:

```
template<class T> class C {};
template <> C<int> { int i; };
```

Partial specialization can be done as follows:

```
template<class T, class U> class C {};
template<class A> class C<int, A> {};
```

```
C<int, char> x; // uses partial specialization
C<char, int> y; // uses template
```

Explicit call to destructor of scalar type

This problem is not template-specific, but usually occurs in templates. For example:

```
typedef int INT;
INT *p;
// ...
p->INT::~INT(); // ok in ISO C++ 1998
```

OS/390 C++ gives a warning to the explicit destructor call. This warning can be ignored.

4.7 Language Environment (LE) issues

Compliance with the ISO ‘98 C++ standard resulted in some changes to LE.

4.7.1 Changes to string and wide character function prototypes

To comply with the ISO ‘98 C++ standard, certain string and wide character function prototypes have changed. The header files string.h and wchar.h changed. As a result, some code may no longer compile. If you want to use the old style functions, you can specify the compiler option LANGLVL(OLDSTR).

For example, consider the sample program:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
const char *string1 = "needle in a haystack";
const char *string2 = "haystack";
char *result;

result = strstr(string1,string2);
/* Result = a pointer to "haystack" */
printf("%s\n", result);
}
```

The pre-ISO '98 C++ prototype for `strstr()` was:

```
char * strstr (const char *, const char *);
```

and for the ISO '98 C++ support, this was changed to include two separate prototypes:

```
const char * strstr (const char *, const char *);
char * strstr (char *, const char *);
```

Because neither of these two prototypes exactly matches the original, older pre-ISO '98 C++ code will receive the compiler message:

```
CCN5216 (S) An expression of type "const char *" cannot be converted to type "char *".
```

To eliminate this message without updating the C++ source code, add the compiler option LANGLVL(OLDSTR) to expose the old prototype. Since only C++ supports overloaded function names, the original single prototype continues to exist for C applications.

In a similar manner, the signatures of several other standard C library routines are replaced in the C++ standard. The affected routines are:

- `strchr()`
- `strpbrk()`
- `strrchr()`
- `strstr()`
- `memchr()`
- `wcschr()`
- `wcspbrk()`
- `wcsrchr()`
- `wcsstr()`
- `wmemchr()`

4.7.2 Standard C++ library header files

There are several new header files that are shipped with the z/OS Language Environment to support the ISO '98 C++ functions (also see 5.4, "Using C functions in C++ applications" on page 85). These headers exist in both the HFS and the z/OS Language Environment data sets. Because these headers are named without the typical .h suffix, they are shipped in a new <prefix>.CEE.SCEEH data set, instead of the existing <prefix>.CEE.SCEEH.H data set. This is done to avoid conflicts due to duplicate base names (e.g., <new> and <new.h>). For more details, refer to the manual *z/OS Language Environment Customization*, SA22-7564.

4.7.3 Run Time Type identification (RTTI)

RTTI information must be compiled for operator typeid and dynamic_cast to work. RTTI support is shipped with LE but is highly integrated with the compiler.

4.8 Commonly used environment variables

In the following, <xxx> stands for one of C89, CC or CXX. It specifies the utility you want to receive the environment setting.

`_<xxx>_CCMODE`	Set to 1 to allow mixing options and file names on the command line.
`_<xxx>_STEPS`	Set to -1 to use the prelinker.
`_CXX_CXXSUFFIX`	Set to the suffix you want to be identified as C++ source files.
`LIBPATH`	The DLL search path. Add the path names of any DLLs used by the application.
`_BPX_SHAREAS`	Set to YES to spawn child processes in the same address space as the parent.
`_CEE_RUNOPTS`	Set to XPLINK(ON) in order for a non XPLINKed application to call an XPLINKed DLL (for example, STL is built as XPLINKed DLL).

5

Standard C++ Library

The Standard C++ Library consists of two big parts, the Standard Template Library (STL), and the rest of the library.

The STL is part of the ISO C++ 1998 Standard. It provides a set of containers and algorithms to manipulate basic types and user-defined types (classes). The STL is very versatile and flexible; it can promote code reuse and improve program correctness if applied appropriately. The full functionality of STL requires many of the features in ISO C++ 1998, for example namespaces and the latest template support.

Before z/OS V1R2, the C++ compiler did not provide a set of STLs. To satisfy customer requirements, a free downloadable STL ported by SGI referred to as STLPort was used. See Appendix D, “STLPort” on page 167.

The STL supplied with the C++ Compiler is more compilant with the ISO C++98 standard. We will refer to it as ANSI STL in this document.

The rest of the Standard C++ Library contains facilities such as the iostream library, an earlier version of which was shipped with the IBM Open Class. The new iostream library is again more standard compliant and feature rich.

5.1 Difference between ANSI STL and IBM Open Class (IOC) and SGI STLport

5.1.1 Iostream library

The ANSI STL contains a new ISO C++98 compliant iostream library. There are a few things to watch out for when using i compared to the older IBM Open Class (IOC) iostream library.

The header files for the ANSI STL do not have the .h extension, for example:

```
#include <iostream>
```

The new iostream’s symbols are contained in the std namespace. One can either use the fully qualified name, e.g. std::cout, or the directive at the begining of your source file,

```
using namespace std;
```

before any call to the iostream library.

Mixing and matching the new and old iostream libraries is not recommended; the results are unpredicable.

5.1.2 Floating point numbers

The z/OS ANSI STL has been extended to support both the IEEE floating point number format as well as the S/390 native HEX format. The support is transparent to users and is selected by using the FLOAT() compiler option. However, you should not mix and match compilation units built with different FLOAT options.

The purpose of this support is to enable new applications to take advantage of the newest C++ library features while still being able to access data in HEX floating point format. The option for IEEE is FLOAT(IEEE) and, similiarly, FLOAT(HEX) for HEX.

5.1.3 Character set: ASCII vs. EBCDIC

For those porting from an ASCII platform, this STL also includes seamless support for ASCII and EBCDIC. This is controlled by the compiler options ASCII and NOASCII. The ASCII and NOASCII compilation units should not be mixed in one program. This ASCII option was designed to simplify porting applications that need to communicate with ASCII-based Applications (for example, a Web server).

5.1.4 Large file support

Access to hierarchical file system (HFS) files that are over 2 GB in size is available only with the ANSI STL iostream library. This is known as large file support, which is not available in the older IOC iostream library.

5.2 Debugging applications using the STL

There are some considerations when debugging problems that arise using the ANSI STL.

The STL has been extended to support the features unique to z/OS, but as discussed before, some of these are mutually exclusive. If your application has unresolved symbols, make sure the symbols that are missing do not fall into the namespace of a mutually exclusive feature.

For example, if an application is built with the FLOAT(HEX) and ASCII options and the following unresolved symbol is encountered:

```
traps__Q3_3std9_FLT_IEEE15_Num_float_base
```

the demangled version of the symbol is:

```
std::_FLT_IEEE::_Num_float_base::traps
```

The `_FLT_IEEE` namespace in the symbol name should raise suspicions that an object file was compiled with the wrong FLOAT() flag.

Tip: How to demangle symbol names in z/OS UNIX

Create an empty file in the ~/bin direcotry named CXXFILT, using the **`touch`** command, then turn on the sticky bits. This instructs the shell to search the STEPLIB and LPA for the command. Make sure ~/bin is in PATH.

```
#touch CXXFILT
#chmod +t CXXFILT
```

Now to demangle the symbol name, do this:

```
#CXXFILT
```

and enter the mangled name.

5.3 Deciding between ANSI STL and IOC libraries

We highly recommend that you use the ANSI STL for all new application development. And only use the IOC libraries for backward compatibility reasons. All feature improvements will only be done in the ANSI STL, as it is the direction of the C++ Standard.

5.4 Using C functions in C++ applications

Due to changes in the C++ Standard, all standard library functions are in the std namespace. When developing C++ applications and the use of C functions arises, use the C++ version of header files. For example, stdio.h is now imported into the std namespace in cstdio. Similarly for the other C header files, the C++ header file counter part is in the form of c”header filename” but without the .h extension.

5.5 Difference between climits and limits

Header climits is for using the C style limit functions to obtain the system limit information. Header limits contains the limits information using the ISO C++ 98 Standard numeric_limit class to access limit information. The floating-point limit information returned by the numeric_limit class depends on the FLOAT() compile time option.

Example 5-1 Getting max int using limits.h

```
#include <stdio.h>
#include <limits.h>

int main() {
   printf("%d\n",INT_MAX);
   return 0;

}
```

Example 5-2 Getting max int using limits

```
#include <stdio.h>
#include <limits>

int main() {
        printf("%d\n", std::numeric_limits<int>::max);
        return 0;
}
```

6

Adaptive Communication Environment (ACE)

The Adaptive Communication Environment (ACE) is a set of object-oriented frameworks, C++ wrappers, and C APIs that perform many of the mundane tasks required by applications. It is especially geared toward real-time applications that require interprocess communication and synchronization. ACE is distributed in source for many different platforms, including z/OS UNIX.

ACE was developed by Douglas Schmidt and the Center for Distributed Object Computing (DOC) of Washington University, St. Louis, Missouri, USA. ACE continues to be developed by Dr. Schmidt, now at DARPA, and DOC personnel. Since ACE is open source, many other people in academia and in industry contribute to ACE's development.

Since ACE is being developed by its user community on different platforms, the newer beta releases are not tested on all platforms. And since z/OS UNIX is not a common platform in the ACE world, more recent distributions may contain incompatibilities with z/OS UNIX.

See the following URL for ACE copyright and license information:

    http://www.cs.wustl.edu/~schmidt/ACE-copying.html

6.1 ACE components

ACE can be invoked at different levels or layers. The application designers must balance the need for shielding the application from operating system dependencies and low-level complexity, with the need for controlling the processing. Figure 6-1 on page 90 illustrates the ACE layers.

6.1.1 C API layer

The C API layer, also called the OS adaptation layer, provides platform-independent C-callable functions, such as:

Concurrency and synchronization
APIs for multi-threading, multi-processing, and synchronization.

Interprocess communication (IPC) and shared memory
APIs for local and remote IPC and shared memory.

Event demultiplexing mechanisms
APIs for synchronous and asynchronous demultiplexing I/O-based, timer-based, signal-based, and synchronization-based events.

Explicit dynamic linking
APIs for explicit dynamic linking, which allows application services to be configured at installation time or run time.

File system mechanisms
APIs for manipulating files and directories.

6.1.2 C++ wrappers layer

The following components constitute the C++ wrappers layer:

Concurrency and synchronization
ACE abstracts native OS multi-threading and multi-processing mechanisms, for example mutexes and semaphores, to create higher-level Object-Oriented concurrency abstractions like Active Objects and Polymorphic Futures.

IPC and file system APIs
The ACE C++ wrappers encapsulate local and/or remote IPC mechanisms, such as sockets, TLI, UNIX FIFOs and STREAM pipes, and Win32 Named Pipes. In addition, the ACE C++ wrappers encapsulate the OS file system APIs.

Memory management
The ACE memory management components provide a flexible and extensible abstraction for managing dynamic allocation and deallocation of interprocess shared memory and intraprocess heap memory.

6.1.3 Frameworks layer

The following components constitute the frameworks layer:

Event demultiplexing
The ACE Reactor and Proactor are extensible, object-oriented demultiplexers that dispatch application-specific handlers in response to various types of I/O-based, timer-based, signal-based, and synchronization-based events.

Service initialization
The ACE Acceptor and Connector components decouple the active and passive initialization roles, respectively, from application-specific tasks that communication services perform once initialization is complete.

Service configuration
The ACE Service Configurator supports the configuration of applications whose services may be assembled dynamically at installation-time or at run-time.

Hierarchically-layered stream
The ACE Streams components simplify the development of communication software applications, such as user-level protocol stacks, that are composed of hierarchically-layered services.

ORB adapters
ACE can be integrated seamlessly with single-threaded and multi-threaded CORBA implementations via its ORB adapters.

6.1.4 Distributed services layer

ACE provides a standard library of distributed services that are packaged as self-contained components. These service components are not strictly part of the ACE framework library. They provide reusable implementations of common distributed application tasks such as naming, event routing, logging, time synchronization, and network locking.

They also demonstrate how ACE components like Reactors, Service Configurators, Acceptors and Connectors, Active Objects, and IPC wrappers can be used effectively to develop flexible, efficient, and reliable communication software.

6.1.5 Higher-level middleware components

The following higher-level middleware components are also available with ACE:

The ACE Orb (TAO)
TAO is an open source, standards-compliant implementation of CORBA optimized for high-performance and real-time systems. It is designed to provide efficient, predictable service to applications. TAO contains the network interface, operating system, communication protocol, and CORBA middleware components and features

Java Adaptive Web Server (JAWS)
JAWS is a high-performance, adaptive Web server built using the framework components and patterns provided by ACE.

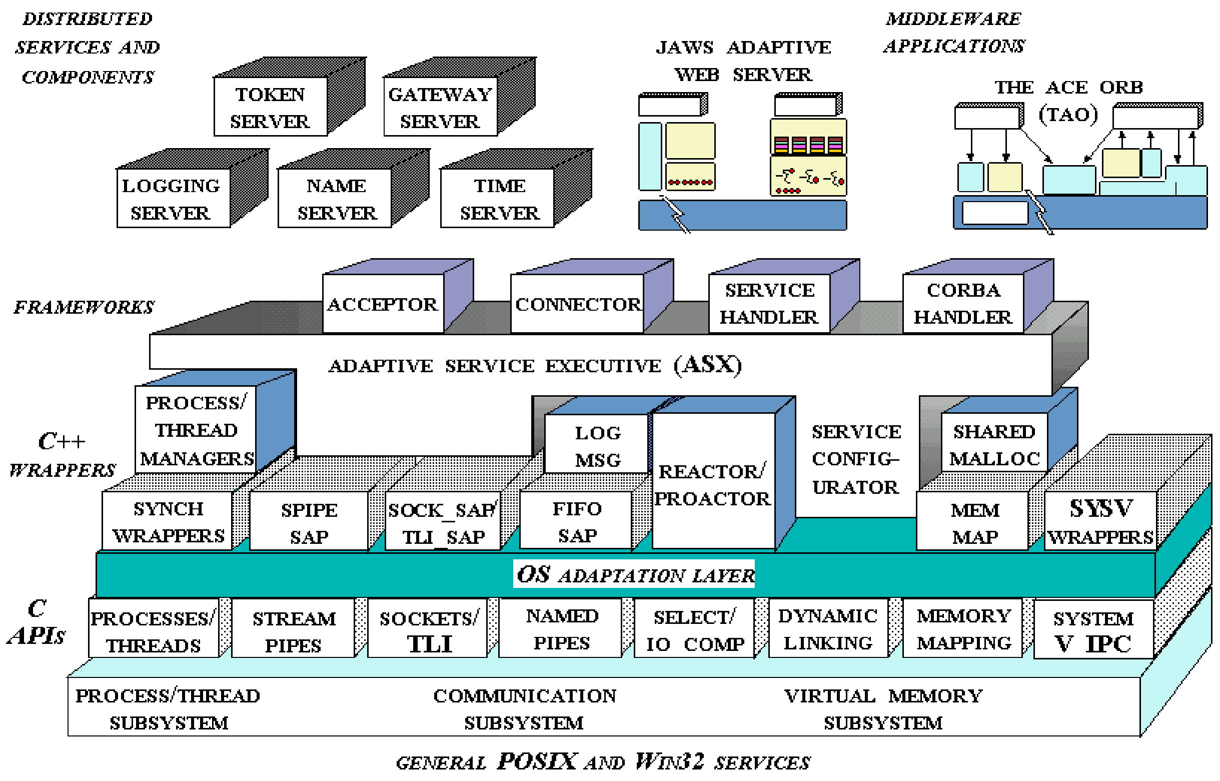


Figure 6-1 Overview of ACE

6.2 Installing ACE on OS/390

The following steps were taken to install ACE on OS/390.

6.2.1 Obtaining ACE

The ACE source code was downloaded from:

```
http://deuce.doc.wustl.edu/Download.html
```

It is in tar-gzip format and any text files in the tar file will be in ASCII. Unwind the tar file with the following command:

```
$ gzip -d --stdout ACE-5.1.tar.gz | pax -o from=ISO8859-1 -r
```

6.2.2 Building ACE

Building ACE is fairly straightforward. See the ACE-INSTALL file in the distributed source. It is also on the Internet at:

```
http://www.cs.wustl.edu/~schmidt/ACE-install.html
```

The ACE-INSTALL file gives instructions in general and for various platforms in particular. The main points are worth emphasizing, since we seem to forget one or another of them each time we install a new version of ACE.

1. Set _CXX_CCMODE

ACE makefiles create compile and link commands with options and operands intermixed. By default, C and C++ on OS/390 UNIX do not allow this. Override the default by adding the following line to .profile or in your working environment (for bash, sh and similar shells)

```
$ export _CXX_CCMODE=1
```

2. Set $ACE_ROOT

   ACE makefiles depend on an environment variable, ACE_ROOT, to find where ACE is installed on your system. Set ACE_ROOT to point to the ACE_wrappers directory where you installed ACE. For example, add the following line to .profile:

   ```
      $ export ACE_ROOT=/xxxxxxxx/ACE_wrappers
   ```

   where `xxxxxxxx` is the path where you unpacked ACE.

3. Copy config.h

   In the $ACE_ROOT/ace directory, copy the file config-mvs.h to config.h, or create a symbolic link. The config-mvs.h header contains macros relevant to OS/390.

4. Copy platform_macros.GNU

   In the $ACE_ROOT/include/makeinclude directory, copy the file platform_mvs.GNU to platform_macros.GNU, or create a symbolic link. This file contains makefile directives relevant to OS/390.

5. Make ACE

   To build all of ACE, including network services, tests, examples, Web services, etc, run the GNU make command from the ACE_wrappers directory. To just build enough of ACE to be callable from other applications, run GNU make from the ACE_wrappers/ace directory.

   Note: You must use GNU make. The native OS/390 make does not work with ACE. Refer to 3.4, “Make tools” on page 44, for instructions on obtaining GNU make.

6.3 ACE resources

The ACE Web pages contain a great deal of useful information, such as descriptions, tutorials, technical papers, install instructions and much more.

You can reach the ACE Web pages at:

```
http://ace.cs.wustl.edu/
```

6.3.1 Riverace

Riverace Corporation (www.riverace.com) supports ACE commercially under an “open source business model”, meaning that they respect the terms of the ACE copyright agreement. They offer pre-built versions of ACE and various support plans. Currently, Riverace offers support plans for ACE on AIX, HP-UX, Solaris, Linux, Windows NT and Windows 2000.

6.3.2 OCI

Object Computing Inc (OCI) supports TAO commercially under a similar open source license arrangement. OCI offers support on several different platforms including Linux, Solaris, HP UX and Windows-NT and Windows 98; refer to the following Web site:

```
http://www.theaceorb.com/
```

6.3.3 List servers

There is a great deal of discussion and collaboration in the ACE user community; check the following sources.

ACE Announcements listserv

Used by the developers to announce new releases of ACE, you can subscribe at:

```
ace-announce-request@cs.wustl.edu
```

Messages are also forwarded to the general discussion list.

General Discussion listserv

For discussing usage, enhancements, and porting issues regarding ACE. Subscribe at ace-users-request@cs.wustl.edu. The list is archived at:

```
http://www.egroups.com/messages/ace-users/
```

Bug Report listserv

Used for submitting bug reports to the ACE developers, you can subscribe at:

```
ace-bugs-request@cs.wustl.edu
```

The list is archived at:

```
http://www.egroups.com/messages/ace-bugs/
```

A bug report form is distributed with the source files and on the Internet at:

```
http://www.cs.wustl.edu/~schmidt/ACE_wrappers/PROBLEM-REPORT-FORM
```

Usenet discussion group

A usenet discussion group, *comp.soft-sys.ACE*, it is also available at:

```
http://www.dejanews.com
```

Type in the discussion group name and you will find announcements, bug reports and discussion messages posted on the listservers that are also copied here.

7

Performance and tuning techniques

There are many items to consider for optimal performance. The redbook *Tuning Large C/C++ Applications on OS/390 UNIX System Services*, SG24-5606 proved to be an excellent reference, and we used it to determine which performance and tuning techniques should be investigated. We summarize the items here.

Besides the OPTIMIZE and IPA options, following are some of the additional build time options (compile, IPA or bind) that can be used to tune up the runtime and compile time performance. Note that not all options are applicable to a particular application; try them and select the ones that apply. Refer to the *OS/390 C/C++ User's Guide*, SC09-2361 and to *Tuning Large C/C++ Applications on OS/390 UNIX System Services*, SG24-5606 for detailed information about options.

- Compile option -Wc,ANSIALIAS
- Compile option -Wc,ARCH(3)
- Compile option -Wc,COMPRESS
- Compile option -Wc,NOCVFT
- Compile option -Wc,FASTTEMPINC
- Binder option -Wl,EDIT=NO
- Use “built-in functions”
- Use the options RPTOPTS(ON) and RPTSTG(ON) to generate an LE information report. Then use the information to fine-tune heap usage by using the following runtime options:
- LE run-time option ALL31(ON)
- LE run-time options ANYHEAP and BELOWHEAP
- LE run-time options HEAP and HEAPPOOLS
- Use the OS/390 C/C++ performance analyzer:

  http://www.ibm.com/software/ad/c390/pt/pa1.htm

7.1 Environment variables related to spawn

Ensure the _BPX_SHAREAS and _BPX_SPAWN_SCRIPT environment variables are both set to YES:

```
$ env | grep BPX
_BPX_SPAWN_SCRIPT=YES
_BPX_SHAREAS=YES
```

The _BPX_SHAREAS environment variable controls where child processes are spawned. If YES, then spawned processes usually run in the same address space as the parent process (except when the program spawned has sticky bit on or is an external link, or is a SETUID or SETGID program or has exhausted its private storage). If this variable is not set or is not set to the value YES, the shell creates all processes in separate address spaces.

No matter how the shell is started, you must set _BPX_SHAREAS=YES if processes started by the shell itself are to run in processes nested in the shell's address space.

Some processes cannot execute correctly in a shared address space. For example, if a process needs to reserve MVS system resources that are common to all processes in an MVS address space, it must run by itself.

If two processes using the same MVS resource attempted to execute concurrently in the same address space, they would compete for these resources, thus causing at least one of them to fail. When a potential storage shortage is detected, the new processes are created in their own address spaces, even if _BPX_SHAREAS=YES is present in the invoker's environment.

Note: For performance reasons, you do not want `cc`/`c89`/`cxx` to use a shared address space (local spawn). The extended file attributes let you mark a file as ineligible for a local spawn. The `cc`, `c89`, and `cxx` utilities should not be locally spawned, so use the `extattr` command with the -s option to turn off this extended attribute for these utilities. For example:

```
 $ extattr -s /bin/c89 /bin/cc /bin/cxx /bin/c++
```

The _BPX_SPAWN_SCRIPT environment variable controls how some executables are invoked. The following explanation is from *OS/390 UNIX System Services User's Guide,* SC28-1890, section 4.6 “Improving the Performance of Shell Scripts”:

> To improve the performance of shell scripts, set the _BPX_SPAWN_SCRIPT environment variable to a value of YES.
>
> If _BPX_SPAWN_SCRIPT=YES is not already placed in /etc/profile, you can put it in your $HOME/.profile.
>
> Here is what the variable does: if the spawn callable service determines that a file is not an HFS executable or a REXX exec, this setting causes spawn to run the file as a shell script directly. In the default processing, however, if the spawn callable service determines that a file is not an HFS executable or a REXX exec, the spawn fails with ENOEXEC and the shell then forks another process to run the input shell script. Setting this variable to YES eliminates the additional overhead of the fork.
>
> You may want to set the variable to NO when you are running a non-shell application. For example, if an application does not support shell script invocations, set the variable to NO. Likewise, if an application is in test mode and the ENOEXEC returned would be a useful indication of an error in the format of the target executable file, set the variable to NO.

Also, if you invoke OS/390 UNIX from ISPF or TSO with the OMVS command, avoid setting the environment variable STEPLIB. Alternatively you can set it to none, as shown:

```
$ echo $STEPLIB
none
```

This applies only to OMVS sessions. It has no effect for telnet or rlogin sessions.

The following three options apply in most situations.

7.1.1 EDIT=NO bind option

The bind option EDIT=NO tells the binder that the load module or program object output will not be used in a rebind later on. This saves the binder from generating additional information into the load module, thereby reducing the load module or program object size. The restriction, of course, is that the load module cannot be used as input to the binder for reprocessing. Also, you cannot use STROBE.

7.1.2 Prelinker

Try using the prelinker to prelink all the objects into one object first before invoking the binder. The prelinker was used in the older releases to process long external names (longer than 8 characters); the linker used in the older releases could not handler external names longer than 8 characters. This processing is now incorporated into the binder and the prelinker is no longer needed.

But prelinking the objects still has a desirable side effect of reducing the final load module size. To use the prelinker, set the environment variables _xxx_STEPS to -1, as shown in the following example:

```
$ export _C89_STEPS=-1
$ export _CC_STEPS=-1
$ export _CXX_STEPS=-1
```

Then run `c89`, `cc` or `c++` as usual to bind the program.

The Prelinker cannot handle objects in the Generalize Object File Format (GOFF). Thus, objects compiled with XPLINK cannot use the prelinker.

7.1.3 ANSIALIAS compiler option

When you are first porting an application to OS/390, it is advisable to turn off the ANSIALIAS option (i.e. use NOANSIALIAS). Turn this option on only if you are sure the source code is *strictly* conforming to ANSI rules in terms of aliasing.

In a nutshell, the ANSI aliasing rule says that a pointer can only be dereferenced to objects of the same type; pointers to char are exceptions. Following is a typical example:

```
int foo() {
   float g = 1.0f;
   int *p = (int*) &g;
   printf("g (in hex) = %x\n", *p); /* p is actually pointing to
                                       a float; referencing the
                                       float as an int voliates
                                       ANSI aliasing. */
}
```

The NOANSIALIAS option tells the compiler to be careful about code like this and to avoid certain aggressive optimizations; refer to the following example:

```
int *p;
int *faa(double *g); /* casting double to int */
```

```
void foo(double f) {
   p = faa(&f);
   /* p is pointing to f, but the optimizer */
   /* does not know about it.*/
   f += 1.0;
   printf("f=%x\n", *p);
}

int *faa(double *g) { return (int*) g; }

void main() {
   foo(0.0);
}
```

In the preceding example, inside function foo, faa is essentially a cast operator that casts a `double` pointer to an `int` pointer. Later on in the `printf`, p is used to reference f. But the optimizer expects *p to be an `int`; this assumption is basing on the ANSI aliasing rule. So the optimizer thinks that the result of `f += 1.0` is never really used; it might remove this statement altogether. The output from the print could be zero.

We used the function faa here just to illustrate that faa could be in a separate source file and to make it easy to create the error. The cast could be in the current source file or somewhere else. As long as you dereference a pointer to a different type, the error may potentially show up.

7.2 Tuning OS/390 UNIX for compile-intensive systems

Tuning is critical for improving the performance of a system that is primarily used for C/C++ application development. There are two types of tuning that can be done:

System tuning
In system tuning, your systems programmer tunes the system for compile-intensive workloads. OS/390 UNIX System Services has a list of release-specific tuning targets at the following URL:

`http://www.s390.ibm.com/oe/perform/bpxpftgt.html`

Programmer tuning
In programmer tuning, you can select options and code structures to improve the compile time of your application.

7.2.1 System tuning: frequently used modules in LPA

On systems where application development is the primary activity, performance may benefit if you put the following modules into the link pack area (LPA):

- Frequently used Language Environment run-time library routines should be put into the LPA. Specifically, do the following:
  - Include the CEE.SCEELPA data set in the LPA list.
  - Include the following members of the CEE.SCEERUN data set in the LPA list using the Dynamic LPA capability (SET PROG=):
    - CEEBINIT
    - CEEBLIBM
    - CEEEV003

    - EDCRNLIB
    - EDCRNLST
    - EDCZV
    - ECCZ24
  - Include the CEE.SCEERUN data set in the link list ((LNKLSTxx parmlib).

- The z/OS C/C++ compiler V1R2 and later should be put into the LPA by including CCN.SCCNCMP in the LPA list. The z/OS C/C++ compiler V1R1 and earlier should be put into the LPA by including CBC.SCBCCMP in the LPA list.
- The linkage editor or binder should be put into the modified link pack area (MLPA) by making these changes to the IEALPAxx parmlib member:

```
INCLUDE LIBRARY(SYS1.LINKLIB)
     MODULES(
             IEFIB600
             IEFXB603
             IEWBLINK
             HEWL
             HEWLDRGO
             HEWLH096
             HEWLOAD
             HEWLOADR
             IEWBLDGO
             IEWBLOAD
             IEWBLODI
             IEWBODEF
             IEWL
             IEWLDRGO
             IEWLOAD
             IEWLOADI
             IEWLOADR
             LINKEDIT
             LOADER
            )
```

- The **c89**/**cc**/**cxx** commands should be put into LPA by following these steps:

  a. Make sure that you have not set the external attribute (that is, you have *not* issued the command `extattr -s /bin/c89 /bin/cc /bin/cxx /bin/c++`).

  b. Use this JCL to link the **c89** command into a PDS (you can do the same for the **cc** and **cxx** commands):

```
//PUTINLPA  JOB  MSGLEVEL=(1,1)
//*                                                           *
//* INLMOD DD STATEMENT SPECIFIES THE DIRECTORY THAT CONTAINS  *
//* THE PROGRAM.                                              *
//*                                                           *
//* THE INCLUDE STATEMENT SPECIFIES THE NAME OF THE FILE TO    *
//* RUN FROM THE LPA.                                         *
//*                                                           *
//* THE NAME STATEMENT SPECIFIES THE FILE NAME BUT IN          *
//* UPPERCASE. THIS MUST BE SAME AS THE FILE NAME.            *
//*                                                           *
//LINK     EXEC PGM=IEWL,REGION=4M,
//  PARM='LIST,XREF,LET,RENT,REUS,AMODE=31,RMODE=ANY,CASE=MIXED'
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(10,10))
//SYSPRINT DD SYSOUT=*
//INLMOD   DD PATH='/bin/'
//SYSLMOD  DD DSN=OECMD.LPALIB,DISP=SHR
//SYSLIN   DD *
```

```
INCLUDE INLMOD(c89)
ALIAS CC
ALIAS CXX
ENTRY   CEESTART
NAME      C89(R)
```

c. The data set containing the **c89**/**cc**/**cxx** modules, OECMD.LPALIB, should be put into the LPA by including it in the LPA list.

d. Then turn on the sticky bit for these files in the HFS, using these shell commands:

```
chmod +t /bin/c89
chmod +t /bin/cc
chmod +t /bin/cxx
```

e. Make c++ a symbolic link to **cxx** (because c++ is not a valid LPA module name), using these commands:

```
mv /bin/c++ /bin/c++2
ln -s /bin/cxx /bin/c++
```

7.2.2 System tuning - I/O tips

The following I/O tips may improve performance of compile-intensive systems:

- Ensure that control unit caching with DASD fast write is enabled.
- To avoid I/O contention, give each user a separate mountable file system. This lets you spread user file systems across multiple DASD devices.
- Tune your jobs to avoid channel and pack contention. This can occur when the compiler is not in LPA and resides on the same pack as the header files in /usr/include/ and/or /usr/lpp/ioclib/include/ and multiple compiles are executing concurrently.
- You can use the **filecache** command to store the most frequently used system header files in an HFS file system.
- Exploit the DFSMS/MVS V1R5 performance enhancements that allow limits to be set for virtual storage and the sync daemon interval. For more information, refer to the redbook *Hierarchical File System Usage Guide*.
- You can define /tmp as a RAM disk by specifying:

  ```
  FILESYSTYPE TYPE(TFS) ENTRYPOINT(BPXTFS)
  ```

  This is described in more detail in *OS/390 UNIX System Services Planning*.

7.2.3 Programmer tuning

The following programming tips may improve performance of your C/C++ compiles:

- The higher the optimization level used to compile your application to generate better performing code, the higher the compile time will be. Where run-time performance is not an issue, use OPT(0).
- You can add code to the beginning and end of a header file to ensure that it is not processed unnecessarily during compilation. The following example contains code that is included in a header file called myheader.

