OS/390 Reliability, Availability, and Serviceability

Guidelines for Ported Software

Bob Abrams

OS/390 Software Design
IBM Corporation, Poughkeepsie, NY

IBM Internal Phone: 8+295-6832
Outside Phone: 914-435-6832
E-mail: abrams@us.ibm.com
Lotus Notes: Robert Abrams/Poughkeepsie/IBM

May 1999

OS/390 RAS Guidelines for Ported Software

Part 1. Introduction

When applications are ported to the OS/390 environment, they generally inherit and can take advantage of the defaults and protections of the broad Reliability, Availability and Serviceability (RAS) facilities that are provided by OS/390:

- Reliability is the capability of a program to perform its intended function under specified conditions for a defined period of time.
- Availability is the capability of a program to perform its function whenever it is needed.
- Serviceability is the ability for a program to capture failure data and enable quick error analysis to determine the cause and solution to a problem that affects the operation of the program.

When a C/C++ application fails, the system ensures that the application is isolated from other work going on in the system, and its recovery doesn't result in a re-IPL ("reboot") of the system. Although the S/390 server's availability characteristics are better than other UNIX servers, the characteristics of a ported business application remain the same as in its original environment unless the application is changed to take advantage of the additional RAS services that are supported by OS/390 (the base MVS operating system and its associated member functions).

When a failure occurs within the application itself, the OS/390 UNIX System Services (OS/390 UNIX) recovery default is to terminate the process (usually corresponding to an OS/390 address space), which reflects the same behavior that would occur in a standard UNIX server. The Language Environment component provides a basic storage dump to help describe the state of the program at the time of the problem. But in general, the RAS characteristics of ported applications, without further investment, are the same as on their source platforms. Very often only sketchy information is provided for debug purposes (error logs, traces, etc.). Many of the programs were originally written to run on workstations or PC servers, where rebooting the operating system is an acceptable recovery action, and an outage only impacts a single or a few users on those systems. The OS/390 equivalent of canceling and restarting the application may clear up an unknown problem that couldn't be diagnosed, but will adversely affect the application's availability delivered to the many hundreds, or even thousands, of clients using the service. Therefore the typical UNIX recovery action (i.e., re-boot the server) is unacceptable when running on S/390 servers.

At the broadest level, OS/390 customers may assume that when the application terminates, it could be restarted automatically through the use of the installation's automation product. However, this requires that the automation product be able to intercept the "signal" from the application that it has encountered some error. This level of function is currently unavailable, unless the application explicitly raises its errors in a form that is recognized by OS/390, such as issuing generic alerts, or "write-to-operator" (WTO) message to the system log.

To obtain RAS characteristics consistent with those expected by OS/390 customers, a level of investment must be planned. OS/390 customers require that products remain continuously available. This means that software products must recover from failures, even if some requests are caused to fail. Useful diagnosis information about product failures must be captured the first time they occur so that problems can be identified and fixes applied (if possible), to minimize re-occurrences. This automatic gathering of information is called “first failure data capture”, and promotes serviceability. The resulting actions that are required include:

- requiring good development techniques like creating descriptive messages with unique message identifiers, and returning erroneous requests with unique error codes
- applying application recovery techniques within your programs, like driving thread termination and retry in the applicable cases (similar to OS/390 capabilities)
- for each call to a service that can return error indicators, testing those indicators
- capturing enough diagnostic data in the event of an error to ensure that the cause can be isolated and repaired (first failure data capture)
- allowing for the delivery of individual fixes for problems
- providing good serviceability (production diagnosis / debugging) documentation

Application recovery processing and unique error messages, ABEND codes, and reason codes are examples of information that can result in improved availability characteristics of the application environment because it can be better managed. Other forms of serviceability, such as tracing and error logging must be enabled in the application, just as would be done when developing OS/390 functions to enable rapid identification of the root causes of problems. Some debug tools are provided for C/C++ applications; they are discussed later in this document. Good documentation on what to look for when diagnosing a problem related to the application is of utmost importance.

Part 2. Basic serviceability guidelines for ported code

The following high level guidelines are intended to be followed when porting an application to the OS/390 environment. This section attempts to describe each guideline in terms that do not require OS/390 expertise, and that should be understood by experienced non-390 programmers. Each guideline area is explained in more detail in Part 4 of this document, along with OS/390 implementation suggestions. You may wish to refer to the detailed sections while determining how to implement each guideline point - the detail follows the same outline as this section.

1. **Minimize the impact of a failure, and improve the availability characteristic of the application.** Verify the recovery logic of the base application, and provide recovery logic where missing. Ensure that the resulting response is appropriate for a high scale, multi-user environment. The objective is to minimize the impact of a failure, and restart the erroneous thread or process so that client programs and users do not perceive an outage. This approach assumes that one or more pieces of work can be sacrificed or restarted to allow the service to recover and remain available for subsequent requests. It is necessary to differentiate where standard process termination is required, as opposed to errors where thread termination and recovery is more appropriate, to improve the availability characteristics of the overall application.