  ```
  ??=ifndef __myheader
  ??=ifdef __COMPILER_VER__
  ??=pragma filetag ("IBM-1047")
  ??=endif
  #define __myheader 1
  .
  .
  ```

```
. /* header file contents */
??=endif
```

You must ensure that the filetag statement, if used, appears before the first statement or directive except for any conditional compilation directives. The ifndef statement is the first non-comment statement in the header file (the actual token used after the ifndef statement is your choice). The define statement must follow; it cannot appear before the filetag statement, but it must appear before any other preprocessor statement (other than comments).

Note that the header can contain comment statements.

A brief detour

filetag and **trigraph**

`filetag` is an OS/390-specific pragma telling the compiler the codepage of the current source file. IBM-1047 is the compiler default; you don't have to use this pragma if the entire application use the same code page.

Note that a `filetag` specified in a source file extends its effect to all the included header files unless the header has its own `filetag`. That is why system headers use the pragma `filetag` to explicitly specify codepage IBM-1047; this way the header can be safely included from any source files that use other character encodings. The effect of `filetag` ends at the end of the source file specifying the pragma.

A related syntax is the use of *trigraphs*. Since source files can be coded in code pages other than IBM-1047, we have to be careful in using characters that are encoded differently in different code pages. (These characters are called *variant* characters. The opposite are *invariant* characters. All alpha-numerics are invariant, as are characters in trigraphs.)

The # character is a variant character. That is why the equivalent trigraph, ??=, is used in the preceding example. You don't have to use trigraphs after the pragma filetag because the compiler now knows the codepage of the source file and can process the variant characters correctly.

- Use the system header files from HFS instead of partitioned data sets to improve compile time. Specify the following compiler options to do this:

  ```
  For C++: NOSEARCH SEARCH('/usr/include/', '/usr/lpp/ioclib/include/')
  For C: NOSEARCH SEARCH('/usr/include/')
  ```

- The SEARCH and LSEARCH sub-options should be listed from the headers in the most frequently located directory to the headers in the least frequently located directory.
- With the MEMORY compiler option (the default), the compiler uses a memory file in place of a work file, if possible. This option increases compilation speed, but you may require additional memory to use it.

  If the compilation fails because of a storage error, increase your storage size or recompile your program using the NOMEMORY option.

- If your application has many recursive templates, the FASTTEMPINC compiler option may improve the compilation time. This option defers generating object code until the final versions of all template definitions have been determined. Then, a single compilation pass generates the final object code; no time is wasted generating object code that will be discarded and generated again.

  If your application has very few recursive template definitions, NOFASTTEMPINC may be faster than FASTTEMPINC.

- If a C++ source file does not have try/catch blocks or throws objects, the NOEXH C++ compiler option may improve the compilation time. However, the resultant code will not be ANSI-compliant if the program uses exception handling.
- Try to remove functions from your source code that are not needed (that is, there are no calls to them). Removing these functions should reduce the compile time, in addition to reducing the size of the module.

  On OS/390, the binder does not prune unreferenced functions--so if they are not required, they should be removed from the source (preferably by commenting them out with a #ifdef - #endif block).

  (Note that if you build your application with IPA, unreferenced functions *are* pruned.)
- To improve your OPT compile time at the expense of run-time performance, you can specify:
  - MAXMEM, which limits the amount of memory used for local tables of specific, memory intensive optimizations. If this amount of memory is insufficient for a particular optimization, the compiler performs somewhat poorer optimization and issues a warning message.

    Reducing the MAXMEM value from 2 G to 10 M may disable some optimizations, which in turn may cause some decrease in execution performance.
  - NOINLINE disables inlining, and may decrease the compile time with a decrease in execution performance.
- You may also be able to *marginally* improve your build time by using options to reduce the size of the generated code. Specifically:
  - Use built-in functions and macros by including the appropriate system header files (this will also provide a run-time performance improvement).
  - Use the compiler option COMPRESS to suppress the generation of function names in the function control block.
  - Use the C++ compiler option NOCVFT to reduce the size of the writable static area (WSA) for constructors that call virtual functions within the class hierarchy where virtual inheritance is used.
  - Use the compiler options ROSTRING and ROCONST to remove string literals and const variables from the WSA.
  - Use the compiler option NOTEST and NOGONUMBER to generate programs that do not contain any information required for debugging with a debugger, or for tracing with the Performance Analyzer.
  - If you are using DLLs, do not use the compiler option EXPORTALL. Instead, use the #pragma export directive or the C++ _Export keyword.
  - If you do not need the newer capabilities of the binder, you can use the prelinker in your build process to get smaller executables.
  - If you do not need rebind your executables in your build process, use the EDIT=NO binder option.
- Use of the following binder options can increase the bind time:
  - MAP and XREF can add as much as 20% to the bind time.
  - ALIASES(ALL) can add 20% to the bind time.

8

net.TABLES Application

This chapter describes the net.TABLES application that was ported to OS/390 UNIX.

8.1 Introduction

net.TABLES is a TCP/IP-enabled, high performance, multi-platform table management facility for distributed, heterogeneous networks. It is designed to enable the rapid development of applications that require processing of many table lookups. net.TABLES manages tables in memory and allows multiple users across disparate platforms to read, modify and permanently store data concurrently.

Based upon years of experience with our IBM mainframe product tableBASE, Data Kinetics Ltd. developed net.TABLES to target system architects and application developers in the financial services, retail sales, and telecommunications industries. For full details see:

`http://tables.dkl.com/`

8.2 Components

net.TABLES currently consists of the five components described in the following sections.

8.2.1 net.TABLES server

The server is a multi-threaded engine that manages table data in memory. It supports client/server over TCP/IP, as well as optimized local access though shared memory. Several table organizations are supported. Rows may be fixed or variable length.

8.2.2 C-API client

The C programming language Application Programming Interface (C-API) is a powerful, thread-safe function set. The communications component of the C-API automatically detects platform differences so that communications are optimized no matter where the client is running with respect to the server. The C-API allows simultaneous multiple connections to multiple net.TABLES servers.

8.2.3 net.LOGGER server

The logger facility accepts error messages, informational messages, and trace messages from net.TABLES servers. All critical errors are automatically logged. Users can turn trace logging on dynamically on their connection, or on a per-call basis. System administrators and support personnel can trace remote connections to help debug and analyze client applications.

8.2.4 net.DRIVER

net.DRIVER is a platform-independent command line utility built on the C-API that enables you to execute net.TABLES functions without writing any code. It allows you to quickly create, modify, and manage tables and perform system administration functions. A powerful feature is net.DRIVER's recursive script, record, and playback mechanism.

8.2.5 net.EDITOR

net.EDITOR is a Windows NT Graphical User Interface (GUI) application built on the C-API to create, modify, and manage tables. Several tables can be viewed at the same time and data can be viewed in several formats: ASCII, EBCDIC, or hexadecimal.

Figure 8-1 shows how net.TABLES components fit together.

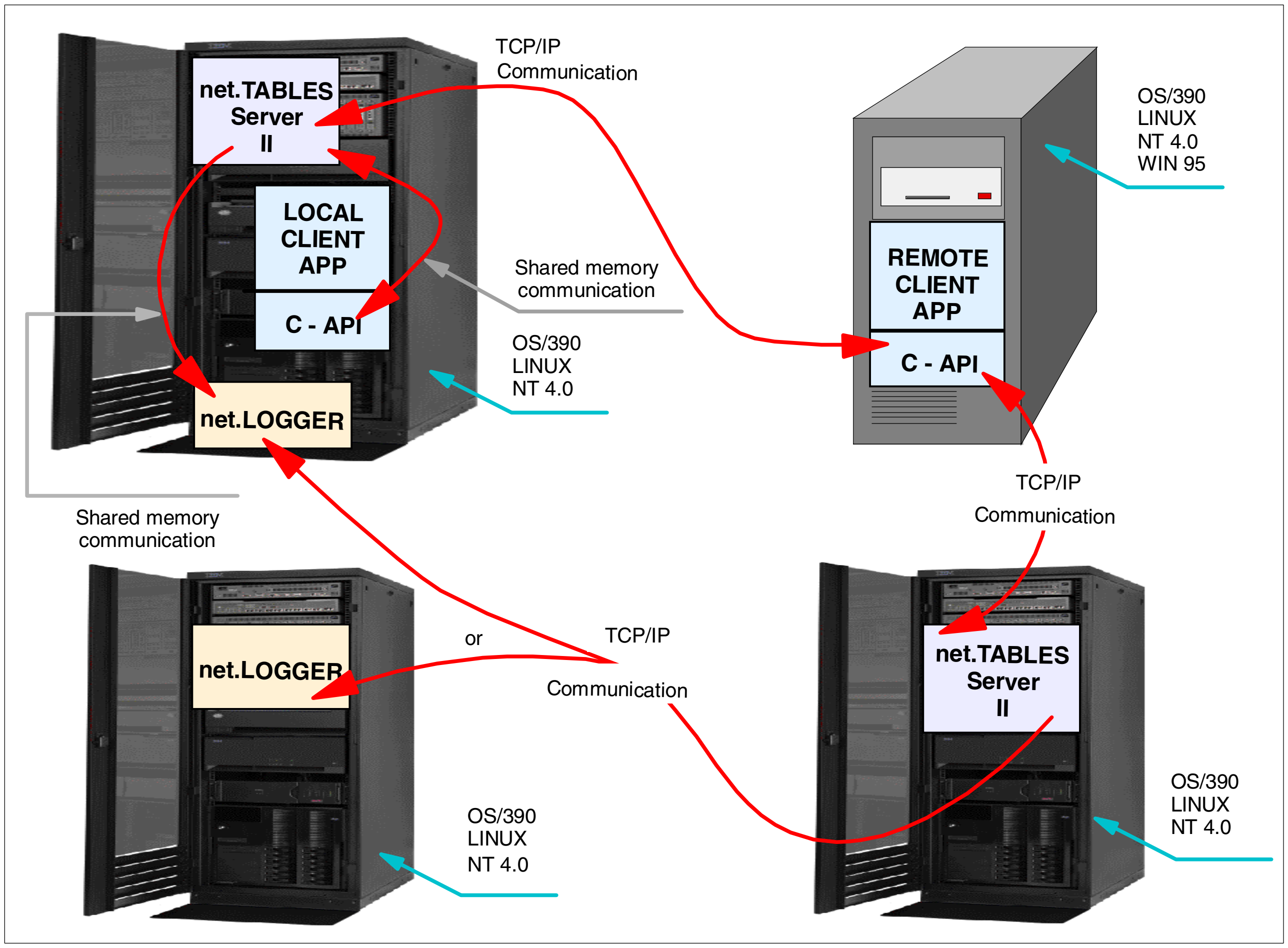


Table 8-1 net.TABLES components

8.3 How net.TABLES was developed

net.TABLES was developed from the beginning with the understanding that it would be ported to several platforms. We wanted to develop using a single code base for all platforms. Our strategy was to develop the net.TABLES server simultaneously on Intel-based Linux and NT, and the C-API simultaneously on Linux, NT, and Win95. Because of our mainframe history, we planned an early port to OS/390 UNIX from the outset.

We wanted to quickly show proof of concept and have interoperability with a minimum of risk developing on Linux and NT. We took considerable care to write portable code and to implement solutions that we were confident would apply to many platforms. Then we would port to OS/390 and USS as a true test of portability and tie in with our tableBASE mainframe product. Factions arose but we managed to maintain a single code base with a minimum of `#ifdef`s.

Early in the project we decided to use the Adaptive Communication Environment (ACE). This greatly contributed to the portability of our code and allowed for rapid development. Although ACE is a vast and complex environment, its implementations are very efficient. See Chapter 6, “Adaptive Communication Environment (ACE)” on page 87 for a description of ACE.

One feature of ACE is that it is updated regularly (almost weekly) and constantly improving. This has its advantages in that there is a very short turnaround between reporting a problem and receiving a fix (usually submitted by yourself). A disadvantage is that we spent considerable time updating and testing the releases of ACE, which detracted from our development time.

Because of this, we decided to freeze on an official release with most of the features we needed. The disadvantage of staying behind the latest release of ACE is not having all the latest fixes. For example, we found a problem with a cast in an `sprintf` call, invented a quick workaround, and then found that the same problem had been solved by someone else and the fix was already in the latest beta version of ACE.

In addition to staying on a major release for some time, we pruned ACE down to include only the files needed by net.TABLES. As explained in Chapter 6, “Adaptive Communication Environment (ACE)” on page 87, ACE is composed of several different layers. For net.TABLES, we kept most of ACE at the C++ wrappers layer and below.

This had several benefits. It allowed us to concentrate on our requirements, reduce the overhead in complexity and compile times, and ignore the irrelevant ACE tests that did not succeed. We kept the Acceptor and Connector from the higher layers and discarded everything else. We recommend that you examine ACE and only use the portion that applies to your needs.

For net.TABLES, we wanted to have a good deal of control so we invoked ACE at the C++ wrapper level. This also helped reduce the size of the net.TABLES executables.

To avoid future surprises, we attempted a preliminary port to OS/390 version 2.4 early in the project, just to see where we stood with portability issues. We quickly discovered that using a service bureau with an older version of OS/390 over a slow connection was unfeasible. We did not have TCP/IP between our desktop PCs and OS/390, so file transfer was especially painful; we used IND$FILE to transfer files between the desktop and MVS, and the TSO OGET and OPUT commands to transfer between MVS and USS.

We tried again later at a different site using OS/390 version 2.7. This was significantly more fruitful. Our first discovery was that compiling ACE requires a significant amount of CPU and elapsed time. We strongly recommend that you tune your environment. Pruning ACE to only the needed components will probably help more than any tuning of the environment. For more information, refer to *Tuning Large C/C++ Applications On OS/390 UNIX Systems Services*, SG24-5606, and the OS/390 Performance Web pages at:

    http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1tun.html

8.4 Portability observations

Software developers are always keen to use the latest features and enhancements available. However, leading edge features are not always conducive to highly portable code. Despite our better judgment, we could not resist experimenting with some of the more common C++ features. Here we discuss some observations.

8.4.1 Standard Template Library

The Standard Template Library (STL) is popular on Intel-based Linux and Win32 platforms. STL is a good prototyping tool for small, contained projects but has several drawbacks for a production product. Some issues we perceived are:

- Implementations differ among platforms.
- Debugging templates is very difficult.
- The classes are overladen with functionality.
- More efficient implementations are easily developed.

More critical drawbacks, which compel us to not recommend STL in any application, are:

- Memory management is not under your control.
- Exceptions are compiled into the STL library.
- STL uses `new` and `delete`, which can compromise error trapping and is less efficient.

With STL, if you need to handle out-of-memory conditions in a controlled manner, then your only mechanism is through exception handling. This is a cumbersome and inefficient mechanism for a production server application.

STLPort

The C++ Standard Template Library (STL) is not yet supported on OS/390, but a freeware STL can be downloaded and built on USS; refer to Chapter 5, “Standard C++ Library” on page 83.

A major concern for multi-threaded applications is thread safety. Although STLPort is thread-safe, this feature has not been tested on OS/390 and is not supported.

8.4.2 #pragma

Preprocess #pragma directives, by their nature, are specific to individual platforms. Therefore, we suggest avoiding using them unless absolutely necessary. In our opinion they only serve to complicate the reading of code and can often mislead the programmer while debugging.

8.4.3 Exceptions

Exceptions are great, in theory. In practice, however, they cause more frustration than they’re worth. For some GUI development, they free programmers from thinking about unforeseen errors and allow them to concentrate on the application at hand. This way of thinking has become more prevalent in recent times as users of client applications become accustomed to their applications aborting on unhandled exceptions. The inevitable solution is a monolithic exception handler that must be aware of every finite state of the application.

We did not use C++ exceptions for these other reasons:

- Exceptions are not supported on all platforms.
- They will make the code bigger (by as much as 25%, on some platforms).
- They hinder the performance of the software (a critical point for net.TABLES) by as much as 15%, on some platforms.

8.4.4 Threads

Threads and synchronization mechanisms (mutexes, condition variables, etc.) can be handled differently on OS/390 than on Linux or Windows NT (Win32). Linux threads are fully POSIX-compliant, while OS/390 threads comply with draft 6 of the POSIX standard (Win32 threads are wildly different from threads available on other operating systems; they are not POSIX).

We use the ACE library to handle threads independently of the underlying operating system. Without the availability of the wrappers from ACE (or any other similar library), the task of writing a portable, multi-threaded engine would become a lot more complex.

8.5 ACE considerations

As discussed in 8.3, "How net.TABLES was developed" on page 103, the ACE toolkit is an integral part of net.TABLES. We explain here some considerations that led us to adapt ACE to use it more efficiently for our product. These observations pertain to ACE Version 5.1, released March 15, 2000.

8.5.1 ACE overloads main()

ACE overloads `main`() to ensure that the singletons are initialized correctly, that the memory allocator and the message logger are created as early as possible, and so on.

However, we have our own memory allocator and we use a different message logging strategy, so we decided to bypass the standard behavior of ACE on this item. To realize this, the next line must be included in $ACE_ROOT/ace/config.h:

```
#define ACE_DOESNT_INSTANTIATE_NONSTATIC_OBJECT_MANAGER
```

The two function ACE::init() and ACE::fini() must be called explicitly by the process to ensure proper initialization and termination of the ACE library.

8.5.2 CString Class does not use ACE_Allocator

To maintain full control over memory allocation and avoid abending in out-of-memory conditions, try not to use the CString class. CString does not use the ACE_Allocator class; this class enables you to properly manage memory allocation and deallocation. Some of the memory allocation is also done in the constructors and does not allow the interception of errors if not using C++ exceptions (see 8.4.3, "Exceptions" on page 105 for the role of C++ exceptions in net.TABLES).

8.5.3 Overloading the delete operator

$ACE_ROOT/ace/Svc_Handler.h fails to compile the line:

```
void operator delete (void *, void *);
```

This has been corrected in ACE version 5.1.5 with the inclusion of the following in $ACE_ROOT/ace/config-mvs.h:

```
#define ACE_LACKS_PLACEMENT_OPERATOR_DELETE
```

8.5.4 Build time

A full compile and link of ACE takes considerable time on OS/390 (“considerable time” in this context means on the order of 2.5 hours on an otherwise idle LPAR of a G6 machine). The LPAR runs OS/390 2.10 and has 384 Megabytes of memory.

A compile and link of just the main ACE directory, i.e. ignoring the tests, netsvcs, etc., takes about 45 minutes on the same machine.

We did not spend much time trying to optimize the build time of the complete ACE product. Some of the techniques in *Tuning Large C/C++ Applications On OS/390 UNIX Systems Services*, SG24-5606, especially in section 3.5, may help.

9

Establishing a development environment

Before we moved the application to OS/390, we first set up a minimal development environment so the tools necessary for the port were available. We decided that no source code control system was required because OS/390 is not the application's primary platform, the porting period was relatively short, and the team was small.

Setting up the development environment for the port did not take a lot of time. We had the luxury of starting with clean OS/390 systems. We chose OS/390 V2R9 as the primary development platform, and also had access to a beta copy of V2R10.

The V2R9 system (SC59) ran in an LPAR on a 9672-X77 (G6 machine). It had 768 M of central storage and ran OS/390 V2R9. The V2R10 beta system (SC64) ran in an LPAR on a 9672-R76 (G5 machine). It had 384 M of central storage and ran OS/390 V2R10.

Besides setting up OS/390, we customized some tools on the desktop. All of these issues are addressed in this chapter.

9.1 OS/390 install

It is beyond the scope of this book to detail all that is needed for a proper installation of OS/390 and UNIX System Services. Refer to *OS/390 UNIX Planning*, SC28-1890, and to the IBM Redbook *OS/390 Version 2 Release 6 UNIX System Services Implementation and Customization*, SG24-5178, for more information on these subjects. (Although the redbook describes a Release 6 install and is thus somewhat dated, most of the concepts remain the same.)

However, it is still valuable to understand the salient points of the install process. They are as follows:

- V2R9 server pac install
- UNIX System Services customization
- RACF profiles for BPX.DAEMON, users, groups, etc.
- Customize inetd
- Build the TCP/IP stack
- Configure ftpd and syslogd
- Customize file systems
  - Set up /tmp as a Temporary File System (TFS)
  - Set up a separate file system for each user
- Automount user file systems
- Validate the install with the OS/390 UNIX Setup Verification Program (SVP). See the OS/390 UNIX Tools page on the Web at:

  `http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty2.html`

  Search for the string oesvp.

In addition to the previous generic steps, the following explicit values should be considered. The values we used for our project are included.

- BPXPRMxx values
  - MAXCPUTIME(2147483647)
  - MAXASSIZE(2147483647)
  - MAXPROCUSER(10125)
  - FORKCOPY(COW)
  - MAXTHREADS(20000)
  - MAXTHREADTASKS(10000)
- Size of HFS for users: 210 cylinders primary extent, 42 cylinders secondary extents

9.2 Setting up open source tools on OS/390

To port the application, the open source tools `gzip` and `gmake` (at a minimum) were needed to unwind and build ACE. We used the set of open source software tools that comes with the redbook *Open Source Software on OS/390 UNIX*, SG24-5944. For more details, refer to this book. It is on the Web at:

`http://www.redbooks.ibm.com/abstracts/sg245944.html`

The redbook includes the tools both in source and binary format. We chose to build all of the tools from source using the install script 5944.install that is shipped with the package.

We got the source tar file, 5944src.tar, onto the OS/390 system by FTPing it from a PC in binary. We then unwound, built, and installed the source into a separate HFS file system, /usr/local, as follows:

```
MIKEM@SC59:/usr/local> pax -rf 5944src.tar
MIKEM@SC59:/usr/local> ls -F
5944.executables   automake/          flex/              m4/                rsh/
5944.install*      bash/              gmake/             makeBin*        rxvt.ti
5944.manpages      bin/               gnats/             man/               samba/
5944.packages      bison/             groff/             misctools/         share/
5944src.tar        cvs/               gzip/              nedit/             var/
README             diffutils/         idutils/           patch/             vim/
ansi.ti            emacs/             itools/            perl/              xterm/
apache/            examples/          less/              rcs/            xterm.ti
autoconf/          findutils/         lib/               rlogin/
MIKEM@SC59:/usr/local> time 5944.install /usr/local > 5944.log 2>&1 &
[1]     16781148
```

The output of the install script was written to the file 5944.log.

We made sure that everyone on the project had /usr/local/bin in their PATH before /bin, so the unqualified command `make` would pick up GNU make before /bin/make.

```
$ grep PATH .profile
export PATH=$HOME/bin:/usr/local/bin:/bin:.
```

We can see that the `gmake` command being picked up by the PATH is a symbolic link to `/usr/local/bin/make` - both of which are GNU make.

```
$ ls -l `whence gmake`
lrwxrwxrwx   1 STC   SYS1   4 Sep 13 12:19 /usr/local/bin/gmake -> make
```

9.3 Setting up the clients

During the redbook residency, we used Windows NT desktop PCs. To access OS/390 UNIX, we needed a good telnet client and the ability to display using the X Window system (an X server).

9.3.1 A good telnet client

It is commonly accepted that the Windows telnet client is mediocre at best. A much better choice is the freeware package Tera Term Pro. The Tera Term home page is on the Web at:

http://hp.vector.co.jp/authors/VA002416/teraterm.html

From there, click **Download (ttermp23.zip; 943,376 bytes)** to get Tera Term Pro version 2.3 for Windows 95/NT. Unzip it and a simple install is kicked off by opening (invoking) **setup**.

The Tera Term client talking to the OS/390 does have a problem with wrapping lines. In order to have lines wrap properly, we set up the ANSI terminal type on OS/390 UNIX. You must first obtain the ANSI terminal definition file, ansi.ti, which is included with the open source software package (see 9.2, “Setting up open source tools on OS/390” on page 110).

Then you can use the **`tic`** command, which puts terminal entries into the terminfo database. As root, you can invoke the command **`tic ansi.ti`**, which should create the file /usr/share/lib/terminfo/a/ansi. Then any user can put the following command in their .profile:

```
if [ -f /usr/share/lib/terminfo/a/ansi ]; then
  export TERM=ansi
```

```
fi
```

Note that it is not enough to set the TERM environment variable from the command line. Telnet apparently negotiates the protocol at the start of a session, and it is important that TERM is set properly up front.

Other telnet clients may be equally good. For example, some users are quite happy with Attachmate EXTRA!

9.3.2 An X server

An X server is needed on the desktop in order to display the output from X clients, such as xterm, emacs and nedit. We installed Hummingbird's Exceed, which is a popular third-party product. There is also a "try and buy" X server available from MicroImages at:

```
http://www.microimages.com/freestuf/
```

Click **MI/X 2.0 for Windows**.

9.4 Move code from the PC to OS/390 UNIX

Usually when porting an application from one UNIX to OS/390, it is easy to get the code into a tar file. However, on the port described in this redbook, the source code originated from a CD which contained a hierarchical file system.

You'd think it would be easy to get the source code from the PC to OS/390. Although it did not prove to be difficult, some thought was required to get the code over properly and without doing unnecessary work.

There are two differences in the format of text files between the PC and OS/390. The obvious one is that on the PC, text is encoded in ASCII, while on OS/390 it is in EBCDIC. Also, on the PC, each line of text ends with a carriage return, then a line feed (hex 0D0A in ASCII, also referred to as CR-LF), while on OS/390 UNIX, each line ends in a newline (hex 0A in ASCII). Therefore, the text must be converted to EBCDIC and the carriage return characters must be stripped out of each line.

The de facto compression and archival format on the PC is zip files, while on UNIX the archival format is USTAR with a choice of compression formats (see 3.1, "Archiving, compression and text translation" on page 40).

So we had the choice of moving a zip file, or moving a tar file, or moving each file over individually without compressing and archiving into a single file. We considered each possibility:

- Creating a zip file on the PC is easy, but a zip tool is not common on OS/390 (though Info-ZIP is available).
- It is easy to unwind and translate a tar file on OS/390, but it is not easy to create one on the PC. For example, the popular Winzip reads but does not write tar files, as is illustrated in the following quote from the WinZip help: "Since almost all new archives are created in Zip format (sic) WinZip does not provide facilities to add to or create files in these formats."
- It seems cumbersome and error-prone to use an FTP command-line client to move a directory structure over to OS/390. The FTP subcommand **`mput *.*`** could be used to move an entire directory at a time, but each directory would have to be visited.

However, some GUI-based FTP clients on the PC allow you to drag and drop directories, and they recursively move all files and subdirectories as well. One FTP client that offers this function is the one that came with Hummingbird Exceed. This approach has the added benefit that the FTP will perform both the text translation to ASCII and strip out the carriage return characters that are not necessary on UNIX.

In the end, we used the third approach. However, we also document the second approach.

9.4.1 The PC nttar command

A freeware version of the GNU **tar** command is available for the PC. We searched the Internet for the string “nttar” and found it at:

http://www.winsite.com/

The **nttar** command runs as a DOS program, but requires path names to use forward slash instead of backslash. For example, the following command will create the tar file myfiles.tar and populate it with all files and directories in or below C:\some\dir:

```
tar -cf myfiles.tar c:/some/dir
```

Once you have a tar file on the PC, you can use native Windows FTP from DOS to transfer the tar file to OS/390 UNIX. Be sure to transfer in binary mode. Then on OS/390, you can use the following command to unpack the files and directories:

```
pax -r -o from=ISO8859-1 -f myfiles.tar
```

The `-o from=ISO8859-1` option converts from ASCII to EBCDIC, assuming the default to value (IBM-1047, in this case) is correct. Conversely, the following command allows you to create an ASCII tar file from all files below the mydir directory:

```
pax -wf myfiles.tar mydir -o from=IBM-1047,to=ISO8859-1
```

9.4.2 The Hummingbird FTP client

With all Hummingbird products, a large set of UNIX and TCP/IP tools are also available. When installing (Exceed, for example), be sure to include the additional tools (which is the default). Once you have installed and brought up the FTP client, be sure to set Options-Preferences-General-Use Original case for Local Drives to ON; otherwise, you may not get the correct case in filenames on the host.

By default, FTP will convert between ASCII and EBCDIC for text transfers. By setting binary transfers on, text translation is turned off.

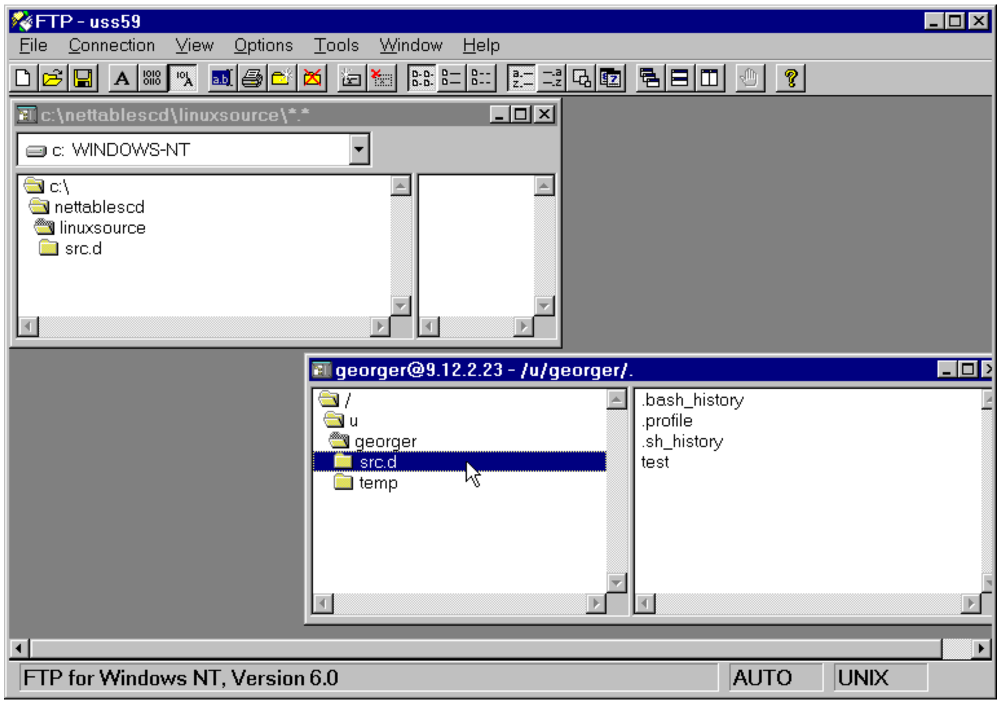

FTP - uss59
File Connection View Options Tools Window Help
c:\nettablescd\linuxsource\*.*
c: WINDOWS-NT
c:\
nettablescd
linuxsource
src.d
georger@9.12.2.23 - /u/georger/.
/
u
georger
src.d
temp
.bash_history
.profile
.sh_history
test
FTP for Windows NT, Version 6.0
AUTO
UNIX

Figure 9-1 Hummingbird FTP GUI

10

Porting net.TABLES

We made these observations while porting to OS/390 V2R9, although many of them are relevant for other versions of OS/390 UNIX as well.

10.1 Coding considerations

As explained in 8.3, “How net.TABLES was developed” on page 103, we made some preliminary attempts at porting the code before this residency. Our previous attempts led to major rewrites of some sections of the code. The end result is a highly portable code base and we encountered only minor incompatibilities during this residency.

This section describes the problems we solved both during this residency and in our previous efforts.

10.1.1 64-bit integers - long long data type

Our application uses 64-bit integers. To increase the portability of the code, we use the *ACE_UINT64* data type provided by ACE. ACE can use either the supported `long long int` or can emulate it for systems where it is not available. This was the case for OS/390 until version 2.6 when support for a native 64-bit integer was introduced.

To get support for the native 64-bit integer on OS/390, we use the following compiler option (see the explanation of langlvl in 4.1.4, “Commonly used z/OS C/C++ options” on page 59):

```
-Wc,”langlvl(extended)”
```

The $ACE_ROOT/ace/config-mvs.h file should contain something like:

```
#if __COMPILER_VER__ < 0x22060000   /* before OS/390 r2.6 */
#define ACE_LACKS_LONGLONG_T
#endif /* __COMPILER_VER__ < 0x22060000 */
```

This is now part of ACE beginning with version 5.1.5.

10.1.2 Signal handlers and pointers to C++ functions

With OS/390, signal handlers must use C linkage, not C++ linkage. We rewrote the signal handling routine of net.TABLES to accommodate this. See 4.6.3, “Extern “C”” on page 72 for a more detailed explanation.

10.1.3 Enums

The compiler will define a field only large enough to hold the largest value of an `enum`. For example:

```
typedef enum {
SUCCESSFUL =  0,
FAILED   = -1,
} ENUM_1_BYTE;
```

This enum will actually only occupy one byte of storage. Compilers on other operating systems may treat enums as int and reserve two or four bytes.

Our application (net.TABLES) contained an enum as part of the header of the command buffer exchanged between client and server. This caused a problem when the enum was compiled to different lengths on the different platforms. One workaround for this problem is to include a dummy value in your enum to force the size of field that you need. For example:

```
// This example only works on systems where an int is 4 bytes
typedef enum {
SUCCESSFUL = 0,
FAILED   = -1,
ENUM_32  = INT_MAX
} ENUM_4_BYTES;
```

In our case, we solved this problem by rewriting the communication protocol to only use fixed-length integers.

See also section 4.6.2, “Enum size” on page 72 for a further discussion on enumerations.

10.1.4 Big-endian and little-endian byte ordering

Byte order is the order in which bytes of an integer are stored by a machine’s hardware. It is different on OS/390 than on Intel-based machines. Integers with the most significant digits on the *left* (as you would normally write them) are called “big-endian”. Integers with the most significant digits on the *right* are called “little-endian”. Some machines, including those that run OS/390, use big-endian. Intel-based machines use little-endian.

The conventional method of communicating between processes on different machines is to translate everything into network-byte order (big endian) before sending it. Then the receiving process translates the message from network-byte order into whatever format it wants to use.

Processes designed to be operating system- or platform-independent usually do the conversion to and from network byte order. The functions **htonl** and **htons** are used to translate integers (long and short) from host to network byte order, while similar functions, `ntohl` and `ntohs`, do the inverse translations. On big-endian machines, these functions do nothing, but on little-endian machines, the bytes are swapped.

In order to increase the performance of net.TABLES, we do not execute this translation if both client and server are running on machines with identical endian characteristics. The relative byte order between the two processes is determined when the initial communication is established.

10.1.5 ASCII/EBCDIC conversions

OS/390 uses the EBCDIC character set, while Linux and Windows NT use ASCII[1]. Any text fields transmitted between programs running on OS/390 UNIX and programs running on other operating systems must be translated between character sets. This is not a difficult operation, merely a table lookup and character substitution, as can be seen from this example:

```
//====================================================================
// EBCDIC to ASCII

char chEBCDICtoASCII[257] =
{
  "\x00\x01\x02\x03\xCF\x09\xD3\x7F\xD4\xD5\xC3\x0B\x0C\x0D\x0E\x0F" // 00-0F
  "\x10\x11\x12\x13\xC7\xB4\x08\xC9\x18\x19\xCC\xCD\x83\x1D\xD2\x1F" // 10-1F
  "\x81\x82\x1C\x84\x86\x0A\x17\x1B\x89\x91\x92\x95\xA2\x05\x06\x07" // 20-2F
  "\x20\xEE\x16\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\x20\xC1\x1A" // 30-3F
  "\x20\xA6\xE1\x80\xEB\x90\x9F\xE2\xAB\x8B\x9B\x2E\x3C\x28\x2B\x7C" // 40-4F
  "\x26\xA9\xAA\x9C\xDB\xA5\x99\xE3\xA8\x9E\x21\x24\x2A\x29\x3B\x5E" // 50-5F
  "\x2D\x2F\xDF\xDC\x9A\xDD\xDE\x98\x9D\xAC\xBA\x2C\x25\x5F\x3E\x3F" // 60-6F
  "\xD7\x88\x94\xB0\xB1\xB2\xFC\xD6\xFB\x60\x3A\x23\x40\x27\x3D\x22" // 70-7F
  "\xF8\x61\x62\x63\x64\x65\x66\x67\x68\x69\x96\xA4\xF3\xAF\xAE\xC5" // 80-8F
  "\x8C\x6A\x6B\x6C\x6D\x6E\x6F\x70\x71\x72\x97\x87\xCE\x93\xF1\xFE" // 90-9F
  "\xC8\x7E\x73\x74\x75\x76\x77\x78\x79\x7A\xEF\xC0\xDA\x5B\xF2\xF9" // A0-AF
  "\xB5\xB6\xFD\xB7\xB8\xB9\xE6\xBB\xBC\xBD\x8D\xD9\xBF\x5D\xD8\xC4" // B0-BF
```

[1] During this residency, the only two code sets we used were the IBM-1047 and the ISO-8859-1 (or Latin-1).

```
   "\x7B\x41\x42\x43\x44\x45\x46\x47\x48\x49\xCB\xCA\xBE\xE8\xEC\xED" // C0-CF
   "\x7D\x4A\x4B\x4C\x4D\x4E\x4F\x50\x51\x52\xA1\xAD\xF5\xF4\xA3\x8F" // D0-DF
   "\x5C\xE7\x53\x54\x55\x56\x57\x58\x59\x5A\xA0\x85\x8E\xE9\xE4\xD1" // E0-EF
   "\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\xB3\xF7\xF0\xFA\xA7\xFF" // F0-FF
};

//---------------------------------------------------------------------

void DklFromIBM1047String( char * str )
{
   int len, i;

   len = strlen( str );
   for( i = 0; i < len; ++i )
      str[i] = chEBCDICtoASCII[ str[i] ];
}

//=====================================================================
// ASCII to EBCDIC


char chASCIItoEBCDIC[257] =
{
   "\x00\x01\x02\x03\x37\x2D\x2E\x2F\x16\x05\x25\x0B\x0C\x0D\x0E\x0F" // 00-0F
   "\x10\x11\x12\x13\x3C\x3D\x32\x26\x18\x19\x3F\x27\x22\x1D\x35\x1F" // 10-1F
   "\x40\x5A\x7F\x7B\x5B\x6C\x50\x7D\x4D\x5D\x5C\x4E\x6B\x60\x4B\x61" // 20-2F
   "\xF0\xF1\xF2\xF3\xF4\xF5\xF6\xF7\xF8\xF9\x7A\x5E\x4C\x7E\x6E\x6F" // 30-3F
   "\x7C\xC1\xC2\xC3\xC4\xC5\xC6\xC7\xC8\xC9\xD1\xD2\xD3\xD4\xD5\xD6" // 40-4F
   "\xD7\xD8\xD9\xE2\xE3\xE4\xE5\xE6\xE7\xE8\xE9\xAD\xE0\xBD\x5F\x6D" // 50-5F
   "\x79\x81\x82\x83\x84\x85\x86\x87\x88\x89\x91\x92\x93\x94\x95\x96" // 60-6F
   "\x97\x98\x99\xA2\xA3\xA4\xA5\xA6\xA7\xA8\xA9\xC0\x4F\xD0\xA1\x07" // 70-7F
   "\x43\x20\x21\x1C\x23\xEB\x24\x9B\x71\x28\x38\x49\x90\xBA\xEC\xDF" // 80-8F
   "\x45\x29\x2A\x9D\x72\x2B\x8A\x9A\x67\x56\x64\x4A\x53\x68\x59\x46" // 90-9F
   "\xEA\xDA\x2C\xDE\x8B\x55\x41\xFE\x58\x51\x52\x48\x69\xDB\x8E\x8D" // A0-AF
   "\x73\x74\x75\xFA\x15\xB0\xB1\xB3\xB4\xB5\x6A\xB7\xB8\xB9\xCC\xBC" // B0-BF
   "\xAB\x3E\x3B\x0A\xBF\x8F\x3A\x14\xA0\x17\xCB\xCA\x1A\x1B\x9C\x04" // C0-CF
   "\x34\xEF\x1E\x06\x08\x09\x77\x70\xBE\xBB\xAC\x54\x63\x65\x66\x62" // D0-DF
   "\x30\x42\x47\x57\xEE\x33\xB6\xE1\xCD\xED\x36\x44\xCE\xCF\x31\xAA" // E0-EF
   "\xFC\x9E\xAE\x8C\xDD\xDC\x39\xFB\x80\xAF\xFD\x78\x76\xB2\x9F\xFF" // F0-FF
};

//---------------------------------------------------------------------

void DklToIBM1047String( char * str )
{
   int len, i;

   len = strlen( str );
   for( i = 0; i < len; ++i )
      str[i] = chASCIItoEBCDIC[ str[i] ];
}
```

The original version of the conversion tables was borrowed from the ACE toolkit and can be found in the file $ACE_ROOT/ace/Codeset_IBM1047.cpp.

Note: Use care in the order in which you do ASCII/EBCDIC translation and byte-order translation. For example, if a message contains integer control information about itself, such as the length of text fields within the message, you must convert the byte-order of the control fields before translating the text fields.

10.1.6 Bool data type not supported

The bool data type was not supported in the OS/390 C/C++ compiler. Though it is now supported in the z/OS V1R2 C/C++ compiler, we created an include file containing the following code to work around this:

```
#ifndef BOOL_H
#define BOOL_H
#ifdef __MVS__
typedef int bool;
const bool false = 0;
const bool true = 1;
#endif /* __MVS__ */
#endif /* BOOL_H */
```

10.1.7 Non-standard header files

Some macros and functions may not be in the same header files on OS/390 as on other platforms. On OS/390, for example, TCP_NODELAY is defined in /usr/include/xti.h. In Linux, the definition is in /usr/include/netinet/tcp.h. We did not realize this at first and assumed that TCP_NODELAY was not available at all in OS/390.

Usually header files as defined by POSIX are standard. When they are not, they must simply be handled on a case-by-case basis.

10.1.8 Thread_id data type

On OS/390, you cannot treat POSIX thread_id as an int. It is a struct; therefore, you cannot initialize it to zero (in a constructor, for example) as you can on Linux or Win32. You also must be careful when using thread_id for something other than managing threads (for example, when writing diagnostics or logging messages).

10.1.9 Overloading the delete operator

$ACE_ROOT/ace/Svc_Handler.h fails to compile the line:

```
void operator delete (void *, void *);
```

This has been corrected in ACE version 5.1.5 with the inclusion of

```
#define ACE_LACKS_PLACEMENT_OPERATOR_DELETE
```

in $ACE_ROOT/ace/config-mvs.h.

10.2 Build considerations

We strongly recommend using GNU make instead of the default make provided by OS/390. If you are porting from another UNIX system, then you are probably using GNU make for that system and using the same make on OS/390 will make your life easier. See 3.4, “Make tools” on page 44 for more information.

10.2.1 Failed bind and unusable executables

A bind operation may create an executable and not delete it if the operation does not complete successfully. Part of the reason for this is that, by default, OS/390 treats some messages as simply warnings although the developer regards them as error conditions.

You can add the target .DELETE_ON_ERROR to your makefile. Then make will delete the executable if the binder return code is 8 or higher. (The binder returns only success or failure to make. Success is a return code of 0 or 4; failure is a return code of 8 or higher.)

You can set an _ACCEPTABLE_RC environment variable for the compiler. This effectively removes the executable if the return code is greater than the variable setting.

There is different variable for each compiler:

- Use _CC_ACCEPTABLE_RC for the CC compiler
- Use _C89_ACCEPTABLE_RC for the C89 compiler
- Use _CXX_ACCEPTABLE_RC for the C++ compiler; for example:

```
export_CXX_ACCEPTABLE_RC=0
```

A good example of this is provided by ACE. In its implementation of some of its template classes, ACE explicitly calls the destructors of scalar types. A warning will be issued and if _ACCEPTABLE_RC is set to zero, the compilation will fail.

10.2.2 Identifying the operating system in makefiles

Since compilers may use different options and have different names for the same options, etc., one of the first crucial points in our effort to port net.TABLES was to provide a way for the makefiles to identify correctly the underlying operating system (OS). There are different ways to do this:

Manually
This is how ACE does it. The makefile containing the proper variable definitions for a specific operating system is copied manually to the proper location and is included by the overall makefile. This operation must be repeated for every installation.

Use an environment variable
You can define a new environment variable, or use an existing environment variable (the bash shell, for example, provides the variable OSTYPE).

Use a makefile variable
The type of operating system can also be extracted with the uname command and assigned to a make variable:

```
THE_OS := $(shell uname -s)
```

We chose to use a makefile variable since it is simple to use and eliminates most potential mistakes. Our makefiles use the THE_OS variable to import the proper variables (macros), define the build targets, etc., for each operating system.

Here are two examples of this solution:

```
#
# Example 1. inclusion of local macros
#
ifeq ($(THE_OS),OS/390)
include ../../mvs.make
else
...
endif

#
# Example 2. really_clean target
```

```
#
ifeq ($(THE_OS),OS/390)
really_clean: clean
-rm -f objects/*.rpo CEEDUMP.*
else
really_clean: clean
-rm -f core depends/*.d
endif
```

10.2.3 Generation of dependency files

Maintaining dependencies between source files on relatively large projects is a huge task. We used the -MM option provided by the GNU compiler (gcc on Linux on the PC) to automatically create dependency files. The following piece of code was used.

```
# --------------------------------
# How to get DEPENDENCIES
# --------------------------------
ifeq ($(THE_OS),OS/390)
   depends/%.d : %.cpp
   touch depends/$*.d
else
   depends/%.d : %.cpp
   if [ ! -d depends  ] ; then mkdir depends; fi;
   -rm -f depends/$*.d
   $(CC) $(CFLAGS) -MM $*.cpp | sed 's/^\(.*\).o[ ]*:/objects\/\1.o \
   depends\/\1.d :/' > depends/$*.d
endif
```

An alternative would be to use **makedepend** on z/OS. Because we already had the dependencies generated, we chose not to do this.

10.2.4 Creation of debug and release versions

On Linux, we created the release versions of the executables by removing the symbols from the debug versions using the command:

```
strip -o exec_name exec_name_dbg
```

To create both debug and release versions, we had to compile each source file twice, providing different options to the compiler, and we had to bind each version of the executables. Here is an example of the GNU make variables we used to compile net.TABLES during this residency:

```
# release version:
CFLAGS += -c -+ -I. -I../../include -Wc,NOANSIALIAS -O2
# debug version:
DCFLAGS += -c -+ -g -I. -I../../include -I../../ACE -Wc,GONUM
```

10.3 Run-time considerations

The following run-time considerations were addressed.

10.3.1 LIBPATH

LIBPATH specifies directories to be searched for DLLs. If it is not set, the current working directory is searched. For example:

```
LIBPATH=/lib:/usr/lib:.:/u/georger/ACE_wrappers/ace
```

10.3.2 _BPX_SHAREAS and _BPX_SPAWN_SCRIPT

We set both the environment variables _BPX_SHAREAS and _BPX_SPAWN_SCRIPT to YES to get better response in the command shell. See 7.1, “Environment variables related to spawn” on page 94 for more details.

10.4 Reconciliation with original Linux version

Once we successfully ported net.TABLES to OS/390 UNIX, we had to take the source back to Linux and rebuild the application to ensure that any changes made for OS/390 UNIX did not break on Linux.

This process mostly involved synchronizing the makefiles. The whole process took about an hour. We then took the source back to OS/390 and re-built the application without a glitch (not counting the setup of ACE as explained in 6.2.2, “Building ACE” on page 90).

11

Tuning net.TABLES

This chapter describes the parameters we adjusted to improve the application’s runtime performance.

11.1 Tuning for an improved performance

Several techniques are available to optimize the performance of net.TABLES. We measure the relative performance of the program by measuring the time of execution of three characteristic API function calls - SearchByKey, GetByKey and GetFirst. Table 11-1 shows the parameters we use for these tests.

Table 11-1 Values of some parameters used in the tests

Test characteristics	Values
Number of tables	100
Number of rows per table	100
Number of bytes per row[a]	320
Number of bytes in a key	20
Number of calls per run[b]	100,000

a. The tables are created with variable length rows but, for practical reasons, all inserted rows have the same length.
b. This value represents how many times each of the three measured functions are called.

Note the following regarding the optimization tests:

- The measured time is the clock time. Since net.TABLES is a client/server application, the elapsed time viewed by the clients is our most important metric.
- The tests were run a few times to eliminate any possible bias caused by other processes running at the same time.
- To measure accurately the performance of the table engine, we compile and link the client library and the server into a unique executable. This procedure eliminates one of the standard bottlenecks of client/server software, namely the elapsed time in the communication protocols, and enables the optimization of the core engine.

11.1.1 Compiler/linker optimization techniques

As with most C/C++ compilers, you can specify different levels of optimization. With OS/390, you can also specify InterProcedural Analysis (IPA) to further optimize your code. Specifying debug or nodebug (+-g) will also affect the application's performance.

11.1.2 Performance Analyzer

We also used the performance analyzer to identify functions that were the most frequently called. See Appendix F, "Performance analyzer output" on page 179.

11.1.3 Effect of compiler optimization settings

Table 11-2 shows the results obtained with different optimization levels. We got our best results using the IPA linker optimization.

Table 11-2 Tuning performance (all values are expressed in thousands of calls per second)

Tuning options (compiler options)	**SearchByKey**	**GetByKey**	**GetFirst**
-g -0[a]	35	32	81
-0[b]	92	85	212
-2[c]	195	167	368
-2, -Wc,IPA(OBJONLY)[d]	192	169	371
-0 -WI[e]	261	220	504
-2 -WI[f]	267	224	503

a. debug ON and no optimization
b. no optimization
c. best optimization
d. best optimization and IPA during compile
e. no optimization and IPA pre-link step
f. best optimization and IPA pre-link step

11.1.4 Other compiler/linker tuning techniques

Besides the optimization level, there are compiler and linker options that can affect performance.

EDIT=NO

We reduced the size of the executables and dynamic libraries by about 50% by using the EDIT=NO binder option (-WI,EDIT=NO). However, this reduction in size has no noticeable effect on the performance of net.TABLES.

ARCH(3)

ARCH specifies the lowest hardware level on which your application will run. The compiler may generate code containing instructions not available on lower-level machines. We did not see any significant increase in performance by setting ARCH for the hardware that we used.

TUNE(3)

TUNE instructs the compiler to optimize the code for fastest performance on the specified level of hardware architecture machines. We did not see any significant increase in performance by setting TUNE for the hardware that we used.

Built-in functions

Some include files use built-in functions and some do not. For example, string.h has built-in functions memcpy and memset, while strings.h has non-built-in versions of the memcpy and memset. Using the built-in versions gave up to 30% improvement for these tests.

Effect of other compiler/linker tuning techniques

Table 11-3 shows the results of setting ARCH and TUNE and using built-in functions in string.h. We have included the best performing optimization results (-2 -WI) for comparison (all tests are run with our best optimization options turned on).

Table 11-3 More tuning performance (all values are expressed in thousands of calls per second)

Tuning options	SearchByKey	GetByKey	GetFirst
-2 -WI	267	224	503
-Wc,ARCH(3)	264	216	503
-Wc,TUNE(3)	266	218	500
-Wc,ARCH(3),TUNE(3)	261	216	517
Using string.h	350	291	647

11.1.5 Runtime tuning techniques

OS/390 UNIX supplies many parameters for changing the characteristics of a program at runtime in addition to the usual commands, such as `ulimit` and `nice`, found on other UNIXes.

Default runtime parameters

You can obtain the current values of runtime parameters by executing the program with the environment variable _CEE_RUNOPTS set on:

```
export _CEE_RUNOPTS='RPTOPTS(ON)'
```

Here is an extract from the report produced by this setting:

```
Options Report for Enclave main 07/13/00 3:14:48 PM
Language Environment V2 R9.0

LAST WHERE SET              OPTION
----------------------------------------------------------------------------
Installation default          ABPERC(NONE)
Installation default          ABTERMENC(ABEND)
Installation default        NOAIXBLD
Installation default          ALL31(OFF)
Installation default          ANYHEAP(16384,8192,ANYWHERE,FREE)
Installation default        NOAUTOTASK
Installation default          BELOWHEAP(8192,4096,FREE)
Installation default          CBLOPTS(ON)
Installation default          CBLPSHPOP(ON)
Installation default          CBLQDA(OFF)
Installation default          CHECK(ON)
Installation default          COUNTRY(US)
Installation default        NODEBUG
Installation default          DEPTHCONDLMT(10)
Installation default          ENVAR("")
Installation default          ERRCOUNT(0)
Installation default          ERRUNIT(6)
Installation default          FILEHIST
Default setting             NOFLOW
Installation default          HEAP(32768,32768,ANYWHERE,KEEP,8192,4096)
Installation default          HEAPCHK(OFF,1,0)
Installation default          HEAPPOOLS(OFF,8,10,32,10,128,10,256,10,1024,10,2048,10)
Installation default          INFOMSGFILTER(OFF,,,,)
Installation default          INQPCOPN
Installation default          INTERRUPT(OFF)
Installation default             LIBRARY(SYSCEE)
Installation default             LIBSTACK(4096,4096,FREE)
Installation default             MSGFILE(SYSOUT,FBA,121,0,NOENQ)
Installation default             MSGQ(15)
Installation default             NATLANG(ENU)
Installation default           NONONIPTSTACK(4096,4096,BELOW,KEEP)
Installation default             OCSTATUS
Installation default           NOPC
Installation default             PLITASKCOUNT(20)
Default setting                  POSIX(ON)
Installation default             PROFILE(OFF,"")
Installation default             PRTUNIT(6)
Installation default             PUNUNIT(7)
Installation default             RDRUNIT(5)
Installation default             RECPAD(OFF)
Invocation command               RPTOPTS(ON)
Invocation command               RPTSTG(ON)
Installation default           NORTEREUS
Installation default             RTLS(OFF)
Installation default           NOSIMVRD
Installation default             STACK(131072,131072,BELOW,KEEP)
Installation default             STORAGE(NONE,NONE,NONE,8192)
Installation default             TERMTHDACT(TRACE)
Installation default           NOTEST(ALL,"*","PROMPT","INSPPREF")
Installation default             THREADHEAP(4096,4096,ANYWHERE,KEEP)
Installation default             TRACE(OFF,4096,DUMP,LE=0)
Installation default             TRAP(ON,SPIE)
Installation default             UPSI(00000000)
Installation default           NOUSRHDLR(,)
Installation default             VCTRSAVE(OFF)
Installation default             VERSION()
Installation default             XUFLOW(AUTO)
```

Collecting runtime statistics

To effectively optimize the runtime environment, you need to collect statistics on the current environment. First, set the environment variable _CEE_RUNOPTS to the appropriate value:

```
export _CEE_RUNOPTS='RPTSTG(ON)'
```

Then run the target program with a typical load (see Table 11-1 on page 124).