2. **Ensure that clear, unique messages are issued about major events and errors that occur.** Ensure that the program provides sufficient information to identify who initiated the work request, clearly convey the context of the event, and identify any related (prior) events. In some cases it may be desirable to make the error data visible through the use of specific tools or product externals. When errors are surfaced, ensure that unique codes are used to identify the root problem and associated data reflecting the error. All messages must have unique message IDs to allow them to be easily located in a publication, and to facilitate interception of the messages by system automation. Appendix 2 contains a set of guidelines for issuing and handling system and application messages (applicable to IBM and ISV messages).

3. **Identify and document the appropriate ways to obtain and analyze a dump** of the ported application (and associated regions) when needed, or define other ways to obtain a “snapshot” of processing appropriate for the intended user (application programmer, system programmer, IBM). For cases where your application is dumped, determine whether a dump of a single process (address space) or multiple processes is required for diagnosis.

4. **Provide clear documentation** (via external publications or web site), containing suggested parameter settings, areas to examine when a problem occurs, data relationships. Identify common setup and runtime errors, suggested solutions, and places to go to for more information

5. **Capture the flow of control** for an application. Allow the customer to obtain a trace of major events and associated data. If necessary, provide a way for the customer to obtain a trace log of entry/exit points and parameter values (keeping in mind that this will greatly affect the performance of the application). Any tracing must be specifiable without recompiling the

application; it is necessary to allow the trace to be activated dynamically, without requiring the application to be stopped and restarted.

6. **Provide the ability to quickly highlight and display key information in an application or system dump.** The data provided should help in problem diagnosis, and may be used together with data from other dependent system functions. Enable users of the dump viewing and analysis utility to obtain a summary of activity related to the instance of the ported application in the dump. Initially, provide formatters for major control blocks or object data mappings. Subsequently, a diagnostic summary can be created which examines key control blocks, highlights key attributes, status, or other semantics, and summarizes the current state of the program. Such a summary is extremely helpful when diagnosing problems related to the application. Note that this is primarily an OS/390 function, although it could apply to other platforms over time.

Again, suggestions for **how to implement** these guidelines are provided in Part 4.

Part 3. Debug Guide: Serviceability functions provided by OS/390 LE

Before discussing specific implementations of the serviceability guidelines, it is important to understand what functions OS/390 Language Environment (LE) provides to allow the capture of failure data and analysis tools for debugging different aspects of your application. Having a basic familiarity with these tools and services will allow you to more easily select from your options for ensuring that your application can be diagnosed during development, and in production (the field).

LE is an integral part of the processing of any C or C++ (as well as PL/I and COBOL) program. Its primary goal is to provide an interface to many OS/390 system services from the different language environments. Certain basic recovery features are available without changing source code, by specifying a runtime variable. Other features are provided for programmer use to enhance the basic recovery provided, and some tools are provided for the programmer to use as diagnosis aids. Additional information on all of these functions can be found in the *LE for OS/390 Programming Guide* and *LE for OS/390 Debugging Guide and Runtime Messages* publications.

Serviceability and dumping - a brief tutorial

OS/390 prides itself in having good first failure data capture (FFDC) techniques, to capture data for any failures that occur during production. This is important since it is impractical to recreate errors during production, just to collect information about the error. A key aspect of FFDC is the ability to dump specific areas of the system related to the problem, in an efficient fashion. The dump contains a snapshot of the “raw” data areas at the time of the error. The data is not formatted for human consumption until a later point, when a tool called IPCS (Interactive Problem Control System) is used to examine the dump, format data areas and provide a form of analysis related to specific OS/390 components.

LE, on the other hand, provides a formatted dump of the process in error, reporting environment and program data (depending on the value of certain run-time LE options). Because the LE dump (called a CEEDump) is formatted when the data is captured, its content is limited to the specific formatted information. An OS/390 unformatted storage dump, on the other hand, contains much more information, which can be interpreted any time after the data is written to a data set.

Debugging tools, like the VisualAge Remote Debugger or DBX, are generally intended for use during the development and test phases. Because a debugger is run in an isolated environment, usually against a program that has been compiled with different options, it is not intended for use with production-level problems where data is captured in a dump.

Because of the differences described above, each type of dump is intended for specific types of environments:

- In general, a business application that fails during production is well-described by a CEEDump, especially when the program is compiled such that program variables can be included in the dump.