```
Storage Report for Enclave main 07/13/00 3:04:16 PM
Language Environment V2 R9.0

    STACK statistics:
      Initial size:                                      131072
      Increment size:                                    131072
      Maximum used by all concurrent threads:             23912
      Largest used by any thread:                         23912
      Number of segments allocated:                           1
      Number of segments freed:                               0
    NONIPTSTACK statistics:
      Initial size:                                           0
      Increment size:                                         0
      Maximum used by all concurrent threads:                 0
      Largest used by any thread:                             0
      Number of segments allocated:                           0
      Number of segments freed:                               0
    LIBSTACK statistics:
      Initial size:                                        4096
      Increment size:                                      4096
      Maximum used by all concurrent threads:                816
      Largest used by any thread:                            816
      Number of segments allocated:                            1
      Number of segments freed:                                0
    THREADHEAP statistics:
      Initial size:                                         4096
      Increment size:                                       4096
      Maximum used by all concurrent threads:                  0
      Largest used by any thread:                              0
      Successful Get Heap requests:                            0
      Successful Free Heap requests:                           0
      Number of segments allocated:                            0
      Number of segments freed:                                0
    HEAP statistics:
      Initial size:                                        32768
      Increment size:                                      32768
      Total heap storage used (sugg. initial size):      3681016
      Successful Get Heap requests:                        12121
      Successful Free Heap requests:                       12071
      Number of segments allocated:                          116
      Number of segments freed:                                0
    HEAP24 statistics:
      Initial size:                                         8192
      Increment size:                                       4096
      Total heap storage used (sugg. initial size):            0
      Successful Get Heap requests:                            0
      Successful Free Heap requests:                           0
      Number of segments allocated:                            0
      Number of segments freed:                                0
    ANYHEAP statistics:
      Initial size:                                        16384
      Increment size:                                       8192
      Total heap storage used (sugg. initial size):        84792
      Successful Get Heap requests:                         2897
      Successful Free Heap requests:                        2859
      Number of segments allocated:                           10
      Number of segments freed:                                9
    BELOWHEAP statistics:
      Initial size:                                         8192
      Increment size:                                       4096
      Total heap storage used (sugg. initial size):           32
      Successful Get Heap requests:                            0
      Successful Free Heap requests:                           0
      Number of segments allocated:                            1
      Number of segments freed:                                0
    Additional Heap statistics:
      Successful Create Heap requests:                         0
      Successful Discard Heap requests:                        0
      Total heap storage used:                                 0
      Successful Get Heap requests:                            0
      Successful Free Heap requests:                           0
      Number of segments allocated:                            0
      Number of segments freed:                                0
    Largest number of threads concurrently active:             1
End of Storage Report
```

HEAP and ANYHEAP

The analysis of this report shows that at least two runtime parameters, HEAP and ANYHEAP, could be modified to increase the performance of the program. The total size used of each of these exceeds the initial size. Thus, the application had to make several calls to get enough storage. By setting the initial size to the total size reported, we can allocate all the storage at once and avoid the overhead of multiple allocations.

```
export _CEE_RUNOPTS='HEAP(3681016),ANYHEAP(84792)'
```

Setting the initial HEAP and ANYHEAP sizes did not make any significant difference to net.TABLES' performance except in an unusual case where we set the initial HEAP size to be very small and then inserted a large number of rows into a table, thus causing the HEAP to be expanded many times. We easily obtained a 50% degradation in performance by doing this.

12

Conclusions from net.TABLES port

The following conclusions and recommendations are based on porting net.TABLES to OS/390 UNIX.

12.1 General

The process of porting net.TABLES to OS/390 UNIX was relatively short and painless. This was mostly due to:

- Using ACE to hide most of the platform dependencies
- Trying the port beforehand at other sites and modifying our code to accommodate the OS/390 UNIX differences that were not handled by ACE
- Having a fast machine and fast communications

We recommend that you reserve enough time for the port. We found issues in earlier attempts that took us completely by surprise.

12.2 Environment

Optimize the compiler on your system by, for example, placing commonly used header files in filecache. Be aware that compiles on OS/390 can be slow and may be frustrating for developers accustomed to other platforms.

IBM's TCP/IP seems to be optimized for large information transfers such as FTP, but not for client/server types of communications which involve many, many small messages passed between client and server. We did not try to optimize this due to lack of time. However, using some of the general OS/390 UNIX tuning techniques may help.

12.3 Coding

The following coding conclusions can be drawn.

12.3.1 Coding for portability

These recommendations may seem obvious but are important enough to state:

- Use wrappers (ACE, some other product, or a homegrown version), #ifdefs, or some other abstraction technique to isolate application logic from platform-dependent code. Using ACE greatly reduced the coding and porting time for net.TABLES.
- Keep the application as POSIX-compliant as possible and do not use platform-dependent extensions.
- If you are porting an application that is being developed concurrently on other platforms, then do the port early in the development cycle. You may encounter a need to rewrite parts of code and this is much cheaper and politically easier before the code has become set on the other platform.

12.3.2 Coding for client/server applications

Following is a list of points you should keep in mind if your application shares information between disparate platforms or operating systems:

- Byte order of integers, i.e., big endian or little endian (see 10.1.4, "Big-endian and little-endian byte ordering" on page 117)
- ASCII, EBCDIC and other character sets or encoding, e.g. UNICODE (see 10.1.5, "ASCII/EBCDIC conversions" on page 117)

- Alignment of fields within C structures, unions and C++ classes

12.4 Building

Allow for OS/390 in the makefiles early in the project. We spent considerable time getting the makefiles generalized enough to work on all our target platforms.

Of the performance tuning techniques that we tried, we got our best results using:

- Optimization level 2
- Inter Procedural Analysis (IPA)
- The built-in functions in string.h

Applications with other characteristics, e.g. a big number cruncher, may find other techniques are more useful, especially using built-in functions from other header files.

12.5 Tools

Use of any particular tool is often a matter of personal preference (`vi` inspires strong opinions in some people), but here are some general observations:

- Use the open source software tools for OS/390 (GNU make, gzip, bash, etc.) and invest the time needed to become familiar with them because they can make your life much easier.
- There are a couple of IDEs for OS/390. They are both useful if you are accustomed to IDEs, but can be slow. If you are accustomed to command line UNIX and editing with `vi`, then OS/390 is comparable to other UNIX environments.
- Use a fast and easy file sharing or file transfer scheme between the desktop and OS/390. SAMBA worked well for us. A Windows FTP program is a good second choice.
- Although we did not investigate the results of the Performance Analyzer program during this port, we will use it in future to tune and improve net.TABLES. The Performance Analyzer is a powerful tool.

13

Porting Xalan-C++

Xalan is an Extensible Stylesheet Language Transformations (XSLT) processor for transforming Extensible Markup Language (XML) documents into HTML, text or other XML document types.

Xalan-C++ version 1.1 is a robust implementation of the W3C Recommendations for XSL Transformations (XSLT) and the XML Path Language (XPath). For more information, see the Web site:

    http://xml.apache.org/xalan-c/index.html

We chose to use this porting experience as an example because its extensive use of the Standard Template Library.

13.1 Requirements

z/OS build requires externals from

- ICU 1.8.1
  The International Components for Unicode(ICU) is a C and C++ library that provides robust and full-featured Unicode support on a wide variety of platforms.

  `http://oss.software.ibm.com/icu`

- XML4C3.5.1
  XML parsers in C++.

  `http://www.alphaworks.ibm.com/tech/xml4c`

You can get the XML Toolkit for z/OS and OS/390 that contains both of the above with precompiled binaries at:

`http://www.ibm.com/servers/eserver/zseries/software/xml/`

The building of these externals is beyond the scope of this book. We will assume these are ready to use.

13.2 Porting considerations

Because Xalan was developed based on the SGI port of STL, there are a few changes required to build it on z/OS V1R2 systems, which include the C++ Standard Library, containing a standard-compliant STL.

With the SGIport STL, Xalan version 1.2 will build natively on an OS/390 V2R10 system. So in the following sections, we simply discuss how to get Xalan to build on a z/OS V1R2 system using the C++ Standard Library.

13.2.1 Getting the source code on 390

You need to download the source package from the Web site already mentioned, and FTP it to 390 as bin. You will need gzip/gunzip installed on your system to unpack the tar.gz file.

If you do not have gzip/gunzip installed, go to the Tools and Toys Web page at

`http://www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html`

You will also need **`gmake`,** which is also available at the Tools and Toys Web page.

To unpack the files:

```
$ gunzip Xalan-C_1_2-aix.tar.gz
$ pax -r -f Xalan-C_1_2-aix.tar -ofrom=ISO8859-1,to=IBM-1047
```

This generates some error message about “cannot create file,” which can be ignored. The unpacked source code is located in xml-xalan/c/src.

13.2.2 Changes required in the Xalan source code

Starting from the extracted directory for Xalan:

```
/home/buildid/xml-xalan/c/
```

src/Include/OS390Definitions.hpp

The following definitions are needed for platform-dependent configurations for the build of Xalan. Simply modify the existing OS390Definitions.hpp file in the Include directory to conform to the following:

```
#define XALAN_PLATFORM_EXPORT _Export
#define XALAN_PLATFORM_IMPORT _Export
#define XALAN_PLATFORM_EXPORT_FUNCTION(T) T XALAN_PLATFORM_EXPORT
#define XALAN_PLATFORM_IMPORT_FUNCTION(T) T XALAN_PLATFORM_IMPORT

#define XALAN_NO_MEMBER_TEMPLATES
#define XALAN_NO_COVARIANT_RETURN_TYPE
#define XALAN_XALANDOMCHAR_USHORT_MISMATCH
#define XALAN_USE_WCHAR_CAST_HACK
#define XALAN_BIG_ENDIAN
#define XALAN_NON_ASCII_PLATFORM
#define XALAN_USE_XERCES_LOCAL_CODEPAGE_TRANSCODERS
#define XALAN_POSIX2_AVAILABLE
#define XALAN_ICU_DEFAULT_LOCALE_PROBLEM
#define XALAN_UNALIGNED
```

src/Include/STLHelper.hpp

Line 119, replace

```
struct select1st : public std::unary_function<PairType, PairType::first_type>
```

with

```
struct select1st : public std::unary_function<PairType, PairType>
```

Line 125, replace

```
typedef std::unary_function<PairType, PairType::first_type>     BaseClassType;
```

with

```
typedef std::unary_function<PairType, PairType> BaseClassType;
```

Line 156, replace

```
struct select2nd : public std::unary_function<PairType, PairType::second_type>
```

with

```
struct select2nd : public std::unary_function<PairType, PairType>
```

Line 162, replace

```
typedef std::unary_function<PairType, PairType::second_type>    BaseClassType;
```

with

```
typedef std::unary_function<PairType, PairType> BaseClassType;
```

src/Makefile.in

Under the OS390 SPECIFIC OPTIONS, in the 2 lines of PLATFORM_COMPILE_OPTIONS, remove

```
-I ${STLPORTROOT}/stlport
```

and add

```
-D_MSE_PROTOS
```

This is what the resulting 2 lines look like: (The following wrapped around due to length.)

```
PLATFORM_COMPILE_OPTIONS =-Wc,dll -WO,"langlvl(extended),float(ieee)" -D${PLATFORM}
-D_OPEN_THREADS -D_XOPEN_SOURCE_EXTENDED -D_MSE_PROTOS
PLATFORM_COMPILE_OPTIONS2 =-Wc,dll -WO,"langlvl(extended),notempinc,float(ieee)"
-D${PLATFORM} -D_OPEN_THREADS -D_XOPEN_SOURCE_EXTENDED -D_MSE_PROTOS
```

At line 308,

```
# We need the Xerces library
ifeq ($(PLATFORM), OS390)
ALLLIBS += $(XERCESCROOT)/lib/libxerces-c1_5_1.x
else
ALLLIBS += -L$(XERCESCROOT)/lib -lxerces-c1_5_1
endif
```

`xerces-c1_5_1.x` needs to be changed to `libxerces-c1_5.x`.

src/PlatformSupport/ArenaBlock.hpp

There is a known compiler defect causing a compile time error for explicit destructor calls; replace it with the following:

```
theObject.~ObjectType();
```

src/PlatformSupport/DOMStringHelper.cpp

There is a Xalan source coding typo:

```
XALAN_PLATFORMSUPPORT_EXPORT_FUNCTION(XalanDOMString) &
```

should be

```
XALAN_PLATFORMSUPPORT_EXPORT_FUNCTION(XalanDOMString &)
```

These changes are required because the ISO C++ 98 Standard is more strict about where _Export can be specificied.

src/PlatformSupport/DOMStringHelper.hpp

There is a known compiler defect caused by interaction between the `inline` keyword and templates. To avoid this defect, remove the two occurrences of `inline` keyword from the function `void OutputString()`.

There is a Xalan source coding typo:

```
XALAN_PLATFORMSUPPORT_EXPORT_FUNCTION(XalanDOMString) &
```

should be

```
XALAN_PLATFORMSUPPORT_EXPORT_FUNCTION(XalanDOMString &)
```

src/XalanDOM/XalanDOMString.cpp

Comment out the second assert() on XalanDOMSTring::length(), to circumvent a compiler optimization defect.

src/XSLT/StylesheetHandler.cpp

There is a compiler optimization defect that prevents compiling this file at Opt 2. The workaround is to compile only this file at Opt(2),NOINLINE.

Tip: The compiler, like any other application, has certain hard limits built in. For example, the number of symbols it can handle, or the size of code it can handle. In this case, we have reached the limit of the compiler. The solution is to turn down the optimization. As a general rule, the higher the optimization, the more memory and/or time is required.

Here is the error message you would see when the compiler runs out of memory:

```
SEVERE ERROR CCN1002: Virtual storage exceeded.
```

This may be due to you having too little virtual storage available. Consult your system programmer on turning up the amount of virtual storage available to your ID. However, if the maximum amount of virtual storage is already available and the message still occurs, try turning down the optimization.

Changes required in the other required packages

The following changes are only needed because of configuration differences caused by the externals having been built with a previous version of the compiler.

Starting from the extracted directory for ICU:

```
/home/buildid/icu/
```

Starting from the extracted directory for XML4C:

```
/home/buildid/xml4c3_5_1/
```

In xml4c3_5_1 package, include/util/Compilers/MVSCPPDefs.hpp

Comment out #define NO_NATIVE_BOOL, because the z/OS V1R2 compiler has native bool support.

In xml4c3_5_1 package, include/sax/InputSource.hpp

In the InputSource class, move copy constructor InputSource(const InputSource&) from private to protected.

In ICU package, include/unicode/platform.h

Incorrect #define, using old iostream.h

The values below are the configuration for ICU. It tells the ICU which version of iostream is supported by the compiler. 198508 refers to the old iostream.h type of iostream and the 199711 refers to the ISO C++ 1998 Standard compliance iostream.

Change

```
#define U_IOSTREAM_SOURCE 198508
```

to

```
#define U_IOSTREAM_SOURCE 199711
```

Compiler defects

We notified the compiler development team about the defects we discovered in the porting exercise and they will be fixed in the future.

13.2.3 Building the application

Assuming the Xalan source is extracted to /home/buildid/xml-xalan/c:

```
$ export XALANCROOT=/home/buildid/xml-xalan/c
$ export XERCESCROOT=/home/buildid/xml4c3_5_1
$ export ICUROOT=/home/buildid/icu
$ export LIBPATH=$XALANCROOT/lib:$XERCESROOT/lib:$ICUROOT/lib:$LIBPATH
$ export PATH=$XALANCROOT/bin:$PATH
$ unset _CXX_CXXSUFFIX
$ export CXX=c++
$ export CXXFLAGS=-2
$ export _CXX_CCMODE=1
$ export XALAN_USE_ICU=1
$ cd $XALANROOT/src
$ configure --prefix=$XALANROOT
$ export _CXX_CXXSUFFIX=cpp
$ gmake 2>&1 | tee build.logL
```

13.2.4 Binding a symbol name longer than 1024 characters

The z/OS V1R2 binder can only handle symbols shorter than 1024 characters. In our example, we had a few objects that contain symbol names longer than that limit. We had to work around this by using the prelinker/binder combo or #pragma MAP. The binder development has been notified of this limit and it will hopefully be addressed in a future version of the binder.

For the current release (z/OS V1R2) of the binder, PMR number 32579,057,649 is opened for the lack of message output for reaching this limit and APAR OW49602 updates the binder to display a better diagnosis message of this problem.

There are two ways to bind with symbols longer than 1024 characters, and we discuss them here:

Using the prelinker/binder combo

Using a prelinker/binder combo instead of the binder will let you handle a symbol longer than 1024 characters. However, it is only viable if the application does not contain any Generalized Object File Format (GOFF) object file. Since the XPLINK object prerequests GOFF, any XPlinked application cannot use this solution.

To invoke the prelinker/binder combo, set the following environment variables:

```
export _C89_STEPS=-1
export _CC_STEPS=-1
export _CXX_STEPS=-1
```

And make sure the objects are compiled as non-GOFF.

Using #pragma MAP

The #pragma MAP directive tells the compiler to convert all references to an identifier to the mapped name. Only symbol definitions need to be mapped. If there is a long symbol used by the code, the map name has to come from the object file that defines it. (This is to make sure a single map name is used across the whole program.)

Because a template is involved, the #pragma MAP must appear after the template has been defined. So for each CU that contains symbols longer than 1024 characters, we created a header file and #include'd it at the end of the CU's source code.

Tip: If no template is used, we can keep it simple by constructing one common header containing all the #pragma MAPs for the whole application. Note that a #pragma map is allowed in a compilation unit (CU) that neither uses nor defines the symbol; it is simply ignored. This is key to making a common header possible.

First we need to find out what symbols are too long and need the mapping. We wrote the following script to make our task easier.

Example 13-1 nmlength script

```
#!/bin/sh

# Usage:
#   nmlength objectfilename
# this will list all the symbols name in the object file
if [ $# = 0 ]; then
  echo USAGE: `basename $0` objectfilename
  echo ""
  echo "Output format:"
  echo "  symbol_name_length object_filename: symbol_name"
else
  nm -A $1 | awk '{if ($3 == "T" && length($4) > 1024) print length($4)" "$1" "$4}'
  # Uncomment the following line to list the object file containing long symbols
  # nm -A $1 | awk '{if ($3 == "T" && length($4) > 1024) print $1}'
fi
```

Using this script on an object file, it will list all the symbols that are longer than 1024 characters. These are the ones we needd to use a #pragma MAP to workaround the binder limit:

```
$for i in *.o; do; nmlength $i; done
```

Here is an example output; we only show part of one symbol:

```
1059 /c390/tools/kl/xalan/lotusxsl-c-src_1_2/obj/CountersTable.o:
_Construct__3stdHPQ3_3std9_Tree_nodXTQ2_3std12_Tmap_traitsXTPC10ElemNumberTQ2...
```

This will list all the object files with symbols longer than 1024 characters in the current directory. From this, we can construct the #pragma MAP header file that renames the long symbol name to “CountersTable_1”.

Example 13-2 OS390_CountersTable_mapping

```
#ifndef OS390_COUNTERSTABLE_MAPPING_H
#define OS390_COUNTERSTABLE_MAPPING_H
// file: CountersTable.o
// map name CountersTable

#pragma map( std::_Construct<std::_Tree_nod<std::_Tmap_traits<const ElemNumbe\
r*,std::vector<Counter,std::allocator<Counter> >,std::less<const\
 ElemNumber*>,std::allocator<std::pair<const ElemNumber* const,s\
td::vector<Counter,std::allocator<Counter> > > >,0> >::_Node*,st\
d::_Tree_nod<std::_Tmap_traits<const ElemNumber*,std::vector<Cou\
nter,std::allocator<Counter> >,std::less<const ElemNumber*>,std:\
:allocator<std::pair<const ElemNumber* const,std::vector<Counter\
,std::allocator<Counter> > > >,0> >::_Node*>(std::_Tree_nod<std:\
:_Tmap_traits<const ElemNumber*,std::vector<Counter,std::allocat\
or<Counter> >,std::less<const ElemNumber*>,std::allocator<std::p\
air<const ElemNumber* const,std::vector<Counter,std::allocator<C\
ounter> > > >,0> >::_Node**,std::_Tree_nod<std::_Tmap_traits<con\
st ElemNumber*,std::vector<Counter,std::allocator<Counter> >,std\
::less<const ElemNumber*>,std::allocator<std::pair<const ElemNum\
ber* const,std::vector<Counter,std::allocator<Counter> > > >,0> \
>::_Node* const&)\
,"CountersTable_1")
#endif /* OS390_COUNTERSTABLE_MAPPING_H */
```

This is an example of one long symbol being mapped. Let's name this file OS390_CountersTable_mapping.h.

Now we need to include this file in the CU containing the long symbol names, which is CountersTable.cpp. Then we have to repeat the mapping procedure for all the other parts containing long symbol name. Here is a list of all the object files that have longer than 1024 character symbol names in Xalan:

- CountersTable.o
- KeyTable.o
- NamespacesHandler.o
- NodeSorter.o
- Stylesheet.o
- StylesheetExecutionContextDefault.o
- XPathEnvSupportDefault.o
- XTokenNumberAdapterAllocator.o
- XTokenStringAdapterAllocator.o
- XalanSourceTreeParserLiaison.o
- XalanSourceTreeProcessingInstructionAllocator.o
- XalanTransformer.o
- XercesDocumentBridge.o

We have to include the map after the definition of the templates, and since the #pragma must appear after the template has been defined, we will #include the mapping header file at the end of the source code.

Then we rebuild all the objects and we will be able to bind the long symbol name correctly. (The configure step can be skipped.)

13.2.5 Executing an application that calls the C++ Standard Library

Since the C++ Standard Library is built as an XPLINKed DLL, the following environment variable must be set:

```
_CEE_RUNOPTS=XPLINK(ON)
```

We can test our application by cd to ../bin and execute:

```
$ testXSLT -?
```

And we can try the sample XML code that comes in the package:

```
$ testXSLT -in ../samples/SimpleTransform/foo.xml -xsl \
../samples/SimpleTransform/foo.xsl -out foo.out
```

The generated file foo.out will be in ASCII. Use an ASCII reader to view it. This is the expected output:

```
<?xml version="1.0" encoding="UTF-8"?>
<out>Hello</out>
```

13.3 Conclusion

The porting exercise of Xalan gave us some insight into the difference between the SGI STL and the C++ Standard Library, as well as differences between the OS/390 V2R10 C++ compiler and the z/OS V1R2 C++ compiler.

We have the following recommendations:

- Allocate enough time for porting; the differences between environments can be larger than they seem.
- If possible, start the port from a similar environment. We recommend starting from an AIX variant if available, otherwise some POSIX-compliant UNIX environment.
- When developing a new application intended to be multiplatform, try to follow the published ANSI C/C++ standard as much as possible. This will minimize porting headaches in the future.
- First build the application at Opt(0), test and make sure it’s working before trying higher optimization levels.

A

Sample code

This appendix contains sample DB2 code described in 2.6, “DB2” on page 34:

- “C code with embedded SQL” on page 146 contains the C source code and embedded SQL statements.
- “Precompile REXX EXEC” on page 154 contains the Precompile EXEC which was originally contained in the redbook *Porting Applications to the OpenEdition MVS Platform*, GG24-4473.

C code with embedded SQL

```
/*********************************************************************/
/***MVS & OpenEdition MVS dependentcies see line 18*******************/
/***DB2 dependcies                      see line 182 and 187***********/
/*********************************************************************/
/* Include external definitions                                      */
/*********************************************************************/
#pragma linkage(DSNALI, OS)
#pragma linkage(DSNTIAR, OS)
/*********************************************************************/
/* Include C library definitions                                     */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*********************************************************************/
/* General declarations                                              */
/*********************************************************************/
#define OMVS     1 /* ***value 0 for MVS, value 1 for OpenEdition MVS */
#define NOTFOUND 100
/*********************************************************************/
/* Input / Output files                                              */
/*********************************************************************/
FILE *cardin;                     /* Input control cards             */
FILE *report;                     /* Output phone report             */
/*********************************************************************/
/* Input record structure                                            */
/*********************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
struct {
  char action[2];                 /* L for list or U for update  */
  char lname[16];                 /* last name or pattern- L mode*/
  char fname[13];                 /* first name or pattern-L mode*/
  char eno[7];                    /* employee number- U mode     */
  char newno[5];                  /* new phone number- U mode    */
       } ioarea;
char trail[43];                   /* unused portion of input rec */
char slname[16];                  /* unmodified last name pattern*/
EXEC SQL END DECLARE SECTION;
/*********************************************************************/
/* Report headings                                                   */
/*********************************************************************/
struct {
  char hdr011[30];
  char hdr012[32];
       } hdr0 = {
                      "    REQUEST  LAST NAME        ",
                      "FIRST NAME    EMPNO   NEW XT.NO"};
#define rpthdr0 hdr0.hdr011
struct {
  char hdr111[29];
  char hdr112[21];
  char hdr113[30];
       } hdr1 = {
                      "-----------------------------",
                      " TELEPHONE DIRECTORY ",
                      "-----------------------------"};
#define rpthdr1 hdr1.hdr111
struct {
  char hdr211[10];
  char hdr212[11];
  char hdr213[ 8];
  char hdr214[ 6];
  char hdr215[ 9];
  char hdr216[ 5];
  char hdr217[ 5];
  char hdr221[ 7];
  char hdr222[ 7];
  char hdr223[ 5];
  char hdr224[ 5];
  char hdr225[ 5];
       } hdr2 = {
                    "LAST NAME",
                    "FIRST NAME",
                    "INITIAL",
                    "PHONE",
                    "EMPLOYEE",
                    "WORK",
                    "WORK",
                    "NUMBER",
                    "NUMBER",
                    "DEPT",
                    "DEPT",
                    "NAME"
                };
#define rpthdr2 hdr2.hdr211,hdr2.hdr212,hdr2.hdr213,hdr2.hdr214,\
                hdr2.hdr215,hdr2.hdr216,hdr2.hdr217,hdr2.hdr221,\
                hdr2.hdr222,hdr2.hdr223,hdr2.hdr224,hdr2.hdr225
/**********************************************************************/
/* Report formats                                                     */
/**********************************************************************/
static char fmt1[] = "\n %s\n";
static char fmt2[] = " %9s%17s%10s%6s%10s%5s%5s\n%43s%7s%7s%5s%5s\n";
static char fmt3[] = " %-16s%-16s%-5s%-7s%-9s%-5s%-36s\n";
static char fmt4[] = " %1c%15c%12c%6c%4c%43c";
static char fmt5[] =
   "\n\n %s\n %s\n   --%-7s--%-15s--%-12s--%-5s--%-9s--\n";
/**********************************************************************/
/* Fields sent to message routine                                     */
/**********************************************************************/
char outmsg[70];                       /* error/information msg buffer*/
char module[ 8] = "DSN8BD3";           /* module name for message rtn */
/* extern DSN8MDG(); *?                /* message routine             */
/**********************************************************************/
/* SQL communication area                                             */
/**********************************************************************/
EXEC SQL INCLUDE SQLCA;
/**********************************************************************/
/* SQL declaration for view VPHONE                                    */
/**********************************************************************/
EXEC SQL DECLARE       DSN8610.VPHONE TABLE
       (LASTNAME       VARCHAR(15)  NOT NULL,
        FIRSTNAME      VARCHAR(12)  NOT NULL,
        MIDDLEINITIAL     CHAR( 1)  NOT NULL,
        PHONENUMBER       CHAR( 4)          ,
        EMPLOYEENUMBER    CHAR( 6)  NOT NULL,
        DEPTNUMBER        CHAR( 3)  NOT NULL,
        DEPTNAME       VARCHAR(36)  NOT NULL);
/**********************************************************************/
```

```
/* Structure for pphone record                                       */
/*********************************************************************/
/* Note: since the sample program data does not contain imbedded      */
/*       nulls, the C language null terminated string can be used to  */
/*       receive the varchar fields from DB2.                         */
EXEC SQL BEGIN DECLARE SECTION;
struct {
  char lastname[16];
  char firstname[13];
  char middleinitial[2];
  char phonenumber[5];
  char employeenumber[7];
  char deptnumber[4];
  char deptname[37];
  } pphone;
EXEC SQL END   DECLARE SECTION;
/*********************************************************************/
/* SQL declaration for view VEMPLP (used for update processing)       */
/*********************************************************************/
EXEC SQL DECLARE DSN8610.VEMPLP TABLE
       (EMPLOYEENUMBER    CHAR( 6)  NOT NULL,
        PHONENUMBER       CHAR( 4)          );
/*********************************************************************/
/* Structure for pemplp record                                       */
/*********************************************************************/
EXEC SQL BEGIN DECLARE SECTION;
struct {
  char employeenumber[7];
  char phonenumber[5];
       } pemplp;
EXEC SQL END   DECLARE SECTION;
/*********************************************************************/
/* SQL cursors                                                       */
/*********************************************************************/
/* cursor to list all employee names */
EXEC SQL DECLARE TELE1 CURSOR FOR
         SELECT *
         FROM DSN8610.VPHONE;
/* cursor to list all employee names with a pattern */
/* (%) or (_) in last name                          */
EXEC SQL DECLARE TELE2 CURSOR FOR
         SELECT *
         FROM DSN8610.VPHONE
         WHERE LASTNAME LIKE :lname;
/* cursor to list all employees with a specific last name */
EXEC SQL DECLARE TELE3 CURSOR FOR
         SELECT *
         FROM DSN8610.VPHONE
         WHERE LASTNAME = :slname
           AND FIRSTNAME LIKE :fname;
/*********************************************************************/
/* SQL return code handling                                          */
/*********************************************************************/
EXEC SQL WHENEVER SQLERROR   GOTO DBERROR;
EXEC SQL WHENEVER SQLWARNING GOTO DBERROR;
EXEC SQL WHENEVER NOT FOUND  CONTINUE;
/* Start main routine************************************************ */
extern main()
{
/* ***CAF Start *** ************************************************* */
```

```
long int fnret;
char     opnfunc[13] = "OPEN         ";
char     clsfunc[13] = "CLOSE        ";
char     plan[9]     = "DB2CPLAN ";    /* ***name of DB2 PLAN***      */
char     term_opt[5] = "SYNC";
long int return_code;
long int reason_code;
/* extern char     DB2SSID[];        */
char     DB2SSID[4]  = "DBS3";         /* ***name of DB2 subsystem*** */
char     result_msg[64] = "***result_msg***";
int      *rpc_fault_status;
int      r_f_s;
/* ***CAF End*** ***************************************************** */
/* ***CAF Start Open the plans connections with DB2****************** */
printf("main start\n");
rpc_fault_status = &r_f_s;
fnret = DSNALI(opnfunc,
               DB2SSID,
               plan,
               &return_code,
               &reason_code);
printf("Open request fnret = %d\n", fnret);
printf("Open request return_code = %d\n", return_code);
printf("Open request reason_code = %d\n", reason_code);
fflush(NULL);
if (fnret != 0)
{
 sprintf(result_msg,
         "*** Unable to open DB2 for return: *",
         "%ld REASON: %ld",
         return_code,
         reason_code);
 *rpc_fault_status = 1;
 printf("main end\n");
 return;
}
/* ***CAF End******************************************************** */
  /* Open the input and output files */
  if (OMVS == 0)
      cardin = fopen("DD:CARDIN","r");
      else
      cardin = fopen("cardin","r");
  if (cardin == NULL)
      {
       printf("Open error cardin\n");
       exit(16);
      }
  if (OMVS == 0)
      report = fopen("DD:REPORT","w");
      else
      report = fopen("report","w");
  if (report == NULL)
      {
       printf("Open error report\n");
       exit(16);
      }
  /* While more input, process */
  while (!feof(cardin))
  {
    /* Read the next request */
    if (fscanf(cardin, fmt4,
                         ioarea.action,
                         ioarea.lname,
                         ioarea.fname,
                         ioarea.eno,
                         ioarea.newno,
                         trail) == 6)
    {
      /* Display the request */
      DSN8MDG(module, "000I", outmsg);
      fprintf(report, fmt5,
                           outmsg,
                           rpthdr0,
                           ioarea.action,
                           ioarea.lname,
                           ioarea.fname,
                           ioarea.eno,
                           ioarea.newno);
      Do_req();
    }
  } /* endwhile */
  fclose(report);
/* ***CAF Start Close the plans connections with DB2**************** */
fnret = DSNALI(clsfunc,
               term_opt,
               &return_code,
               &reason_code);
printf("Close request fnret = %d\n", fnret);
printf("Close request return_code = %d\n", return_code);
printf("Close request reason_code = %d\n", reason_code);
fflush(NULL);
if (fnret != 0)
{
 sprintf(result_msg,
         "*** Unable to close DB2 for return: *",
         "%ld REASON: %ld",
         return_code,
         reason_code);
 *rpc_fault_status = 1;
 printf("main end\n");
 return;
}
/* ***CAF End*** **************************************************** */
 printf("main end\n");
} /* end main */
/*******************************************************************/
/* Process the current request                                     */
/*******************************************************************/
Do_req()
{
  char *blankloc;                     /* string translation pointer  */
  strcpy(slname, ioarea.lname);       /* save untranslated last name */
  while (blankloc = strpbrk(ioarea.lname, " "))
    *blankloc = '%';                  /* translate blanks into %     */
  while (blankloc = strpbrk(ioarea.fname, " "))
    *blankloc = '%';                  /* translate blanks into %     */
  /* Determine request type */
  switch (ioarea.action[0])
  {
    /* Process LIST request */
```

```
case 'L':
  /* Print the report headings */
  fprintf(report, fmt1, rpthdr1);
  fprintf(report, fmt2, rpthdr2);
  /* List all employees */
  if (!strcmp(slname,"*              ")){
    EXEC SQL OPEN TELE1;
    EXEC SQL FETCH TELE1 INTO :pphone;
    if (sqlca.sqlcode == NOTFOUND){          /* If no employees   */
      DSN8MDG(module, "008I", outmsg);       /* found, display    */
      fprintf(report, " %s\n", outmsg);      /* error message     */
    } /* endif */
    while (sqlca.sqlcode == 0){
      Prt_row();
      EXEC SQL FETCH TELE1 INTO :pphone;
    } /* endwhile */
    EXEC SQL CLOSE TELE1;
    /* List generic employees */
  } else {
    if (strpbrk(slname, "%")) {
      EXEC SQL OPEN TELE2;
      EXEC SQL FETCH TELE2 INTO :pphone;
      if (sqlca.sqlcode == NOTFOUND){        /* If no employees   */
        DSN8MDG(module, "008I", outmsg);     /* found, display    */
        fprintf(report, " %s\n", outmsg);    /* error message     */
      } else {
        while (sqlca.sqlcode == 0){
          Prt_row();
          EXEC SQL FETCH TELE2 INTO :pphone;
        } /* endwhile */
      } /* endif */
      EXEC SQL CLOSE TELE2;
      /* List specific employee */
    } else {
      EXEC SQL OPEN TELE3;
      EXEC SQL FETCH TELE3 INTO :pphone;
      if (sqlca.sqlcode == NOTFOUND){        /* If no employee    */
        DSN8MDG(module, "008I", outmsg);     /* found, display    */
        fprintf(report, " %s\n", outmsg);    /* error message     */
      } else {
        while (sqlca.sqlcode == 0){
          Prt_row();
          EXEC SQL FETCH TELE3 INTO :pphone;
        } /* endwhile */
      } /* endif */
      EXEC SQL CLOSE TELE3;
    } /* endif */
  } /* endif */
  break; /* end of 'L' request */
/* Update an employee phone number */
case 'U':
  EXEC SQL UPDATE DSN8610.VEMPLP
           SET PHONENUMBER = :ioarea.newno
           WHERE EMPLOYEENUMBER = :ioarea.eno;
  if (sqlca.sqlcode == 0){                   /* If employee       */
    DSN8MDG(module, "004I", outmsg);         /* updated, display  */
    fprintf(report, " %s\n", outmsg);        /* confirmation msg  */
  } else {
    DSN8MDG(module, "007E", outmsg);         /* otherwise, display*/
    fprintf(report, " %s\n", outmsg);        /* error message     */
```

```
      } /* endif */
      break;
    /* Invalid request type */
    default:
      DSN8MDG(module, "068E", outmsg);           /* Display error msg */
      fprintf(report, " %s\n", outmsg);
  } /* endswitch */
  return;
DBERROR:
  Sql_err();
} /* end Do_req */
/*********************************************************************/
/* Print a single employee on the report                             */
/*********************************************************************/
Prt_row()
{
  fprintf(report, fmt3, pphone.lastname,
                        pphone.firstname,
                        pphone.middleinitial,
                        pphone.phonenumber,
                        pphone.employeenumber,
                        pphone.deptnumber,
                        pphone.deptname);
}
/*********************************************************************/
/* SQL error handler                                                 */
/*********************************************************************/
Sql_err() {
#define data_len 120
#define data_dim 8
struct error_struct {
  short int error_len;
  char error_text[data_dim][data_len];
  } error_message = {data_dim * data_len};
extern short int DSNTIAR(struct sqlca        *sqlca,
                         struct error_struct *msg,
                         int                 *len);
short int rc;
int i;
static int lrecl = data_len;
  DSN8MDG(module, "060E", outmsg);
  fprintf(report, " %s %i\n", outmsg, sqlca.sqlcode);
  rc = DSNTIAR(&sqlca, &error_message, &lrecl);  /* Format the sqlca  */
  if (rc == 0){                                  /* Print formatted   */
    for (i=0;i<=7;i++){                          /*   sqlca           */
      fprintf(report, "%.120s\n", error_message.error_text  [i]);
    } /* endfor */
  } else {
    DSN8MDG(module, "075E", outmsg);
    fprintf(report, " %s %hi\n", outmsg, rc);
  } /* endif */
  /* Attempt to rollback any work already done */
  EXEC SQL WHENEVER SQLERROR   CONTINUE;
  EXEC SQL WHENEVER SQLWARNING CONTINUE;
  EXEC SQL WHENEVER NOT FOUND  CONTINUE;
  EXEC SQL ROLLBACK;
  if (sqlca.sqlcode == 0){                       /* If rollback       */
    DSN8MDG(module, "053I", outmsg);             /* completed, display*/
    fprintf(report, " %s\n", outmsg);            /* confirmation msg  */
  } else {                                       /* otherwise, display*/
```

```
    DSN8MDG(module, "061E", outmsg);             /* error message     */
    fprintf(report, " %s %i\n", outmsg, sqlca.sqlcode);
  } /* endif */
  fclose(report);
  exit(0);
} /* end of Sql_err */
/* End main routine************************************************** */
/* Start error routine************************************************ */
DSN8MDG(module, icode, outmsg)
char outmsg[70];
char module[8];
char icode[5];
{
/*********************************************************************/
/* Miscellaneous declarations                                        */
/*********************************************************************/
#define NumMsgs 9
register int i;
char mnf[51] = "MESSAGE TEXT NOT FOUND                            ";
/*********************************************************************/
/* Message code array                                                */
/*********************************************************************/
char code[NumMsgs][5] = {
          "000I",    "004I",    "007E",    "008I",
          "053I",    "060E",    "061E",
          "068E",    "075E"};
/*********************************************************************/
/* Message text array                                                */
/*********************************************************************/
char text[NumMsgs][51] = {
"REQUEST IS:"                                      ,/*000I*/
"EMPLOYEE SUCCESSFULLY UPDATED"                    ,/*004I*/
"EMPLOYEE DOES NOT EXIST, UPDATE NOT DONE"         ,/*007E*/
"NO EMPLOYEE FOUND IN TABLE"                       ,/*008I*/
                                      /* GENERAL INFO. MESSAGES */
"ROLLBACK SUCCESSFUL, ALL UPDATES REMOVED"         ,/*053I*/
                                      /* GENERAL ERROR MESSAGES */
"SQL ERROR, RETURN CODE IS:"                       ,/*060E*/
"ROLLBACK FAILED, RETURN CODE IS:"                 ,/*061E*/
"INVALID REQUEST, SHOULD BE 'L' OR 'U'"            ,/*068E*/
"MESSAGE FORMAT ROUTINE ERROR, RETURN CODE IS:"    }/*075E*/;
/*********************************************************************/
/* Main program routine                                              */
/*********************************************************************/
/* Scan message table (0 thru NumMsgs - 1) for requested message */
for (i=0; (i<NumMsgs) && !(strcmp(icode,code[i])==0); i++){}
/* Found message code */
if (i<NumMsgs){
  strcpy(outmsg, "DSN8");
  strcat(outmsg, icode);
  strcat(outmsg, "   ");
  strcat(outmsg, module);
  strcat(outmsg, "-");
  strcat(outmsg, text[i]);
/* Unknown message code */
} else {
  strcpy(outmsg, "DSN8");
  strcat(outmsg, icode);
  strcat(outmsg, "   ");
  strcat(outmsg, module);
  strcat(outmsg, "-");
  strcat(outmsg, mnf);
} /* endif */
return;                               /* return to caller           */
} /* dsn8mpg */
/* End error routine*** ********************************************* */
```

Precompile REXX EXEC

```
/* ***REXX Start***************************************************** */
/* Exec Name:    PRECOMP                                              */
/*                                                                    */
/* Description:  Runs the DB2 precompiler.  This exec will copy       */
/*               C source from the hierarchical file system           */
/*               and place it into a temporary MVS data set for       */
/*               processing by the DB2 precompiler.  The output       */
/*               from the DB2 precompiler (modified C source) will    */
/*               be copied back into the hierarchical file system.    */
/*                                                                    */
/*               Optionally run BIND to bind to a plan.               */
/*                                                                    */
/*               This exec can be run from the OpenEdition MVS shell. */
/*                                                                    */
/* Invocation:   DSNHPC, BIND                                         */
/*                                                                    */
/* Variables :   DB2 subsystem name                                   */
/*               High level qualifier of DB2 data sets                */
/*                                                                    */
/* Input parms:                                                       */
/*                                                                    */
/* in=            (required)  HFS INPUT FILE TO DB2 PRECOMPILER       */
/* ou=            (required)  HFS OUTPUT FILE FROM DB2 PRECOMPILER    */
/*                                                                    */
/* pa=            (optional)  PATH FOR BOTH INPUT and OUTPUT          */
/* pl=            (optional)  NAME OF PLAN IF YOU WANT TO BIND        */
/*                                                                    */
/* Examples:                                                          */
/*Run DB2 precompile but don't bind it to a plan                      */
/*   precomp pa=/u/user/ in=db2cpgm2.i ou=db2cpgm2.c                  */
/*                                                                    */
/*Run DB2 precompile and bind it to a plan called db2cplan            */
/*   precomp pa=/u/user/ in=db2cpgm2.i ou=db2cpgm2.c                  */
/*           pl=db2cplan                                              */
/*                                                                    */
/*Run DB2 precompile & bind and have different input and output paths */
/*   precomp in=/u/user/db2cpgm2.i ou=/u/kjellaa/db2cpgm2.c           */
/*           pl=db2cplan                                              */
/*                                                                    */
/*                                                                    */
/* ***SET TSO ENVIRONMENT******************************************** */
ADDRESS TSO
/* ***FETCH REXX PARAMETER LIST************************************** */
parse arg p.1  p.2  p.3  p.4  p.5
say 'PRECOMP started'
/* ***free and alloc sysprint and systerm*************************** */
'free  ddname(sysprint)'
'alloc ddname(sysprint) da(*)'
'free  ddname(systerm)'
```

```
'alloc ddname(systerm) da(*)'
/* ***Pick up TSO userid********************************************** */
uid = sysvar(SYSUID)                               /* ***TSO UID  *** */
/*                                                                    */
/* *** Define SYSTEM variables ****************************************/
SY = 'SYSTEM(DBS3)'        /*Name of DB2 subsystem*/
db2hlq = 'DB2V61S3'        /*High level qualifier of DB2 data sets*/
/*                                                                    */
/* ***Define data set names******************************************* */
syslibd = ''''||db2hlq'.SRCLIB.DATA'||''''         /*****DB2********* */
dbrm    = ''''||db2hlq'.DBRMLIB.DATA'||''''        /* ***PLAN DSN *** */
preinp  = ''''||UID'.DB2.PREINP'||''''             /* ***WORK FILE*** */
preout  = ''''||UID'.DB2.PREOUT'||''''             /* ***WORK FILE*** */
/* ***Define REXX variables******************************************* */
pathi = ''
patho = ''
pathx = ''
plan  = ''
rch   = 0
rcx   = 0
/* ***CHECK P.1 THRU P.5 ******************************************** */
do i = 1 TO 5 BY 1
   if p.i <> '' then
      do
      say 'p.'||i||'='||p.i
      xxx = substr(p.i,1,3)
      p.i = xxx||substr(p.i,4)
      if substr(p.i,1,3) = 'pl=' then
         plan  = substr(p.i,4); else
      if substr(p.i,1,3) = 'in=' then
         pathi = substr(p.i,4); else
      if substr(p.i,1,3) = 'ou=' then
         patho = substr(p.i,4); else
      if substr(p.i,1,3) = 'pa=' then
         pathx = substr(p.i,4); else
         do
         say 'INVALID VALUE. PARAMETER #'||i||'='||p.i
         say 'PRECOMP ended with RC = 16'
         return 16
         end
      end
end
/* ***CONVERT TO UPPER CASE****************************************** */
UPPER PLAN
/* ***TEST INPUT PARAMETER COMBINATIONS****************************** */
  if (pathi = '' |  patho = '') then
   do
   say 'INPUT FILENAME, OUTPUT FILENAME NOT SPECIFIED'
   say 'PRECOMP ended with RC = 16'
   return 16
   end
if pathx <> '' & (pathi = '' | pathi = '') then
   do
   say 'PATH SPECIFIED, BUT INPUT AND OUTPUT FILE NOT SPECIFIED'
   say 'PRECOMP ended with RC = 16'
   return 16
   end
if pathx <> '' then
   do
   L = length(pathx)
   if substr(pathx,1,1) <> '/' | substr(pathx,L,1) <> '/' then
      do
      say 'PATH SPECIFIED, BUT DOES NOT START AND END WITH /'
      say 'PRECOMP ended with RC = 16'
      return 16
      end
   end
if pathx <> '' & (pos('/',pathi) <> 0 | pos('/',patho) <> 0) then
   do
   say 'PATH SPECIFIED, BUT INPUT OR OUTPUT FILE NAME CONTAINS /'
   say 'PRECOMP ended with RC = 16'
   return 16
   end
if (pathi =  patho) then
   do
   say 'PRECOMPILE, BUT SAME PATH NAME FOR INPUT AND OUTPUT'
   say 'PRECOMP ended with RC = 16'
   return 16
   end
if plan <> '' then
   do
   plen = length(plan)
      if plen > 8 then
      do
      say 'PLAN NAME IS TOO LONG, MUST BE 8 CHARACTERS OR LESS'
      say 'PRECOMP ended with RC = 16'
      return 16
      end
   end
/* ***CREATE SOME VARIABLES***************************************** */
if pathx <> '' then
   pathi = pathx||pathi
if pathi <> '' then
   do
   pathi = ''''||pathi||''''                      /* ***HFS INP  *** */
   say 'pathi='||pathi
   end
if pathx <> '' then
   patho = pathx||patho
if patho <> '' then
   do
   patho = ''''||patho||''''                      /* ***HFS OUT  *** */
   say 'patho='||patho
   end
/* ***DELETE AND ALLOCATE MVS DATA SETS***************************** */
   do
   if sysdsn(preinp) = 'OK' then
      "delete ("preinp") nonvsam"
   if sysdsn(preout) = 'OK' then
      "delete ("preout") nonvsam"
   end
/* ***ALLOCATE WORK FILES******************************************* */
   do
     if sysdsn(preinp) <> 'OK' then
        'alloc fi(preinp) da('preinp') unit(sysallda) new reuse,
         space(16,16) tracks recfm(f b) lrecl(80)'; else
        'alloc fi(preinp) da('preinp') recfm(f b) lrecl(80) shr reuse'
     if sysdsn(preout) <> 'OK' then
       'alloc fi(preout) da('preout') unit(sysallda) new reuse,
        space(16,16) tracks recfm(f b) lrecl(80)'; else
       'alloc fi(preout) da('preout') recfm(f b) lrecl(80) shr reuse'
    end
/* ***ALLOCATE HFS FILES********************************************* */
   do
    'allocate ddname(hfsinp) path('pathi') pathopts(ordwr,ocreat)',
             'pathmode(sirusr,siwusr)'
   end
/* ***OCOPY FROM OMVS TO MVS***************************************** */
say 'OCOPY C input source from HFS'
   do
   say 'OCOPY DB2 PRECOMPILE'
   'ocopy indd(hfsinp) outdd(preinp) text convert(no)'
   call setrch
   say 'OCOPY ended with RC =' rcx
   'free  ddname(preinp)'
   if rcx > 0 then
      return rcx
   end
/* ***FREE INPUT FILES********************************************** */
  'free ddname(hfsinp)'
/* ***executing the DB2 precompiler********************************* */
if plan = '' then
   do
   plan = 'DUMMY'       /*dummy name for plan to get through precomp  */
   end
   do
   dbrmm  = ''''||db2hlq'.DBRMLIB.DATA('PLAN')'||''''
   'alloc ddname(sysin)  da('preinp')    shr reuse'
   'alloc ddname(syscin)  da('preout')   shr reuse'
   'alloc ddname(syslib)  da('syslibd')  shr reuse'
   'alloc ddname(dbrmlib) da('dbrmm')    shr reuse'
   'alloc fi(sysut1)  unit(vio) new reuse space(16 16) track'
   'alloc fi(sysut2)  unit(vio) new reuse space(16 16) track'
   PROG = "'X.Y(DSNHPC)'"
   PARM = "'HOST(C),MARGINS(1,80),NOOPTIONS,FLAG(W),ATTACH(CAF)'"
   call pgmcall  prog parm
   call setrch
   say 'DB2 precompiler ended with RC =' rcx
   'free  ddname(sysin)'
   'free  ddname(syscin)'
   'free  ddname(syslib)'
   'free  ddname(dbrmlib)'
   if rcx > 4 then
      return rcx
end
/* ***executing the DB2 DSN BIND command**************************** */
if plan <> 'DUMMY' then
   do
  'alloc ddname(dbrmlib) da('dbrm') shr reuse'
   queue 'BIND PLAN('PLAN') MEMBER('PLAN') ACT(REP) ISOLATION(CS) FLAG(W)'
   queue 'END'
  'DSN' SY
   call setrch
   say 'DB2 DSN BIND ended with RC =' RC
  'free  ddname(dbrmlib)'
   if rcx > 0 then
      return rcx
   end
/* ***executing OCOPY to copy translated file to hfs**************** */
if patho <> ''  then
```

```
   do
      say 'OCOPY C translated source to HFS'
     'free  ddname(preout)'
     'alloc ddname(mvsinp) da('preout') shr reuse'
     'alloc ddname(hfsout) path('patho') pathopts(ordwr,ocreat)',
          'pathmode(sirusr,siwusr)'
     'ocopy indd(mvsinp) outdd(hfsout) text convert(no)'
      call setrch
      say 'OCOPY ended with RC =' rcx
     'free  ddname(mvsinp)'
     'free  ddname(hfsout)'
      if rcx > 0 then
         return rcx
   end
SAY 'PRECOMP ended with highest RC =' rch
RETURN RCH
/* ***pgmcall subroutine for calling a program with a parm field***** */
pgmcall:
parse arg pgm a
trace o
a = strip(a,"B","'")
p = length(a)
interpret "pl = '"d2x(length(a))"'x"
pl = overlay(pl,'0000'x,3-length(pl)) || a
parse var pgm "(" pg ")"
ADDRESS "LINKPGM" pg  "pl"
return RC
/* ***setrc subroutine for keeping highest return code*************** */
setrch:
rcx = rc
if rcx > rch then
   rch = rcx
return
/* ***REXX End****************************************************** */
```

B

Comparison of z/OS and GNU compilers and make tools

This appendix contains comparisons of commonly used compiler options between the GNU gcc and the z/OS C/C++ compilers, and a comparison between OS/390 and GNU make.

GNU gcc vs. OS/390 C/C++ compiler options

The gcc option list comes from the gcc Web page:

`http://gcc.gnu.org/onlinedocs/gcc_toc.html`

Table B-1 Comparison of GNU and OS/390 C/C++ compiler options

gcc	OS/390	Remark
-c	-c	Compile source files, but do not link.
-S	N/A	Stop at output of assembler source.
-E	-E	Stop after the preprocessing stage.
-o file	-o file	Write executable to “file”.
-v	-V	Verbose output. On OS/390, the option is upper case V.
-pipe	N/A	On OS/390, try the MEMORY option, which keeps temporary files in memory rather than in a physical file; for example: `c89 -Wc,memory hw.c`
-help	N/A	On OS/390, invoking the compiler without any arguments will display the help; for example: `c++`
-x language	N/A	This option specifies the language used. On OS/390, use the `-+` option to force the compiler to compile in C++; for example: `c++ -+ hw.c` Without the `-+` option, the `c++` utility determines the language type from the suffix. In the example, the small .c suffix would cause a C compilation without the `-+` option.
-ansi	-Wc,”lang(ansi)”	Support all ANSI standard C programs.
-fno-asm	N/A	Do not recognize asm, inline, or typeof as a keyword.
-fno-builtin	N/A	Do not recognize built-in functions that do not begin with two leading underscores.
-fhosted	N/A	Compile for a hosted environment.
-ffreestanding	N/A	Compile for a freestanding environment. For OS/390, refer to *OS/390 C/C++ Programming Guide, Using the System Programming C Facilities*.
-trigraphs	N/A	Support ANSI C trigraphs. On OS/390, trigraphs are always supported.
-traditional	N/A	Support some aspects of traditional C compilers. The OS/390 cc utility supports K&R C. But since K&R is not a strict standard, different compilers supporting K&R do not behave exactly the same. Try cc if the application you are porting from uses gcc -traditional.

gcc	OS/390	Remark
-traditional-cpp	N/A	Support some aspects of traditional C compilers. On OS/390, use: `cc -E`
-fcond-mismatch	N/A	
-funsigned-char -fsigned-char	N/A	Use #pragma chars to specify the sign of plain char.
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	N/A	On OS/390 the default bitfield type is unsigned int.
-fwritable-strings	-Wc,norostring	Also #pragma strings(writable).
-fallow-single-precision	N/A	

GNU make vs. OS/390 make

Many aspects of the GNU make and OS/390 make commands are similar; however, the differences are significant enough that often makefiles written for GNU make will not successfully build a program or package using OS/390 make. There may be more differences, but Table B-2 shows some of those commonly encountered.

Table B-2 Comparison of GNU and OS/390 make constructs

Construct or variable	GNU make	OS/390 make
$<	The automatic variable `$<' is just the first prerequisite.	Runtime macro which, in inference rules, evaluates to the single prerequisite that caused the execution of the rule. In normal rules, it evaluates to the same as $?.
$&	Not supported.	The list of prerequisites that are newer than the target.
VPATH	Variable that allows you to specify a list of directories that make should search.	Not supported.

These differences can be illustrated using the following simple makefile:

```
$ cat makefile
OBJS = foo.o bar.o
foo: $(OBJS)
        echo "all prerequisites = $&"
        echo "prerequisites that caused execution of the rule = $<"
        echo "prerequisites newer than the target = $?"
```

A useful description of GNU make can be found on the Web at:

http://apache.btv.ibm.com/gnu/make.html

C

OS/390 C/C++ compiler ASCII support

The native character encoding on OS390 is EBCDIC. In order to enhance interoperability with other UNIX platforms, ASCII support is provided in both the compiler and the runtime library.

An example of such requirements is a client/server application where the server is ported to OS/390 while the client remains on the workstation using ASCII. The server application needs to process the ASCII messages coming from and going to the client.

Roughly speaking, there are two areas you need to examine to plan a port that needs ASCII support: the *literals* inside the program, and the *runtime* library.

To maintain maximum portability, a program should avoid having string and character literals inside instruction statements. Such literals should be defined as data, either in a header or (preferably) in a table of message catalog loaded during runtime. This way the source code itself is *character encoding-neutral*.

Character encoding-neutral means the program should avoid the following two coding styles:

Avoid literals inside code

Following is an example of having string literal inside code:

```
{
   ...
   if (!strcmp(cmmd, “send”)) {
   /* send stuffs */
}
else {
if (!strcmp(cmmd,”receive”)) {
   /* receive stuffs */
}
else {
if (!strcmp(cmmd,”quit”)) {
   /* closedown */
}
   ...
}
```

A better way to do this is to put the string literals in a header:

```
cmmd.h
#if CMMD_OWNER
   char *cmmd_Send=”send”;
   char *cmmd_Receive=”receive”;
   char *cmmd_Quit=”quit”;
#else
   extern char *cmmd_Send;
   extern char *cmmd_Receive;
   extern char *cmmd_Quit;
#endif
```

And change the source to use the `cmmd_xxx` variables in the strcmp.

This way, if your application needs to support other national languages, translations need only be done in selected source files. You can also load such literals from a table at runtime.

Similar considerations apply for messages. Using message catalogs is a good way to manage this.

Avoid logic that relies on specific character encoding

Another coding style the program should avoid is using algorithms that rely on the actual encoding of the characters (for example, checking for upper case/lower case by checking whether the character is less than a, or assuming the numerics are encoded less than the letters).

Use library functions to do the job. Note that often the compiler uses built-ins and macros under the cover for these functions if such algorithms exist; in combination with optimization, the end result is probably the same as the program using these character encoding algorithms explicitly.

Another example of this is the output from lex and yacc, where octal escape sequences are hardcoded into the program code. You cannot move source source code from other platforms (if it uses ASCII) to OS/390; you have to regenerate the source using lex and yacc.

The following describes the facilities available in the compiler and in the library.

Compile option and pragma

The CONVLIT compile phase option controls the encoding of character and string literals in the program. You can specify a codepage in the option. The usual codepage used for ASCII is ISO8859-1. For example:

```
c89 -Wc,”convlit(iso8859-1)” hello.c
```

All the string and character literals in the source program will be translated to ASCII.

Note: You can do the same by using an obsolete feature, defining the reserved macro using the -D compiler option: `-D__STRING_CODE_SET__=ISO9959-1`.

The convlit example presented above only affects the normal literals; it has no effect on the wide literals (for example, L"wide string").

Refer to *OS/390 C/C++ User's Guide*, SC09-2361 for more detailed information.

The #pragma convlit can suspend the effect of the CONVLIT option. It can be used to provide both ASCII and EBCDIC literals in the same program. For example:

```
hello.c
char *p1="User error: "; /* in ascii */

#pragma convlit(suspend)
char *s="%s%s\n"; /* in ebcdic */
#pragma convlit(resume)

char *p2="This message is sent to an ascii client."; /* in ascii */

void main() {
   printf(s, p1, p2);
}
```

Compile it with:

```
c89 -Wc,CONVLIT(ISO8859-1)" hello.c
```

The two messages `p1` and `p2` are in ASCII, while the `printf` format string `s` is in EBCDIC.

Note that you can accomplish the same by using the iconv function. The only difference is that iconv performs the conversion during runtime instead of compile time.

Runtime library

The preceding example leads us to the issue of the runtime library. Since the library is in EBCDIC, the formatting specifier we pass into printf must be in EBCDIC. Examples of such specifiers are %, *, and the numerals that go with them.

Strategically placing `#pragma convlit` in the source may give you the right encoding for the specifiers, but it may not be convenient, or sometimes even possible, to do so. An alternative to this is to use an ASCII version of the library.

ASCII support of selected functions in the runtime is provided by a library called libascii. Before V2R8, you can download a copy of the libascii from:

http://www-1.ibm.com/servers/eserver/zseries/zos/unix/libascii.html

For V2R8 and later releases, the libascii functions are shipped as part of the Language Environment.

The libascii package provides an ASCII interface layer for selected library functions. libascii supports ASCII input and output characters by performing the necessary iconv() translations before and after invoking the C/C++ runtime library functions. Note that not all C functions are supported. Refer to the preceding HTML for further information about using the library.

D

STLPort

Before z/OS V1R2, the C++ compiler did not provide a Standard Template Library (STL). To satisfy customer requirements, a free downloadable STL ported by SGI was used. It is often referred to as the STLPort.

The original SGI STL does not work as is with the OS/390 compilers because of the ANSI C++ language issues. SGI STLPort is an adapted version of SGI STL implementation that is compatible with OS/390 compilers.To obtain a copy of the port, go to the following Web site and follow the links:

`http://www.ibm.com/software/ad/c390/cmvsstlp.htm`

Downloading and extracting STLPort

- Download the tar file stlport.tar.gz.
- Uncompress the tar file using **gunzip**:

```
$ gunzip stlport.tar.gz
```

- Unwind the tar file with the following command:

```
$ pax -o from=ISO8859-1,to=IBM-1047 -rf stlport.tar
```

The **pax** command is used so that we can specify code page conversion. The tar file was created in ASCII. The -o option specifies the from/to code pages during the expand operation.

The configuration header for OS/390 is stl_ibm.h under the config subdirectory.

Using STLPort

The header files are located under the stlport and stlport/old_hp directories. Add these paths to the `-I` option of your compilation. Following is an example (adapted from the test samples in the STLPort download):

```
vectest.cpp:
#include <iostream.h>
```

```
#include <vector>
#include <algorithm>

int main (int, char**)
{
  cout<<"Results of vec6_test:"<<endl;
  int array [] = { 1, 4, 9, 16, 25, 36 };

  vector<int> v(array, array + 6);
  int i;
  for(i = 0; i < v.size(); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;

  v.erase(v.begin()); // Erase first element.
  for(i = 0; i < v.size(); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;

  v.erase(v.end() - 1); // Erase last element.
  for(i = 0; i < v.size(); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;

  v.erase(v.begin() + 1, v.end() - 1); // Erase all but first and last.
  for(i = 0; i < v.size(); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;

  // push something back
  v.push_back(100);
  v.push_back(121);
  v.push_back(144);

  for(i = 0; i < v.size(); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;

  // use random to shuffle it
  random_shuffle (v.begin(), v.end());
  for(i = 0; i < v.size(); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;

  return 0;
}
```

Compile as follows (assume the stlport directory is installed under /usr/local):

```
c++ -Wc,"lang(extended)" -I/usr/local/stlport vectest.cpp
```

If you use the old headers, e.g. `vector.h` instead or `vector`, include also the stlport/old_hp include path:

```
c++ -Wc,"lang(extended)" -I/usr/local/stlport -I/usr/local/stlport/old_hp vectest.cpp
```

Run:

```
a.out
```

This produces the following output:

```
Results of vec6_test:
v[0] = 1
v[1] = 4
v[2] = 9
v[3] = 16
v[4] = 25
v[5] = 36

v[0] = 4
v[1] = 9
v[2] = 16
v[3] = 25
v[4] = 36

v[0] = 4
v[1] = 9
v[2] = 16
v[3] = 25

v[0] = 4
v[1] = 25

v[0] = 4
v[1] = 25
v[2] = 100
v[3] = 121
v[4] = 144

v[0] = 144
v[1] = 121
v[2] = 4
v[3] = 100
v[4] = 25
```

Use the tempinc option to control the directory for template instantiation files.

The port provides full support to all the algorithms and containers specified in the Standard. However, namespaces are not supported and exception handling has not been tested on OS/390.

Include a path for new.h and iostream.h

STLport 3.21 assumes both new.h and iostream.h are in the same path given in the macro STL_NATIVE_INCLUDE_PATH. On OS/390, however, they could be installed under different directories.

If you get a compilation error saying these header files cannot be opened, add the following to stl_ibm.h (under the config subdirectory of the download):

```
#ifdef __MVS__
#undef __STL_NATIVE_INCLUDE_PATH
#define __STL_NATIVE_INCLUDE_PATH /usr/lpp/ioclib/include
#undef __STL_NATIVE_C_INCLUDE_PATH
#define __STL_NATIVE_C_INCLUDE_PATH /usr/include
#endif
```

Then modify the header `stlport/new` near line 37 (**bold** hightlights below):

```
# if !defined (__STL_NO_NEW_NEW_HEADER)
#   if defined (__GNUC__) && (__GNUC_MINOR__ >= 8 )
#    include <../include/new>
#   else
#    include __STL_NATIVE_HEADER(new)
#   endif
#  else
#   ifdef __MVS__
#     include __STL_NATIVE_C_HEADER(new.h)
#   else
#      include __STL_NATIVE_HEADER(new.h)
#   endif
# endif
```

And modify the header `stlport/new.h` near line 31:

```
# if defined (__GNUC__) && (__GNUC_MINOR__ >= 8 )
#   include <../include/new.h>
# elif defined (__BORLANDC__)
#  include
# else
#  ifdef __MVS__
#    include __STL_NATIVE_C_HEADER(new.h)
#  else
#     include __STL_NATIVE_HEADER(new.h)
#  endif
# endif
```

Changes to stlcomp.h

In version 3.00 of the STL, on line 186 in file stlcomp.h, there should be *two* underscores (__) before MVS__ and not one. That is, the line should be:

```
(defined(__MVS__) && ...
```

and not:

```
(defined(_MVS__) && ...
```

This has been corrected in version 3.01.

In Version 3.01 of the STL, uncomment line 143 in file stlcomp.h:

```
#  define  __STL_NONTEMPL_BASE_MATCH_BUG 1
```

Long long support

OS/390 C++ started supporting long long in release 6. The compiler option langlvl(extended) is needed to enable this feature. To instruct the STLPort not to use long long as a native type, modify stl_ibm.h (under the config subdirectory); near line 78 add the **bold** highlighted code:

```
#if __IBMCPP__ >= 350  && !__MVS__
#  define __STL_LONG_LONG 1
#endif
```

Explicit template notation

OS/390 C/C++ requires explicit template notation such as `template_class<Param>` where most other compilers only accept `template_class`.

Note the following example:

```
template <class Param> class template_class {
template_class foo(); // error for OS390 C/C++
template_class<Param> foo(); // OK
...
};
```

The STLport is compatible with the OS/390 C/C++ requirement, but may cause compatibility problems when porting template code from other platforms.

Extern “C”

OS/390 C++ distinguishes between C and C++ function linkages. If a C++ function parameter expects a C++ function pointer, you cannot pass a C function pointer to the function. C function linkage is specified by the `extern` **“**`C`**”** keyword. For example:

```
template <class Result>
pointer_to_void_function<Result> ptr_gen(Result (*x)());
p = ptr_gen(rand);// error for OS/390 C/C++
```

In this template, `ptr_gen` takes a function pointer as its argument and returns a function pointer adaptor (a type of function object), and `rand` is an `extern` **“**`C`**”** pointer pointing the a C library function. This will cause a compile time error complaining about the function linkage.

To work around this problem, provide a C++ wrapper around the `extern` **“**`C`**”** function and pass the C++ wrapper instead:

```
int cxxrand(void) { return rand();}
p = ptr_gen(cxxrand); // OK
```

Function return types

In OS/390 C++, function return types cannot be function types or array types:

```
template <class InputIterator, class OutputIterator, class result>
   OutputIterator adjacent_difference(InputIterator first,
   InputIterator last,
   OutputIterator result);

int main() {
   int number[10];
   int different[5];
   ...
   adjacent_difference(number,
      number + 5,
      different); // error for OS/390 C/C++

...
}
```

In the preceding example, the compiler attempts to create an instantiation that returns an array int[], which is not allowed in OS/390 C/C++:

```
int[] adjacent_difference(int*, int*,  int*)
```

To work around this problem, cast the `int` array to an `int` pointer:

```
adjacent_difference(number,number + 5, (int *) different); // OK
```

E

Dumps

This appendix deals with a subject that's probably the last thing you want to read about (dumps)—and you should probably only refer to this section when other means of troubleshooting fail. Much more robust tools (like the debug tool and the remote debugger) are available on OS/390, and most of us would prefer to stay away from dumps as much as possible. But there are times when reading dumps gives you the most information with a minimum effort, if you know where to look.

This is especially true when doing a port. After all, when porting, the subject application is already running on the source platform; therefore, it is less likely (comparatively speaking) that there are application logic errors--which are best flushed out using a debugger.

In a port, the errors that you are likely to encounter are idiosyncrasies of the individual platforms (for example, freeing a pointer twice might cause an abend). If an error does cause the program to abend, a quick check in the dump is sometimes all you need to find out what's gone wrong and where it happened.

This would save you the trouble of having to set up a debug environment to reproduce the error, then invoke the debugger to trace through the program, only to find out you may not have a clue where to put the first breakpoint.

This appendix attempts to help you to find out where to set the first breakpoint to jumpstart the troubleshooting process. We do not go into details of the different kinds of dumps on OS/390. Instead, we give a quick tour of the dump you are likely to encounter when porting applications under OS/390 UNIX.

The material is self-contained and does not require prior knowledge about OS/390 system internals. You should be able to use this appendix without referring to several different manuals.

CEEDUMP

In this section, we provide information about how to run CEEDUMP.

The traceback

Run the following program under the OS/390 UNIX shell and you will get a dump:

```
void main() {
   int *p = 0;
   *p = 1; /* abend here */
}
```

Use the compiler option GONUM to get statement numbers in the traceback (we will explain what is a traceback shortly):

```
$ c89 -Wc,"GONUM" hello.c
```

Before running the program, delete all files with names prefixed by CEEDUMP; then run a.out:

```
$ rm CEEDUMP*
R a.out
```

You should get the following messages:

```
CEE3204S The system detected a protection exception (System Completion Code=0C4).
From compile unit main at entry point main at statement 4 at compile unit offset
+00000054 at address 0A7DF634.
...
```

The program a.out abends because it dereferences a null pointer. The memory pointed to by the pointer (address zero) is not accessible by the program and the operation causes a *protection exception*. This exception is indicated by the System Completion Code 0C4, which is an interrupt raised by the hardware.

This interrupt goes through different layers in the operating system and eventually gets to the Language Environment, which terminates the program and produces a *CEEDUMP*. The dump is in a file with the following name:

```
CEEDUMP.yyyymmdd.hhmmss.pppppppp
```

where yyyymmdd and hhmmss are the date and time, and pppppppp is a number to make the file name unique.

Figure 13-1 shows the first part of the dump:

```
CEE3DMP V2 R9.0: Condition processing resulted in the unhandled condition.        07/12/00 11:15:54 AM              Page:    1

Information for enclave main

  Information for thread 0AC9E10000000000

  Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offset   Statement  Load Mod  Service  Status
    00079718  CEEHDSP       05705570  +0000302E  CEEHDSP       05705570  +0000302E              CEEPLPKA  UQ39550  Call
    000791E0  main          0A7DF5E0  +00000054  main          0A7DF5E0  +00000054          4  *PATHNAM            Exception
    000790C8                0558F4AE  +000000B4  EDCZMINV      0558F4AE  +000000B4              CEEEV003           Call
    00079018  CEEBBEXT      0000D138  +0000013C  CEEBBEXT      0000D138  +0000013C              CEEBINIT           Call
```

Figure 13-1 Beginning of a CEEDUMP

Of interest to us is the *traceback* starting at line 7. This is printed from selected information on the call stack at the time of the abend. Each line represents a function call.

Within each line, the columns give various information about the function and/or the call. The 5th column Entry is the entry point (name) of the function. The 8th column Statement is the source line at which the function call takes place.

For our purposes, we can logically divide the system into the following layers: the hardware; the operating system (MVS); the Language Environment (LE); the program. LE maintains a consistent environment for all the languages. All user programs run on top of LE; all system service requests go through LE, which in turn calls MVS services, if necessary.

Note: A normal hosted C program runs on top of LE. A freestanding C program can be created using System Programing Facilities. Refer to the *OS/390 C/C++ Programming Guide,* SC09-2362 for details.

The CEEDUMP is printed by LE. There are other dumps, more detailed and low-level, that are printed by MVS. Refer to *OS/390 MVS Diagnosis: Tools and Service Aids*, SY28-1085 for information on getting a dump. Refer to *OS/390 MVS Diagnosis: Procedures*, SY28-1082 for information on diagnosing the dump.

Referring again to the traceback shown in Figure 13-1 on page 174, the last line is the first entry on the stack. In our example, this is entry point CEEBBEXT from load module CEEBINIT (under column 9), and is the LE initialization routine which setups the LE environment for the program.

The next-to-last line, entry point EDCZMINV from load module CEEEV003, initializes the C runtime. EDCZMINV in turn calls main, the third from last line in the traceback. This is where our program begins. If there are other function calls in the call chain, there will be more lines stacking on top of main.

Column 8 Statement shows the source line number within the function where the call takes place; the function being called is represented by the next line up. In our case, main makes a call at statement 4 to CEEHDSP. This is really an exception, not a normal call, as indicated under column 11 Status. Statement 4 is where our program abends and CEEHDSP is the LE handler that handles the exception.

Note that the statement number is shown on the traceback only if the compiler option GONUM is turned on.

A feature worth mentioning here is the Service column (column 10). This comes from the compiler option SERIVCE. For example:

```
c89 -Wc,"gonum service(my_rel_1)" hw.c
```

The text `my_rel_1` will be printed under the Service column against the functions in hw.c in the traceback. This is intended for indicating build releases and versions, and could be useful when you are servicing your product.

Types of abends

There are two types of abends: system abends handled by MVS, and user abends handled by the “user” application (in this context, a user application refers to everything on top of MVS, including LE).

System abends are indicated by system completion code `Snnn`. The common system abends are:

0C1	Invalid instruction. Program might have branched into the data area.
0C4	Protection exception. Cannot access the address. (The address exists, but is protected.)
0C5	Invalid Address. Address does not exist.
0C6	Invalid operation. Similar to 0C1; the op code might be valid but the operations implied by the combination of operands probably not.
0C7	Invalid data. Usually misaligned packed decimal data.
222	Operator killed the job.
322	CPU time limit exceeded.
806	Program module not found.
80A	Region size limit exceeded.
B37	Disk full.
E37	File full.

Refer to *OS/390 MVS Messages and Codes* for details.

LE abends are problems detected and caught by the Language Environment. They are indicated by complete codes U4000 to U4095. Refer to *OS/390 V2R9 Language Environment for OS/390*, GC28-1945 and *VM Debugging Guide and Run-Time Messages*, SC28-1942 for more information.

The rest of the CEEDUMP

The preceding information should provide sufficient information to give you a starting point for debugging the program. For the adventurous, let us go through the rest of the CEEDUMP. If you have read dumps on other systems before, you can probably figure out most of the contents by the titles and remarks contained in the dump.

Following the traceback is the section Condition Information for Active Routines. *Active routines* are functions on the stack at the time of the exception; listed here are the condition and registers of the function running at the time of the exception, `main` in our case.

The *Machine State*, in particular, gives the contents of the program status word (PSW) and the general purpose registers (GPR0-15) at the time of the abend. The second word of the PSW is the address of the next instruction to be executed. As a linkage convention, GPR15 is always the entry point address of the current function; GPR14 is the return address. Note that the address space is 31-bit; the high-order bit in registers is not used to form the address. Therefore, the number `8A7DF610` in GPR15, for example, really means address `0A7DF610`.

The next section is Parameters, Registers, and Variables for Active Routines. This provides additional information for each function on the traceback, starting with the first line in the traceback (CEEHDSP, in our example). Each function is shown in a subsection, which lists the general purpose registers at the time when control passes to the next function. It also dumps the memory around the address pointed to by the registers.

Pseudo-assembly listing

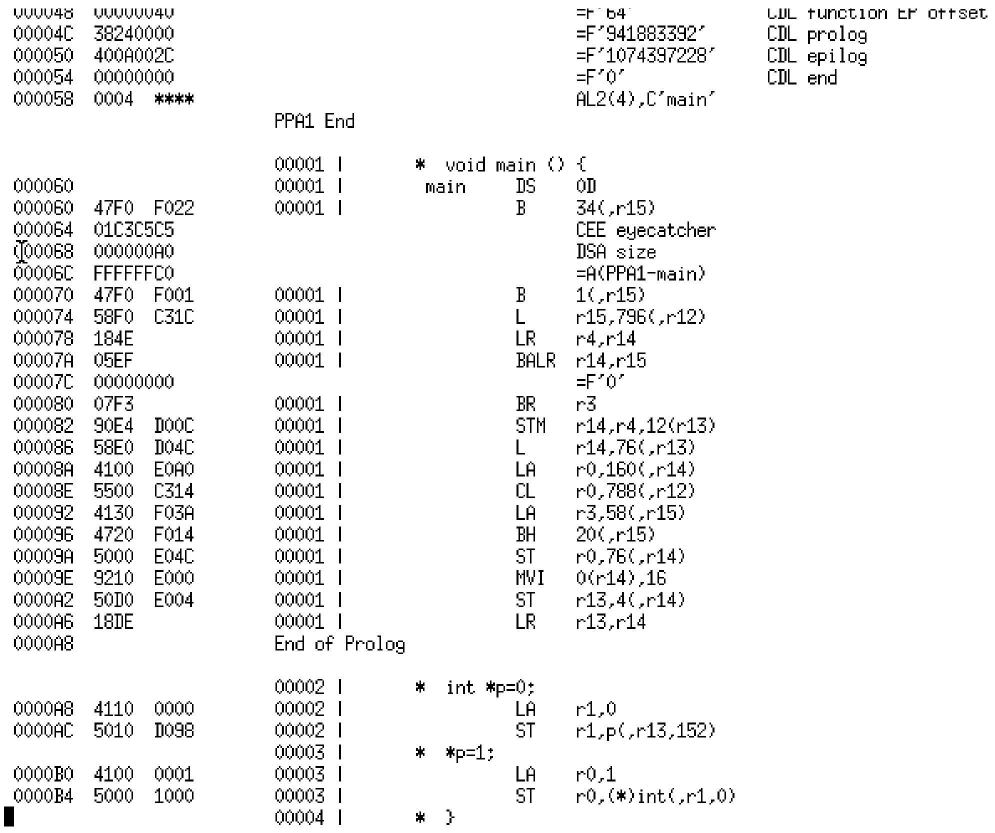
To conclude this appendix, let us briefly go through the pseudo-assembly listing.

You get the pseudo-assembly listing by turning on the LIST compiler option. The SOURCE option is usually used in conjunction with it. For example:

```
$ c89 -Wc,SOURCE,LIST hw.c >hw.lst
```

The output is directed to stdout.

Using the code example in “The traceback” on page 174, we get the pseudo-assembler listing shown in Figure 13-2.



```
000048  00000040                                       =F'64'           CDL function EP offset
00004C  38240000                                       =F'941883392'    CDL prolog
000050  400A002C                                       =F'1074397228'   CDL epilog
000054  00000000                                       =F'0'            CDL end
000058  0004  ****                                     AL2(4),C'main'
                         PPA1 End

                         00001 |       *  void main () {
000060                   00001 |        main     DS    0D
000060  47F0  F022       00001 |                 B     34(,r15)
000064  01C3C5C5                                       CEE eyecatcher
000068  000000A0                                       DSA size
00006C  FFFFFFC0                                       =A(PPA1-main)
000070  47F0  F001       00001 |                 B     1(,r15)
000074  58F0  C31C       00001 |                 L     r15,796(,r12)
000078  184E             00001 |                 LR    r4,r14
00007A  05EF             00001 |                 BALR  r14,r15
00007C  00000000                                       =F'0'
000080  07F3             00001 |                 BR    r3
000082  90E4  D00C       00001 |                 STM   r14,r4,12(r13)
000086  58E0  D04C       00001 |                 L     r14,76(,r13)
00008A  4100  E0A0       00001 |                 LA    r0,160(,r14)
00008E  5500  C314       00001 |                 CL    r0,788(,r12)
000092  4130  F03A       00001 |                 LA    r3,58(,r15)
000096  4720  F014       00001 |                 BH    20(,r15)
00009A  5000  E04C       00001 |                 ST    r0,76(,r14)
00009E  9210  E000       00001 |                 MVI   0(r14),16
0000A2  50D0  E004       00001 |                 ST    r13,4(,r14)
0000A6  18DE             00001 |                 LR    r13,r14
0000A8                   End of Prolog

                         00002 |       *  int *p=0;
0000A8  4110  0000       00002 |                 LA    r1,0
0000AC  5010  D098       00002 |                 ST    r1,p(,r13,152)
                         00003 |       *  *p=1;
0000B0  4100  0001       00003 |                 LA    r0,1
0000B4  5000  1000       00003 |                 ST    r0,(*)int(,r1,0)
                         00004 |       *  }
```

Figure 13-2 Pseudo-assembly listing

In the traceback, columns 6 and 7 are the function’s entry point address (E Addr) and the offset from the entry point where the call happens (E Offset). The called function is on the next line up in the traceback. Let us relate the E Addr and *E* Offset to the assembly listing.

The beginning of the assembly listing is signified by the text:

```
P S E U D O   A S S E M B L Y   L I S T I N G
```

The beginning of a function is signified by:

```
PPA1: Entry Point Constants
```

PPA1 is a control block giving information about a function. One control block is built for each function. The executable code starts right after the end of PPA1 (signified by the text `PPA1 End`).

On the far left of the listing is the OFFSET column. This is the “offset” from the entry point of the function. The OBJECT CODE column shows the binary code of the instruction at that location. All would be easy if the offset here is the same as the `E offset` in the traceback. Unfortunately, they are offsets from different stating points. This is best illustrated by going through our example.

Go back to the CEE dump we obtained before, and look up the section Parameters, Registers, and Variables for Active Routines. Look for main's subsection and read out the memory locations around GPR15.

GPR15 is the entry point address of the function. The first word is 47F0 F022 (for a C function). This is a branch instruction to jump over the eye-catcher and other data. On the listing, this is the first word right after the end of PPA1, the `PPA1 END` marker.

The "offset" here in the listing corresponds to the entry point address of the function (GPR15). If you compile with OSV2R9, the offset for this location on the listing is 000060. When cross-referencing the dump with the listing, use this as the synchronization point to align the offsets. The rest are just linear displacements.

F

Performance analyzer output

We followed the instruction of the performance analyzer to get the trace file of a test run on the net.TABLE server. We then FTP'd the trace file to an NT workstation and analyzed the trace file using the GUI tool provided. The result is shown in the next few diagrams.

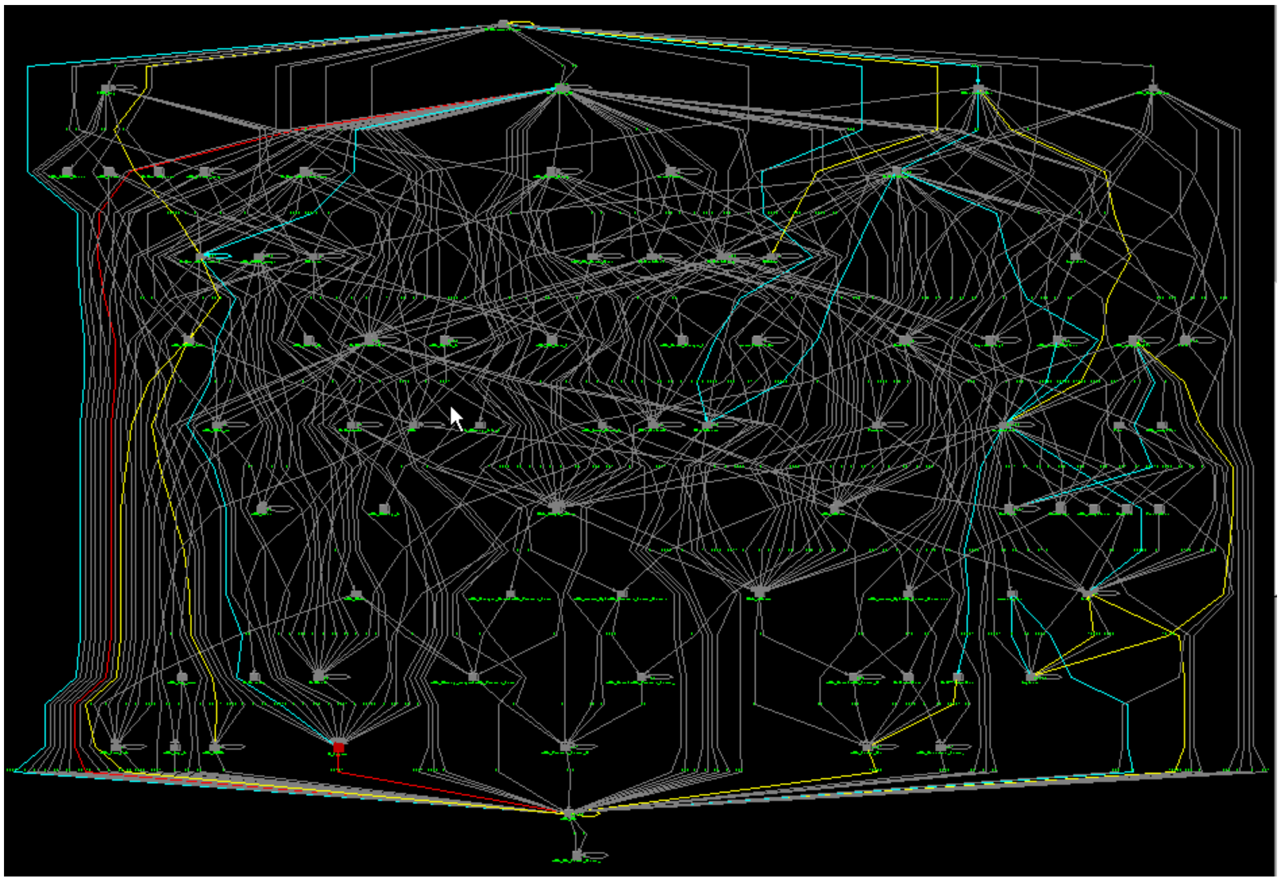


Figure 13-3 Net.TABLES server - Class dynamic call graph

We first obtained an overall picture of the application by displaying the dynamic call graph of the classes; refer to Figure 13-3. The nodes are classes, and a directed arc connecting two nodes represents member function calls.

The diagram is color-coded; highest frequency calls are represented by red. We zoomed into the red spot near the bottom left of the diagram, which led us to Figure 13-4.

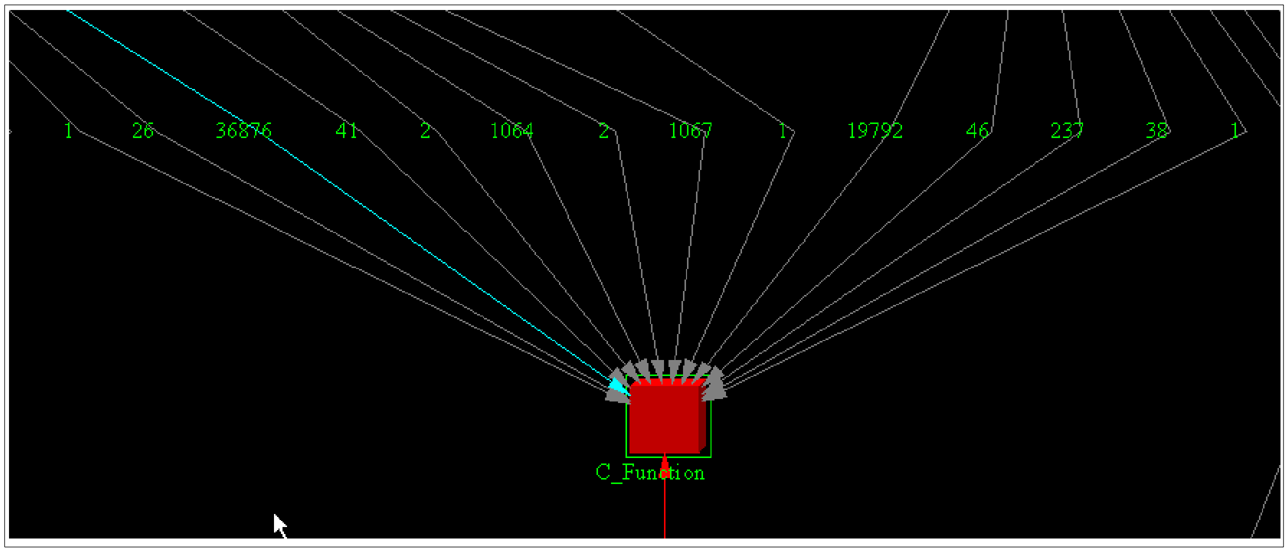


Figure 13-4 Performance analyzer showing a "hot spot"

The *hot spot* turned out to be a C function. Since this diagram is by Class, all C functions are grouped under one node labeled C_Functions.

We could drill into it to find out which function it represents (an alternate view of the call graph is by function.) But before we did that, we traced along the high frequency arcs to get an idea of which classes are invoking these functions.

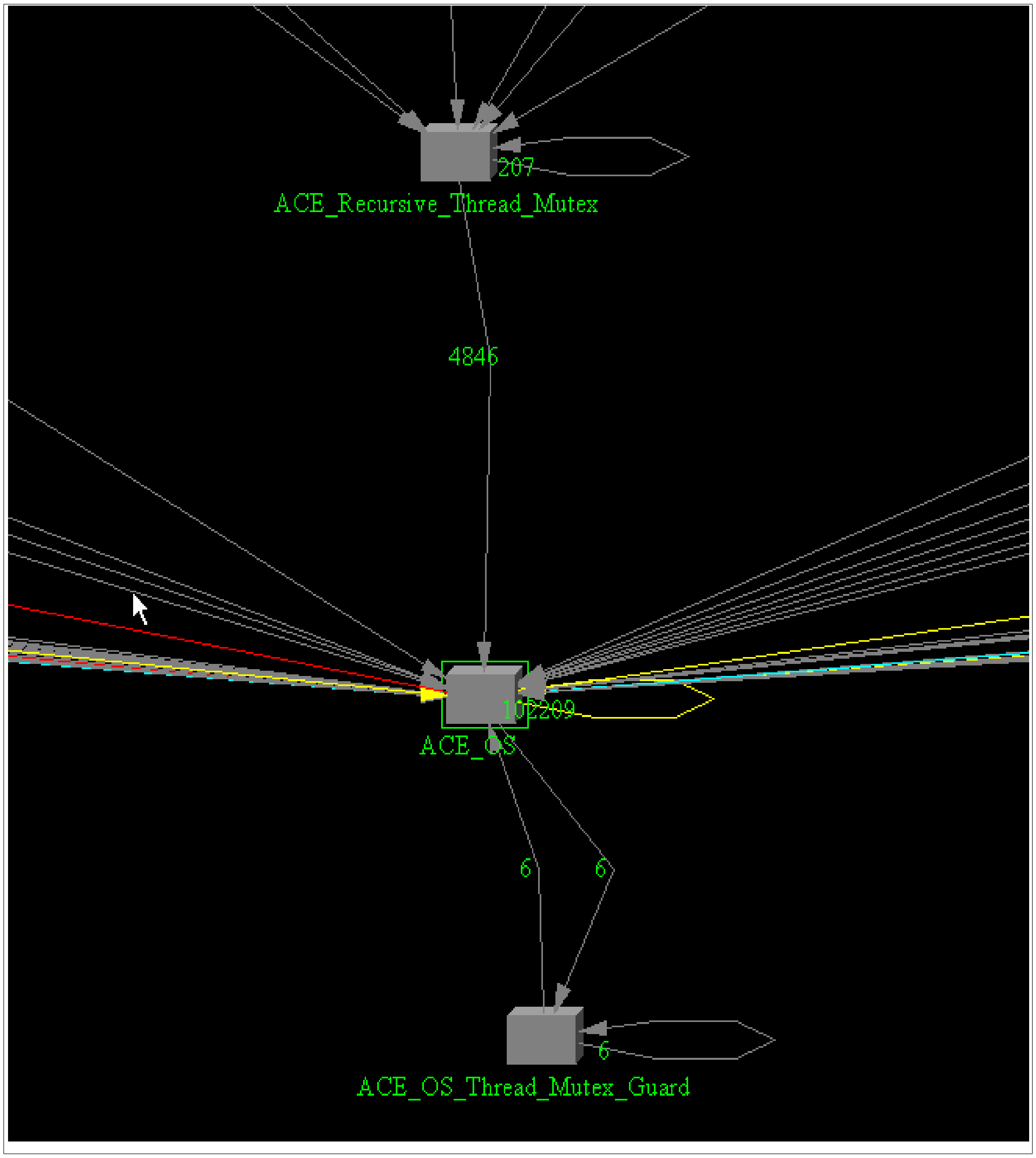
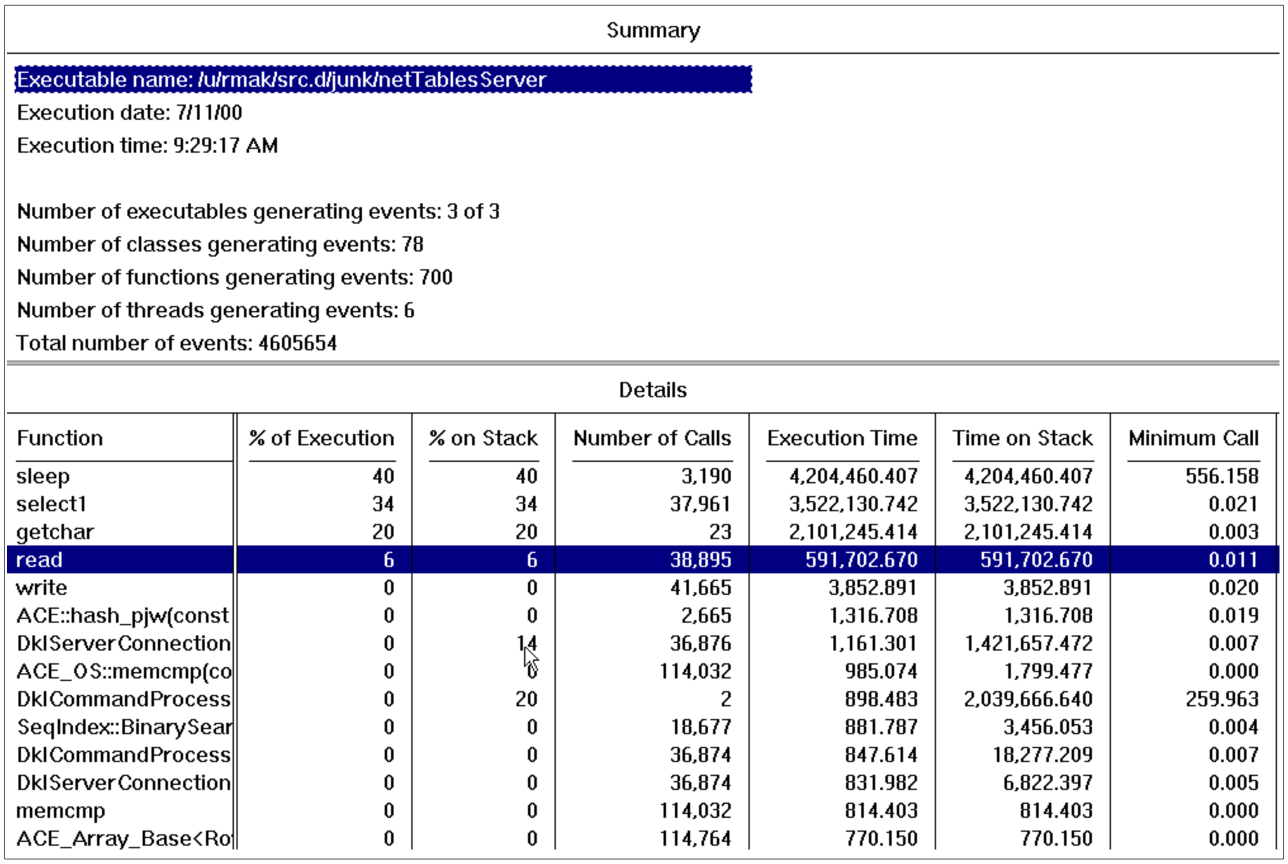


Figure 13-5 Navigating the call graph

We navigated through the call graph by clicking on an arc and then jumped to the caller. (The GUI interface provided the usual facilities.) An example of such navigation is shown in Figure 13-5. Further analysis revealed the read function was the bottleneck.



Summary

Executable name: /u/rmak/src.d/junk/netTablesServer
Execution date: 7/11/00
Execution time: 9:29:17 AM

Number of executables generating events: 3 of 3
Number of classes generating events: 78
Number of functions generating events: 700
Number of threads generating events: 6
Total number of events: 4605654

Details

Function	% of Execution	% on Stack	Number of Calls	Execution Time	Time on Stack	Minimum Call
sleep	40	40	3,190	4,204,460.407	4,204,460.407	556.158
select1	34	34	37,961	3,522,130.742	3,522,130.742	0.021
getchar	20	20	23	2,101,245.414	2,101,245.414	0.003
read	6	6	38,895	591,702.670	591,702.670	0.011
write	0	0	41,665	3,852.891	3,852.891	0.020
ACE::hash_pjw(const	0	0	2,665	1,316.708	1,316.708	0.019
DklServerConnection	0	14	36,876	1,161.301	1,421,657.472	0.007
ACE_OS::memcmp(co	0	0	114,032	985.074	1,799.477	0.000
DklCommandProcess	0	20	2	898.483	2,039,666.640	259.963
SeqIndex::BinarySear	0	0	18,677	881.787	3,456.053	0.004
DklCommandProcess	0	0	36,874	847.614	18,277.209	0.007
DklServerConnection	0	0	36,874	831.982	6,822.397	0.005
memcmp	0	0	114,032	814.403	814.403	0.000
ACE_Array_Base<Ro	0	0	114,764	770.150	770.150	0.000

Figure 13-6 Call Frequency statistics

An alternate view of the data is the call frequency table, as shown in Figure 13-5. The server was predominately in a sleep mode waiting for requests from clients. Therefore, we are only interested in the activities for looking up and serving table data.

The frequency table is sorted by execution time. The first two items, even though having the most execution time, are basically the wait loop. The third item, getchar, is called during initialization. The fourth and fifth items represent the real bottleneck; these are the read/write from/to the socket when serving a client.

G

Bit field

The bit field C construct is not portable. Therefore, if your application uses bit fields and requires them to align the same on different platforms, you need to examine these fields carefully and you may need to change the source code. An example of this is when the structure is the header of a communication protocol.

In the following section, we discuss a few points drawn from our own experience, presented here for reference, and make recommendations:

- Let us call a sequence of adjacent bit fields a *bit field group*. Try to start a bit field group on the *natural* boundary of the machine. In this context, the term “natural boundary” means the address/memory locations that are most efficiently accessed; it is usually a word.
- End a bit field group at the end of a word; if this is not possible, end it at the end of a byte. Put in padding bit fields explicitly and avoid having the compiler to do padding for you. This way, you have better control of how the bit fields are allocated.
- If the compiler provides options to pack the struct (for example to byte, 2-byte, or word boundaries), pack it to byte boundary. When porting the program to another platform, ask for the same packing. This applies not just to bit fields, but to structure members in general. Use dummy members to explicitly align the member, if necessary.
- If a bit field is the last member of a structure, compilers may sometimes insert padding. Avoid having bit field the last member. If it has to be the last member, try to end it on a natural boundary (i.e. explicitly add padding).

A bit field program

Use the following program to determine differences in bit field alignments on different platforms. The comments of the program explain what it does.

Just compile and run the program. If the compiler has different options for structure packing and/or alignments, try out all the option settings and compare the output. The tests are for reference only and do not show all aspects of bit fields allocations. You can add your own tests.

```
$ cat bit.c

#include <stdio.h>
#include <string.h>

/*
   Following are tests to show the bit field alignment. Compile and run the
   program to print the struct bitmaps. Compare the bitmaps from different
   platforms to find differences. Note that these tests are not exhausitive.

   You can add your own tests. Checkout the comment below (before showIt_4)
   for an explanation.
*/

/*
  Alignment tests
  ---------------

  Crossing access-unit boundary.

  A byte is usually the smallest unit of memory individually
  addressable. Bit fields are therefore manipulated by accessing
  a piece of memory of a certain size (1-byte, 2-bytes, 4-bytes,
  etc) and then by using logical AND/OR operations to mask off
  the bits. For our purpose here, call this piece of memory
  the access-unit.

  In platforms that support non-int bit fields, the access-unit
  may have size depending on the bitfield type. In the test
  here, we assume bit field access-units are always 4 bytes.

  If a bit field cannot fit into the remaining space of the
  current access-unit it may cross over to the next one
  or it may start in a new access-unit (i.e. the whole
  bit field is in the new access-unit).

  In struct A below, member g is at access-unit boundary.
  Checkout the output bitmap. If g is at bit offset 20
  it crosses the current access-unit boundary (assuming
  access-unit size 4). If it is at bit offset 32,
  it starts at a new access-unit.

  Last member a bit field.

  If the last member of a structure is a bit field, the
  compiler may pad the structure to the end of the
  access-unit. The structure size may bea  multiple of
  word.
  In struct A below, check where the last member i ends
  and where the struct ends.
*/

struct A {
  int a:3;
  int b:3;
  int c:3;
  int d:2;
  int e:4;
  int f:5;
  int g:14;
  int h:17;
  int i:17;
} x;

#define showIt_1 \
  BEGIN          \
  show(x,a)      \
  show(x,b)      \
  show(x,c)      \
  show(x,d)      \
  show(x,e)      \
  show(x,f)      \
  show(x,g)      \
  show(x,h)      \
  show(x,i)      \
  END(x)         \

/*
  Zero-width bitfield.
  Zero-width bitfield may affect the alignment
  of the next member.
  In struct B below, check where the member c
  starts. If it starts at bit offset 32, the
  previous zero-width bitfield must have forced
  it to word boundary.
*/

struct B {
  int a:3;
  int b:3;
  int :0;
  int c:3;
  int d:2;
  int e:4;
  int f:5;
  int g:14;
  int h:17;
  int i:17;
} y;

#define showIt_2 \
  BEGIN          \
  show(y,a)      \
  show(y,b)      \
  show(y,c)      \
  show(y,d)      \
  show(y,e)      \
  show(y,f)      \
  show(y,g)      \
  show(y,h)      \
  show(y,i)      \
  END(y)         \

/*
  Mixed bitfield types.
  In platforms that support non-int bitfields
  (e.g. char, short), different bitfield types
  may not be allowed to allocate in the same
  access-unit.
  Add your own tests to try it out.
  (For example, in struct C below, change
  the first member char q to char q:0 and
  check out the alignment of g and h.)
*/

struct C {
  char q;
  int a:3;
  int b:3;
  int c:3;
  int d:2;
  int e:4;
  int f:5;
  int g:10;
  int h:21;
  int i:17;
} z;

#define showIt_3 \
  BEGIN          \
  show(z,q)      \
  show(z,a)      \
  show(z,f)      \
  show(z,g)      \
  show(z,h)      \
  END(z)         \


/*
  Add your own tests.

  This is the only location you need to
  change to add your own struct tests.

  The showIt_n macros prints the bitmap of the
  structure. You can select which member
  to show. To add your own struct tests,
  just follow the examples above:

  1) Declare the struct.
  2) Define the showIt_n macro. Add
     show(r,t) to show a member;
     r is the struct variable, t the
     Member.
  3) BEGIN and END(r) must be there.

  You can use showIt_4, and 5. main
  already invokes these macros.
*/

#define showIt_4
#define showIt_5

/* end of the test structures  */


/* ---------------------------------------------------------------------- */
/* Helper functions and macros                                            */
/* ---------------------------------------------------------------------- */

int offset(void *v, int sz);
int width(void *v, int sz);
void bigBit (char m[], char name, int start, int len);
void showBit (char m[], int len, char *s);
int  checkEndian();

char map[1024];
int  bigEndian = 1;                      /* flag for endianess, default big */

#define quoteIt(t) #t

/*
  The following macros probe the bit field
  by assigning all 1s (ones) to the field and
  then check where the 1s begin.
*/
#define AllOnes(t) (t=0,t=~t)            /* set t to all 1s                 */
#define off(r,t) (memset(&r,0,sizeof(r)), AllOnes(r.t), offset(&r, sizeof(r)))
#define wid(r,t) (memset(&r,0,sizeof(r)), AllOnes(r.t), width(&r, sizeof(r)))

/*
  Put member t of struct r on the bitmap
*/
#define BIGBIT(m, r,t) bigBit(m, #t[0], off(r,t), wid(r,t))

#define show(r,t) BIGBIT(map, r,t);

/*
  Helpers to make it pretty.
*/
#define BEGIN   bigBit(map,'.',0,sizeof(map));
#define END(r)  showBit(map,sizeof(r)*8, "struct " quoteIt(r));


int main() {
  bigEndian = checkEndian();
  printf("\nByte order is %s endian.\n\n", bigEndian ? "big" : "little");

  showIt_1;
  showIt_2;
  showIt_3;

  showIt_4; /* not used */
  showIt_5; /* not used */

  printf("\n");
  return 0;
```

```
}


/*
    offset
    Return the bit offset of a bit field member.
    On entry:
      v points to the beginning of the struct.
      sz is the struct size.
      The member to be check is set to all ones;
      the rest of the struct is zero.
*/

int offset(void *v, int sz) {
  int i,j;
  char temp, *b = (char*)v;

  /*
  printf("b=%2.2x %2.2x %2.2x %2.2x\n", b[0], b[1], b[2], b[3]);
  printf("  %2.2x %2.2x %2.2x %2.2x\n", b[4], b[5], b[6], b[7]);
  */

  /* which byte start non-zero */
  for (i=0;(i<sz && b[i]==0);i++);

  if (i==sz) return -1;  /* the whole struct is zero */
  temp = b[i];

  /* which bit */
  if (bigEndian)
      for (j=0; j<8 &&!(temp&0x80u); j++) temp = temp<<1;
 else
      for (j=0; j<8 &&!(temp&0x1u);  j++) temp = temp>>1;

  return i*8+j;
}

/*
    width
    Return the length of a bit field.
    On entry:
      v points to the beginning of the struct.
      sz is the struct size.
      The member to be check is set to all ones;
      the rest of the struct is zero.
*/

int width(void *v, int sz) {
  int i,j, len=0;
  char temp, *b = (char*)v;

  /* which byte start non-zero */
  for (i=0;(i<sz && b[i]==0);i++);

  if (i==sz) return 0;  /* the whole struct is zero */

  for (;i<sz && b[i]; i++) {
       /* count the no. of bits in the byte */
       temp = b[i];
       for (j=0; j<8; j++) { len += temp & 0x1u; temp = temp >>1; }
  }

  return len;
}

/*
     bigBit
     Marks the bitmap m from start to start+len with
     the character name.

     m represets a bitmap, one char per bit. The caller
     allocates the memory for m.
     name is a char used to represent the bit field in
     the map.
     start is the starting offset to be marked.
     len is the length to be marked.
*/

void bigBit (char m[], char name, int start, int len) {
  int i;
  for (i=start; i<start+len; i++)
       m[i] = name;
}

/*
     showBit
     Prints the bitmap m. len is the length to
     be printed (i.e. size of the struct).
     s is a sub-heading text to be printed
     at the beginning.
*/

void showBit (char m[], int len, char *s) {
  int i;
  printf("%s:", s);

  for (i=0; i<len; i++) {
       if (i%64 == 0) printf("\n    %3.3d : ", i/8);
       if (i%8  == 0) printf(" ");
       if (i%32 == 0) printf(" ");
       printf("%c", m[i]);
  }
  printf("\n");
}

int checkEndian() {
  unsigned int n = 0x12345678u;
  char *p = (char*)&n;
  if (p[0]==0x12u && p[1]==0x34u && p[2]==0x56u && p[3]==0x78u)
      return 1;
  else
      return 0;  /* assume little if not big */
}
```

OS/390 UNIX Porting Guide - process management

Following is the entire *Process Management* chapter of the *OS/390 UNIX Porting Guide* (as of July 2000). It is on the Web at:

```
http://www-1.ibm.com/servers/eserver/zseries/zos/unix/pdf/docs/portbk_v1r2.pdf
```

It is included here in its entirety for convenience. The additional benefit of accessing it on the Web is that it will possibly be more current.

Process Management

In order to set up, configure, and possibly debug OS/390 UNIX application programs, such as a Webserver, you need to have a basic understanding of the process model that is used within the OS/390 UNIX environment.

In OS/390, we have the following basic categories of work:

- Started tasks (MVS operator start command)
- Batch jobs (submit via JES)
- TSO/E user (logon)
- APPC/MVS transactions (CPI-C allocate)
- IMS (a started task)
- Other server or system address spaces

All work that is created through one of the above requests finally ends up in an address space that holds every piece of information that is required to describe the work at any moment in time (for example, storage control blocks, program(s) to execute, opened data sets, etc.). Though representing the current work, the address space is not the unit of work MVS dispatches. MVS dispatches a TCB (task control block), or an SRB (service request block). They are all dispatchable units that represent work that runs in an address space.

Basically this does not change in the OS/390 UNIX environment, but we work with a slightly different terminology that is based on the concepts of a process and a thread.

Note: Do not run OS/390 UNIX applications from CICS. Support for this capability is under consideration. You also need to be careful about using OS/390 UNIX services from an APPC multitransaction environment. Both of these environments cause problems for OS/390 UNIX because they change the security environment (ACEE) without giving OS/390 UNIX a chance to react.

During this discussion, we will be referring to C functions and assembler callable services (also known as syscalls). Generally, whenever the C RTL (Run-time Library) needs to do something which requires being in an authorized state, a callable service is used. The C language support passes the C function to the kernel address space via an assembler callable service. The OS/390 UNIX callable services are documented in *OS/390 UNIX Programming: Assembler Callable Services Reference*.

When discussing process management, we will consider:

- Processes
- Threads
- Interprocess communications (IPC)
- Signals

Processes

A process maps into an MVS address space and an MVS task environment exists for the process, in terms of a task control block (TCB) and related control blocks. In addition to the TCB, the kernel address space maintains a number of control blocks that represent a process. These control blocks exist within the kernel address space; they are created when an existing standard MVS program begins using OS/390 UNIX System Services (OS/390 UNIX) or when a new process is created by the OS/390 UNIX process creation functions. When an address space begins using OS/390 UNIX services, we say the MVS address space is dubbed as an OS/390 UNIX process. From a UNIX perspective, this means OS/390 UNIX assigns a PID (process ID) to the process. Control blocks in common storage and the kernel address space are built to represent this piece of work. These control blocks build the infrastructure that OS/390 UNIX uses to keep track of all you do. In addition to the process control blocks, task level control blocks are created in the user address space and in the kernel. Each task is treated the same as a UNIX thread.

There are situations where multiple processes may exist within the same MVS address space, and in such a case a process may be running as the job step task or a subtask.

An OS/390 UNIX program may use a number of OS/390 UNIX services to create new processes or to enable multithreading within the process itself. There are no means to prohibit creation of new processes by an application programmer (although BPXPRMXX parmlib settings can limit the number of processes).

To control processes, the following basic services are available:

- fork() function and BPX1FRK service -- Create a New Process

  fork() replicates the current process into a child process. After the fork(), the child process is running in a new address space, with a single task and a single RB, regardless of the task environment of the parent. 8 virtual storage has been copied from the parent to the child address space.

- spawn() function and BPX1SPN service -- Spawn a Process

  spawn() starts a new process, but the new child process is started with another program in the hierarchical file system (HFS), as indicated by the parent process on the spawn() call. After the spawn() call, the two processes continue as independent processes.

  spawn() can create a new process in a separate address space or in the same address space, depending on the setting of the environment variable _BPX_SHAREAS=YES|NO.

  If your application creates a lot of processes and you want better performance, use spawn(). Similar to fork() and exec, spawn() runs much faster and saves resources because it does not have to copy the address space. In fact spawn() can optionally place the new process in the same MVS address space, even further saving system resources. If your application is multithreaded you must use spawn() instead of fork().

- exec family of functions and BPX1EXC service -- Run a Program

  This does not start a new process, but replaces the program in the current process with another program as indicated on the exec call.

- BPX1ATX service -- Attach a Program Residing in the HFS

  This attach_exec service will create a new process in the same address space and pass control to a program in the hierarchical file system. No C function is provide for this service. The equivalent function is provided by spawn() with _BPX_SHAREAS=YES.

- BPX1ATM service -- Attach an MVS Program

  This attach_execmvs service creates a new process in the same address space and passes control to a program in the normal MVS search order (Job Pack Queue, STEPLIB, LPALIB, LINKLIB).

- BPX1SDD service -- Set the Dub Default Service

  You can call the set_dub_default service and set the default level to Process. Then any subtask making a BPX1xxx call will get dubbed as a separate process. As a separate process, each task does not share any OS/390 UNIX resources with the other processes in the address space.

Forking a new process

To start a new process, a process may use the fork() service. Forking is a very well-known concept in UNIX environments, but it is not a function that directly maps into any traditional MVS system services.

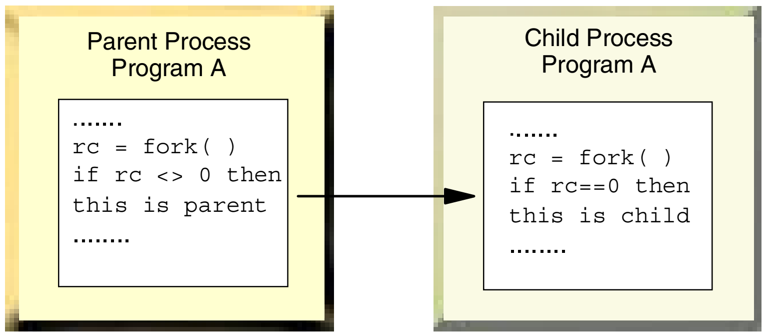


Figure 13-7 Forking a new process

If you have a process running that is executing, for example, program A, and this program calls the fork() function, the kernel address space initiates the following actions:

1. Creates a child address space. Prior to OS/390 V2R4, this was accomplished using APPC initiators. As of OS/390 V2R4, it is done using an internal Workload Manager (WLM) interface. The WLM interface will create a new address space only if the system can tolerate the additional load.
2. Copies user recovery routines and contents supervisor structures from the parent process address space to the child process address space. Private storage (stack, heap, and programs) is propagated from the parent address space to the child address space. The security environment is also propagated.
3. Returns control to the instruction following the fork() call in the parent and child process.

   Just after a fork() call, the two processes are almost identical; the program is the same, they have access to the same storage, and the user security environment is the same. Any HFS file descriptors or socket descriptors that were opened by the parent process are also open and available to the child process. (However, if a file is marked as fdclose on a fcntl() call, the file descriptor will be missing and not propagated to the child.) Positions established by the parent process in sequentially processed files before the fork() call are maintained and preserved in the child process. In fact, the only difference between the two processes is the return value from the fork() call; the parent process receives the process ID (PID) of the child process, while the child process receives a return value of zero. It is important to understand that control is given to the child process at the instruction following the fork() call and not at the program's main entry point, as you, based on traditional MVS experience, might have expected.

   One important aspect of this inheritance concept applies to DD-name allocations of any kind. If the parent process had made a DD-name allocation (using JCL, TSO ALLOC, or dynamic allocation services) before the fork() call, this allocation is not inherited by the child process -- unless it is the STEPLIB DD. The STEPLIB DD is propagated to the child process.

   In most implementations, the parent process will go on doing what it has to do, and the child process will most likely do cleanup and pass control (exec) to a child-specific program that will do whatever the child process has to do.

Spawning a new process

The spawn service is another mechanism for starting a new process. This service works very much like a fork(), except for the fact that the new process is not a copy of the parent process. On the spawn() call, the parent process specifies the name of an HFS program to start in the child process, and the new process is started with this new program being given control at its main entry point.

The default action is to propagate file and socket descriptors to the child process, as is done with a fork(). However, with spawn() the application can specify an fd_map to remap file descriptors, so that the child gets different file descriptors from the parent. If a process is spawned in the same address space, all the DD's are still available. Parent and child in the same address space can use the same DD as long as they understand the rules laid out by Allocation. For example, they which has a DD with DISP=SHR. The rules are complex.

By setting the _BPX_SHAREAS environment variable, the parent process can control if a spawn() call will result in a process being started in another address space or as a task within the same address space as the parent process itself. If _BPX_SHAREAS is set to YES before the spawn() call is initiated, the child process starts within the same address space as the parent process. There are some exceptions where, despite _BPX_SHAREAS=YES, a non-local spawn() (child process starts in another address space) is done. A non-local spawn() is done in any of these cases:

- The program spawned has sticky bit on
- The program spawned is an external link
- The program spawned is a SETUID or SETGID program
- The address space has exhausted its private storage

Some applications allow you to set a configuration variable that is used by the application to control whether new processes are started within the same address space or within a new address space. Where such configuration options are available, it is generally a good idea to turn them on. Starting a new process within the same address space as the parent process requires much less processing and will in general perform better than starting the new process in another address space.

Replacing the program in a process

If a program in a process wants to replace itself with another program, it can use the exec service. This service will preserve the current process environment, but completely replace the program that is running within that process. A successful exec call (the exec family of functions) will never return control to the calling program, but control will be passed to the main entry point of the new program that is specified on the exec call.

The exec service is typically used after a fork() by the child process to replace the parent process program with a child-specific program to perform the child-related functions of the application.

UID/GID Assignment: Process Authorization

As we mentioned earlier, the first attempt to use OS/390 UNIX services dubs the MVS address space as an OS/390 UNIX process. This adds new information to the current address space, of which the UID/GID assignment probably is the most important:

Real UID	At process creation, the real UID identifies the user who has created the process.

Effective UID	Each process also has an effective UID. The effective UID is used to determine owner access privileges of a process. (**Note**: Normally this value is the same as the real UID. It is possible, however, for a program that resides in the hierarchical file system to have a special flag set that, when this program is executed, changes the effective UID of the process to the UID of the owner of the program. A program with this special flag set is said to be a set-user-ID program. This feature provides additional permissions to users while the set-user-ID program is being executed.)
Real GID	At process creation, the real GID identifies the current connect group of the user for which the process was created.
Effective GID	Each process also has an effective group. The effective GID is used to determine group access privileges of a process. (**Note**: Normally this value is the same as the real GID. A program can, however, have a special flag set that, when this program is executed, changes the effective GID of the process to the GID of the owner of this program. A program with this special flag set is said to be a set-group-ID program. Like the set-user-ID feature, this provides additional permission to users while the set-group-ID program is being executed.)

The real UID/GID tells us who we really are; the effective UID and GID are used for file access permission checks; the saved values of UID and GID are stored by the exec function.

See <Table 1> for the relations between the values described above and how they are manipulated by various function calls.

Note: Although we only reference the setuid() function, the same applies to the GID as handled by the setgid() function.

Table 13-1 Summary Showing How the UID May Be Changed

ID	exec set-uid-bit off	exec set-uid-bit on	setuid() superuser
real user ID	unchanged	unchanged	set to UID
effective user ID	unchanged	set from owner UID of program file	set to UID
saved set-user ID	copied from effective UID	copied from effective UID	set to UID

Process groups and job control

In addition to having a process ID, each process belongs to a process group. A process group is a collection of one or more processes. Each process group has a unique process group ID. The most important attribute of a process group is that it is possible to send a signal to every process in the group just by sending the signal to the process group leader. (When sending a kill, you must put a negative sign (-) before the PID of the process group leader in order to have the signal be propagated to the group.)

Each time the shell creates a process to run an application, the process is placed into a new process group. When the application spawns new processes, these are members of the same process group as the parent.

Some process identifiers are used for job control. The several types of process identifiers associated with a process are:

- PID: A process ID. A unique identifier assigned to a process while it runs. The PID is not returned to the system until the parent issues a wait(). Until the wait() is issued, a terminated process still has a PID and its status is ZOMBIE. Each time you run a process, it has a different PID (it takes a long time for a PID to be reused by the system). You can use the PID to track the status of a process with the ps command or the jobs command, or to end a process with the kill command.
- PGID: Each process in a process group shares a process group ID (PGID), which is the same as the PID of the first process in the process group. This ID is used for signaling related processes. If a command starts just one process, its PID and PGID are the same.
- PPID: A process that creates a new process is called a parent process; the new process is called a child process. The parent process (PPID) becomes associated with the new child process when it is created. The PPID is not used for job control.

Process priorities

Process priorities are handled as follows:

- For an MVS address space that gets dubbed, its priority has already been established based on whether it is a batch job, TSO work, started task, etc. Getting dubbed does not change the priority.
- Forked and spawned processes are places in the OMVS subsystem category. Installations control the performance attributes of the OMVS subsystem using SRM or WLM mechanisms. Child processes do not inherit their priorities from the parent. Instead, they are treated as a member of the OMVS subsystem category, which can further be tuned by account number or userid in the appropriate SRM or WLM controls. This is discussed in OS/390 UNIX Planning.

Threads

If a program within an OS/390 UNIX System Services (OS/390 UNIX) process needs to work with more dispatchable units of work, but does not need the advanced functions of fork or spawn, it can use the POSIX threading services that are part of the OS/390 UNIX services.

The application program can create and manage threads via a range of special threading system service calls. To create a new thread, an application program will use the pthread_create() service. If you are writing code in assembler, it is easier to do an ATTACH and let the tasks be dubbed as threads.

There are, in general, three types of threads:

1. Light-weight threading

   OS/390 UNIX does not implement light-weight threads. Light-weight threads are not assigned to individual subtasks, but are managed within one task. This is also sometimes referred to as pseudo-subtasking.

2. Medium-weight threading

   With medium-weight threading services, an application does pthread_create() and after the thread does a pthread_exit(), OS/390 UNIX reuses the task. An example of an OS/390 UNIX application that uses medium-weight threading is the OS/390 Internet Connection Server.

3. Heavy-weight threading

   If the application uses heavy-weight threads, a new MVS subtask is created every time the application requests a new thread to be created. The subtask is terminated when the thread terminates.

OS/390 UNIX resources, such as HFS files, sockets, pipes, and so forth, are available to all threads within a process.

POSIX threads and MVS subtasking are similar in many respects, except for the following: MVS subtasks are normally used to run a piece of code that may run on its own (a separate load module), while the pieces of code that are running on POSIX threads are part of one and the same load module.

Limitation on the number of threads

At the current time, the limiting factor for the number of C threads that you can run in a single address space is the storage below the 16M line. Each thread is an MVS TCB. Each task has a TCB, XSB (extended status block), at least 1 RB (request block), a first save area, and possibly a few other things. These MVS control blocks take up about 1K below the 16M line. Additionally, LE and the C RTL take up about 12K below the line. This is considered a problem and is being worked on.

We generally recommend that the maximum number of concurrent threads in an address space be 100-200. For many applications with large numbers of threads, contention can become a problem. All the work in an address space can drive contention on LE latches, the local lock, and the RSM address space lock for paging activity. Also, debugging can be more difficult.

However, some customer applications are designed in such a way that they successfully use many more threads per process. For example, with proper system tuning, we have seen Java applications that run with well over 1000 concurrent threads in an address space. The largest number of concurrent threads we've seen in a single address space is about 2600 (OS/390 V2R6). But, in this case careful tuning was done, and we ran a simple test case in which all the threads were sleeping.

Stopping threads

An application can stop threads within their process by using pthread_kill() or pthread_cancel().

An operator can terminate an OS/390 UNIX thread, without disrupting the entire process. The syntax of the MODIFY command to terminate a thread is:

```
F BPXOINIT,{TERM}=pid[.tid]
           {FORCE}
```

where `tid` indicates the thread id (TID) of the thread to be terminated. The TID is 16 hexadecimal characters as displayed by the following command:

```
D OMVS,PID=pppppppp
```

In some situations, you may want to want to terminate a single thread when the thread represents a single user in a server address space. Although random termination of a thread usually causes a process to hang or fail, using the MODIFY command to terminate a thread will not cause the process to terminate. Note that some servers, such as Lotus, do not terminate individual threads; rather, all processes are terminated if one terminates.

Porting applications with pthreads

If you are porting pthreads or mutexes, we have a Web page that lists some differences you will encounter. It is:

http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1pth.html

Interprocess communication (IPC)

Four interesting functions in OS/390 UNIX System Services (OS/390 UNIX) come under the heading of interprocess communication:

- Shared memory
- Message queues:
- Semaphores
- Memory mapping

These forms of interprocess communication extend the possibilities provided by the simpler forms of communication between processes: pipes, named pipes or FIFOs, signals, and sockets.

The IPC mechanisms of shared memory, semaphore and message queues are all persistent. To identify abandoned IPC constructs, use the ipcs shell command. To remove them, the owner or superuser can use the ipcrm shell command.

Shared memory

Shared memory is good for sharing data, and it can also be useful for keeping track of resources shared across multiple processes. It saves you from moving the data multiple times, as is done for pipes, message queues, and sockets.

Shared memory provides an efficient way for multiple processes to share data, such as control information that all processes require access to. The processes use semaphores to take turns getting access to the shared memory. For example, a server process can use a semaphore to lock a shared memory area, then update the area with new control information, use a semaphore to unlock the shared memory area, and then notify the sharing processes. Each client process sharing the information can then use a semaphore to lock the area, read it, and then unlock it again for access by other sharing processes. In general, an application would want to have multiple readers and one writer which can only write when it knows the readers are blocked. This can be done with semaphores or with read/write locks. Read/write locks make this much simpler to accomplish.

Shared memory will persist even after all users detach from it. For example, if one process is attached to a shared memory segment and it terminates (either normally or abnormally) without detaching the segment, the segment does not go away.

If you want to use shared memory from C, you can use the C functions: shmget(), shmat(), and shmctl(). For assembler, you can use the services:

- shmget() (BPX1MGT) -- Create/Find a Shared Memory Segment
- shmat() (BPX1MAT) -- Attach to a Shared Memory Segment
- shmctl() (BPX1MCT) -- Perform Shared Memory Control Operations
- shmdt() (BPX1MDT) -- Detach a Shared Memory Segment

shmget() or the BPX1MGT function allows you to define how big an area of shared memory you want. This starts at 4K and the upper limit is controlled by the BPXPRMxx parmlib limit and the amount of storage available in the user region. When you use this function, it reserves space in a kernel data space. Shared memory is permanent, until explicitly freed or detached (just like common storage, CSA).

When you call shmat() or BPX1MAT, it does a getmain for the amount of space in the user private area and uses an RSM service IARVSERV to create page sharing groups between the getmained pages and the pages in the data space. All users (with appropriate permission) that attach to this shared memory segment are similarly connected to these same pages. Hence, after the attach, you share this memory with the other processes which also did the attach. This is regular key 8, user storage which can be used for any existing MVS service. You can read into it, write out of it, and it is shared between your multiple address spaces, but no one else can see it.

If you are using shared memory, be aware of its extended system queue area (ESQA) requirements, and how to reduce the real storage requirements. A number of OS/390 UNIX System Services (OS/390 UNIX) use base OS/390 functions that consume ESQA storage. This storage is fixed, consuming main memory rather than only virtual storage. Installations having constraints on virtual storage or main memory can control the amount of ESQA storage consumed.

If you specify the __IPC_MEGA option on shmget() to request segment-level sharing, it results in significant real storage savings and reduced ESQA usage, especially as number of shares increases. The resulting shared memory segment will be allocated in units of segments instead of units of pages.

Message queues

XPG4 provides a set of C functions that allow processes to communicate through one or more message queues. A process can create, read from, or write to a message queue. Each message is identified with a “type” number, a length value, and data (if the length is greater than zero).

Messages queues are good for handling small messages that are fed to a server. The intended design is that the message queue never get too deep.

A message can be read from a queue based on its type rather than on its order of arrival. Multiple processes can share the same queue. For example, a server process can handle messages from a number of client processes and associate a particular message type with a particular client process. Or the message type can be used to assign a priority in which a message should be dequeued and handled.

If you build up deep queues and use multiple message types for categorizing, this can affect performance.

Message queues are persistent for the duration of the current IPL. You can write a message to a queue and another job or address space can react to it right away or next week. Messages waiting in the queues are kept in kernel data spaces until they are received.

Messages can be very small (1 byte) or quite large (megabytes). For larger messages, consider using shared memory instead. One process can put something in shared memory and a message on a queue can point to it. This avoids moving the data.

The C functions for using message queues are msgget(), msgrcv(), and msgsnd(). The callable services are:

- msgget (BPX1QGT) -- Create or Find a Message Queue
- msgrcv (BPX1QRC) -- Receive from a Message Queue
- msgsnd (BPX1QSN) -- Send to a Message Queue

Semaphores

Semaphores, unlike message queues and pipes, are not used for exchanging data, but as a means of synchronizing operations among processes. Semaphores provide the ability to perform serialization on resources. A semaphore value is stored in the kernel and then set, read, and reset by sharing processes according to some defined scheme.

Typical uses for semaphores are serialization of shared memory, resource counting, and file locking. Frequently, semaphores are used to serialize hunks of shared memory.

The semget() function creates a semaphore set or locates an existing one. Semaphores are convenient for C programmers, but for assembler programming, ENQ is simpler to use and has better recovery and serviceability characteristics.

To serialize between a C program and an assembler program, you can either write an assembler stub for C to do the ENQ or you can have the assembler program code a syscall interface to use semaphores.

You can optimize the behavior of binary semaphores, whereas trying to optimize the behavior of counting semaphores is probably impossible.

- A counting semaphore can range in value from 0 to a large number. Counting semaphores are serialized with a GRS latch, which can inject additional contention into the application.
- A binary semaphore has a value of either 1 or 0, where 0 usually means the semaphore is held and 1 means that the semaphore is available. You can only have one owner of a binary semaphore.

To improve the performance of binary semaphores, use the __IPC_BINSEM option for semget(); this tells the kernel that the application will be using the semaphores only in a binary semaphore fashion. semop() then uses the PLO instruction to atomically update the semaphores in the set, and the GRS latch is eliminated from the semop( ) processing. This implementation is ideally suited to environments where there is heavy semaphore usage and contention. This option was introduced in OS/390 V2R6 and rolled back to OS/390 V2R4 via APAR. See OS/390 C/C++ Programming Guide for further information on semaphore performance.

Memory mapping

In OS/390, a programmer can arrange to transparently map an HFS file into process storage. The use of memory mapping can reduce the number of disk accesses required when randomly accessing a file.

The mmap(), mprotect(), msynch(), munmap(), __map_init() and __map_service() functions provide memory mapping. The callable services are:

- mmap (BPX1MMP) -- Map Pages of Memory
- mprotect (BPX1MPR) -- Set Protection of Memory Mapping
- msynch (BPX1MSY) -- Synchronize Memory with Physical Storage
- munmap (BPX1MUN) -- Unmap Previously Mapped Addresses
- __map_init (BPX1MMI) -- Designate a Storage Area for Mapping
- __map_service (BPX1MMS) -- Set Memory Mapping Service

Signals

The basis for error handling in OS/390 UNIX C/C++ application programs is the generation, delivery, and handling of signals. Signals can be generated and delivered as a result of system events or application programming. You can code your application program to generate and send signals, and to handle and respond to signals delivered to it. These types of signal handling are supported for catching signals: ANSI C, POSIX.1, BSD, and additional functions provided by XPG4.

Examples of ways signal functions can be used are:

- **Maintenance**: Most UNIX systems send a signal to the process in the event of invalid pointers or other indications of a bug in the program. Depending on how the signal handling is set up, this can cause a core dump to be generated and used for debugging purposes by developers.
- **Communication Events**: When two programs are communicating with each other over a file descriptor (it could be a networking (IP) program, a pipe, or something else), if the recipient side of a conversation terminates (normally or abnormally), the sending party receives a SIGPIPE event. Other network related signals are SIGIO and SIGPOLL that indicate an asynchronous I/O event.
- **Timer Functionality**: Either the alarm() or the setitimer() functions cause the signal SIGALRM to be generated.
- **User/tty Interrupts**: Usually the interactive user can cause a signal to be generated by using certain key sequences. For example, Ctrl-C generates a SIGINT and Ctrl-Z causes the SIGTSTP signal to be sent to the process.
- **Interprocess Communication**: We do not recommend using signals for communication. The biggest disadvantage is that multiple signals of the same type can be lost. For general purpose communication, use shared memory, semaphores, messages queues, sockets, and pipes.
- **Process Tracking**: A parent process can have a signal catcher for SIGCHLD, so that it is notified when a child process terminates.

Each process has an action to be taken in response to each signal defined by the system. During the time between the generation of a signal and the delivery of a signal (when the actual action is performed), the signal is said to be pending. It is valid for the process to block it. If a signal that is blocked is generated for a process and the action for that signal is either the default action or to catch the signal, the signal remains pending for the process until the process either unblocks the signal or changes the action to ignore the signal.

A signal can be specified to be blocked in the sigprocmask() and sigsuspend() functions. Each thread has a signal mask that defines the set of signals currently blocked from delivery, and the mask is inherited by a child from its parent. The signal mask is inherited across fork(), exec(), spawn() and pthread_create().

Supported signals - POSIX(OFF)

For OS/390 C/C++ with POSIX(OFF), these signals are supported:

- **SIGABND**: System abend.
- **SIGABRT**: Abnormal termination (software only).
- **SIGFPE**: Erroneous arithmetic operation (hardware and software).
- **SIGILL**: Illegal or invalid instruction.
- **SIGINT**: Interactive attention interrupt by raise() (software only).

- **SIGIOERR**: Serious software error such as a system read or write. You can assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. This minimizes the time required to locate the source of a serious error.
- **SIGSEGV**: Invalid access to memory (hardware and software).
- **SIGTERM**: Termination request sent to program (software only).
- **SIGUSR1**: Reserved for user (software only).
- **SIGUSR2**: Reserved for user (software only).

Supported signals - POSIX(ON)

For OS/390 C/C++ with POSIX(ON), these signals are supported:

- **SIGABND**: System abend.
- **SIGABRT**: Abnormal termination (software only).
- **SIGALRM**: Asynchronous timeout signal generated as a result of an alarm().
- **SIGBUS**: Bus error.
- **SIGCHLD**: Child process terminated or stopped.
- **SIGCONT**: Continue execution, if stopped.
- **SIGDCE**: DCE event.
- **SIGFPE**: Erroneous arithmetic operation (hardware and software).
- **SIGHUP**: Hangup, when a controlling terminal is suspended or the controlling process ended.
- **SIGILL**: Illegal or invalid instruction.
- **SIGINT**: Asynchronous Ctrl-C from the shell or a software generated signal.
- **SIGIO**: Completion of input or output.
- **SIGIOERR**: Serious software error such as a system read or write. Assign a signal handler to determine the file in which the error occurs or whether the condition is an abort or abend. Minimize the time required to locate the source of a system error.
- **SIGKILL**: An unconditional terminating signal.
- **SIGPIPE**: Write on a pipe with no one to read it.
- **SIGPOLL**: Pollable event.
- **SIGPROF**: Profiling timer expired.
- **SIGQUIT**: Terminal quit signal.
- **SIGSEGV**: Invalid access to memory (hardware and software).
- **SIGSTOP**: Stop executing.
- **SIGSYS**: Bad system call.
- **SIGTERM**: Termination request sent to program (software only).
- **SIGTRAP**: Debugger event.
- **SIGTSTP**: Terminal stop signal.
- **SIGTTIN**: Background process attempting read.
- **SIGTTOU**: Background process attempting write.
- **SIGURG**: High bandwidth is available at a socket.
- **SIGUSR1**: Reserved for user (software only).

- **SIGUSR2**: Reserved for user (software only).
- **SIGVTALRM**: Virtual timer expired.
- **SIGXCPU**: CPU time limit exceeded.
- **SIGXFSZ**: File size limit exceeded.

Special notices

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see "How to get IBM Redbooks" on page 208.

- *Experiences Moving a Java Application to OS/390*, SG24-5620
- *Networked Applications on OS/390 UNIX,* SG24-5447
- *Open Source Software for OS/390 UNIX*, SG24-5944
- *OS/390 Version 2 Release 6 UNIX System Services Implementation and Customization*, SG24-5178
- *Porting Applications to the OpenEdition MVS Platform*, GG24-4473
- *Tuning Large C/C++ Applications on OS/390 UNIX System Services*, SG24-5606

Other resources

These publications are also relevant as further information sources:

- *DFSMS/MVS DFSMSdfp Storage Administration Reference*, SC26-4920
- *DFSMS/MVS Using the ISMF*, SC26-4911
- *Learning GNU Emacs, 2nd Edition* by Debra Cameron, Bill Rosenblatt and Eric Raymond, O'Reilly & Associates, ISBN 1565921526
- *OS/390 C/C++ Language Reference,* SC09-2360
- *OS/390 C/C++ Programming Guide,* SC09-2362
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 ISPF User's Guide,* SC28-1239
- OS/390 Language Environment for OS/390, GC28-1945
- *OS/390 MVS Diagnosis: Tools and Service Aids*, SY28-1085
- *OS/390 MVS Diagnosis: Procedures*, SY28-1082
- *OS/390 MVS JCL Reference*, GC28-1757
- *OS/390 MVS Initialization and Tuning Reference*, SC28-1752
- *OS/390 MVS Planning: Workload Management*, GC28-1761
- *OS/390 MVS System Management Facilities*, GC28-1783
- *OS/390 Resource Measurement Facility (RMF) User's Guide*, SC28-1949
- *OS/390 Resource Measurement Facility (RMF) Report Analysis*, SC28-1950
- *OS/390 UNIX System Services Planning,* SC28-1890
- *OS/390 SDSF Guide and Reference,* SC28-1622
- *Using REXX and OS/390 UNIX System Services*, SC28-1905
- *z/OS Language Environment Customization*, SA22-7564

Referenced Web sites

These Web sites are also relevant as further information sources:

- `http://www-1.ibm.com/servers/eserver/zseries/zos/unix/pdf/docs/portbk_v1r2.pdf`

  OS/390 UNIX Systems Services Porting Guide, chapter entitled, “After the port, focus on performance.”
- `http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1ty1.html`

  IBM OS/390 Tools and Toys Web site
- `http://www.gnu.org/software/software.html`

  GNU Web site
- `http://www-4.ibm.com/software/ad/c390/cmvsstlp.htm`

  OS/390 C/C++ STLPort Web site
- `http://ace.cs.wustl.edu/Download.html`

  ACE source code Web site
- `http://tables.dkl.com/`

  Data Kinetics table management site
- `http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1tun.html`

  OS/390 UNIX performance Web pages

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Index

IBM

Redbooks

C/C++ Applications on z/OS and OS/390 UNIX

(0.2"spine)
0.17"<->0.473"
90<->249 pages

C/C++ Applications on z/OS and OS/390

Port UNIX C/C++ applications to z/OS

Learn the strengths of OS/390

Real-world port described

In this IBM Redbook, we focus on how to move applications written in C/C++ from other UNIX operating systems to z/OS UNIX System Services. We highlight the traditional strengths of z/OS, and describe some of the subsystems not always found on other UNIX variations. We address application development tools, the C/C++ compiler, and open source development code such as the C++ Standard Template Library (STL) and the Adaptive Communication Environment (ACE). We also suggest some performance tuning techniques.
Finally, a “real world” port of a C/C++ application is detailed. First we describe the application itself, and then discuss the following aspects of the port:

Before: What we did to set up the application development environment
During: Which issues we encountered while porting to z/OS
After: How we tuned the application once it was running

In conclusion, we summarize our overall findings. Many additional appedixes are included for reference such as:
- A comparison of z/OS and GNU compilers and make tools
- OS/390 C/C++ compiler ASCII support
- STLPort information
- Discussion of dumps
- Performance analyzer output
- OS/390 UNIX Porting Guide - process management

SG24-5992-01 ISBN 0738424544