- For system applications, like areas of DB2, TCP/IP, DCE, Component Broker, or some ISV products, an unformatted dump is best since it contains data that resides in your process address space, or in common storage, which can be interpreted at a later time without recreating the problem or suffering the time it takes to gather and format a CEEDump.

Unformatted dumps come in 2 flavors:
- a SYSMDUMP that contains an unauthorized address space (process), and
- an SVC Dump that can contain one or more system address spaces containing operating system function.[1]

Any application that depends on either of these two forms of dump for debugging must provide a set of IPCS formatters and analysis programs to aid in the viewing of the dump and interpretation of the “snapshot”. Internal IBM tools are available to create format models for your key data areas and analysis programs that are ultimately called by IPCS with a dump image.

The steps required to generate and interpret a SYSMDUMP are summarized at the end of this section.

Base serviceability features

The following serviceability features provide “first failure data capture” for the application program; they are specified by an LE runtime variable when executing the program (or via other means, as described in the *OS/390 Language Environment Programming Guide*). Since each approach is managed through the specification of runtime variables, ***the system programmer or application developer can activate them.***

- **Formatted application-level dump** (for debug purposes) taken upon the occurrence of an abend (such as a program check). A Language Environment dump, called a CEEDump (Common Execution Environment Dump), describes the status of the application’s runtime environment at the time of a failure (or any other point in the program through the use of the *cdump* or CEE3DMP function calls). The CEEDump is typically useful in application environments that do not explicitly utilize other MVS or subsystem services. The type of information that can be obtained is controlled by an LE runtime variable called TERMTHDACT (terminating thread action). The resulting output is a printable/viewable report of key LE structures related to the failed program. The important portions of the CEEDump include:

  - Traceback
  - Enclave identifier
  - Thread identifier
  - Condition information for active routines
  - Arguments, registers and variables for active routines
  - Storage for active routines

[1] An SVC Dump can only be triggered by an authorized MVS program, or through the DUMP or SLIP command. A SYSMDUMP can be requested in the unauthorized program’s JCL, or via an LE runtime option, or by using the MVS IEATDUMP service.

- LE control blocks associated with the thread, enclave, process

The content of the dump can be greatly influenced by how the program in error is compiled. If you compile with the -g or TEST option of the C/C++ compiler, the dump will contain all program variables and other program debug data. The 'debug' version of the program contains symbol table information that describes the names, formats and locations of all program variables. LE uses the data to identify the program attributes and values that appear in the CEEDump. An alternative with the C compiler is to specify "TEST(SYM,NOHOOK) GONUMBER" to have the compiler build the symbol information without runtime hooks. So the information is made available without the performance penalty of the plain TEST option, but you do end up with a larger object deck than if you didn't run with TEST at all. Using the c89 or cxx command, this option would be specified as:
-Wc,TEST\(SYM,NOHOOK\),GONUMBER

If the program is compiled without -g or TEST, or using one of the optimization options, this information is not available and does not appear in the dump. Then, the traceback becomes the primary piece of available, useful debug information.

You can specify the following options with the TERMTHDACT runtime option, and the following types of reports are generated. In all cases, you must also specify TRAP(ON) as a runtime option to have LE generate a CEEDump.

- TRACE: Traceback, showing the sequence of functions at the time of the failure. This is the default value.
- DUMP: Complete dump of the LE environment, including information about conditions, tracebacks, variables, control blocks, stack and heap storage.

- **The Traceback (TERMTHDACT=TRACE):** The traceback is usually the first place to examine to determine the sequence of functions when the program abnormally terminated. It is a report of the LE call stack, which shows the last state of the sequence of programs at the time that the dump is taken. Information for each stack element includes:
  - address of the stack element
  - name of the program file (program unit)
  - program unit address and offset
  - entry point name
  - entry point address and offset
  - statement number
  - load module name
  - Service level information

  More information about the traceback can be found in the *LE Debugging and Runtime Messages* book.

- **Unformatted OS/390 application dump (TERMTHDACT=DUMP)** taken upon the occurrence of an abend. This form of dump, called a SYSMDUMP, contains a snapshot of the application's address space, and must be viewed using a tool (IPCS) that can locate and format program structures in the dump. It is especially useful for locating information that is not formatted by the CEEDump, such as control blocks related to a system service used by the application program, and it is particularly useful for IBM Service diagnosis of complex problems. You can specify the following LE runtime values to have a dump taken. In all cases, specify a SYSMDUMP DD statement in the application's JCL, indicating the location of the MVS dump and that it must be *unformatted*. Here is an example SYSMDUMP JCL statement:

```
//SYSMDUMP DD DSN=datasetname,DISP=SHR
```

The following are other LE unformatted dump options that you can specific with the TERMTHDACT keyword:

- UADUMP: Dump taken as a result of an LE 40xx abend, taken after LE's condition handling is completed
- UAIMM: Dump of the immediate abend, taken before the conditional handler's processing. (OS/390 R7 and later); Note that TRAP(ON,NOSPIE) must be specified as a runtime variable for this option to work. If UAIMM is specified from the shell environment, be sure to specify the _BPX_MDUMP environment variable to indicate where the dump should be written.
- UATRACE: Dump, plus a CEEDump containing only the Traceback

- **Formatted OS/390 application dump (TERMTHDACT=DUMP)** taken upon the occurrence of an abend. Following the same logic related to taking MVS unformatted dumps, if you specify a SYSUDUMP DD statement instead of SYSMDUMP, a formatted MVS dump is taken. This dump may be of limited use in the LE environment, since it does not contain LE-specific data, and cannot be reformatted by a tool. Strategically, SYSMDUMP should be taken to ensure that enough data can be captured upon the first failure.

Where to find CEEDumps that are taken

If a CEEDump is taken while the application is started from the UNIX shell, the dump is usually written to the user's current directory. When an application is running in a forked process or invoked by one of the exec functions, the dump is written to a file in your current directory, unless you are running under the root, in which case the file is written to the /tmp directory. The name of the HFS dump file is
`Fname.Date.Time.Pid`
where Fname is usually "CEEDUMP".

In OS/390 R6, when running in the UNIX System Services shell, you can indicate which directory a CEEDump should be written to by specifying the _CEEDUMP_DIR(directory) environment variable. Then, when an error occurs, the CEEDump is written to directory specified in the environment variable, if it is found. Otherwise, it will attempt to write the file to the current directory (if not the root (/) directory, and if the directory is writeable) or /tmp.

When an application is not running in the UNIX shell, the dump is written to a sequential data set allocated during the job. For example, the following CEEDUMP DD statement can be specified in JCL:

```
//CEEDUMP  DD  DSN=my.dataset,DISP=SHR
```

Note: In general, a “data set” is equivalent to what is known on most operating systems as a “file”. An HFS file is a UNIX file and is part of the HFS file system that is contained in an HFS data set. Application programmers generally do not deal with HFS data sets.

Recovery capability and debug information that can be coded into the program

The following C/C++ serviceability features are provided to the ***application programmer*** to exploit as part of the program logic.

- **Eye catchers for major data areas.** All major data areas must include an EBCDIC eye catcher at the beginning of the mapping. This allows the data area to be visible in a storage dump, allowing it to be located easier by the experienced debugger. In addition, it is good programming practice to include a “modification level” following the eyecatcher. The following example demonstrates how to implement an eyecatcher in a C structure:

```
struct xmpl {                       // Example structure block
   unsigned char  xmpl_id[4];       // Control block id
   char xmpl_version;               // Control block version
   int  .....
   ...
   }

#define xmpl_version_current 1
memcpy(some_ptr->xmpl_id,"XMPL",4);   // set eyecatcher
some_ptr->xmpl_version = xmpl_version_current;  // set version number
```

- **Conditional recovery logic, specified in your program.** Recovery logic includes program logic defined in the application program that could clean up and terminate the thread while allowing the process to continue operating. Recovery logic could release serialization, free storage structures, or clean up any other resources that could interfere with the operation of succeeding threads in the process. Conditional logic can be written in the form of
  - C++ *throw*, *try* and *catch* logic. If you use the C++ exception handling model, only C++ routines can catch a thrown object. When a thrown object is handled by a catch clause, execution will continue after the catch clause in the routine. If a thrown object goes unhandled after each stack frame has had a chance to handle it, C++ defines that the `terminate()` function is called. By default, `terminate()` calls abort(). You can call the C++ library function `set_terminate()` to register your own function to be called by terminate. When `terminate()` finishes calling the user's function, it will call `abort().`
  - Using C functions, you can register a signal handler by using the `signal()` function, and you raise a signal using the `raise()` function. You can use the C `atexit()` function to get control during the termination of an LE enclave. C signal handling functions are recognized in C++ applications.
  - Using LE services, you can register a condition handler by using CEEHDLR, and you raise a condition by using CEESGL.

The semantics for all three environments is described in detail in the *Language Environment Programming Guide.*

- **Specifying the program's release or service level**
  The use of this feature is highly recommended for all applications.
  You can specify your program's service level as part of the code source, by specifying the SERVICE option to the C compiler or via a "#pragma options" statement with a string (up to 64 characters) describing the service level of your code. The string is stored into the object module and is displayed in the traceback of the CEEDump if the application fails. The following is an example of the #pragma statement:

  ```
  #pragma options service(PgmABC 1.0)
  ```

  The traceback in the CEEDump only contains the first ten characters of the service level string, so be sure to place the most significant information at the beginning of the string. See C/C++ Language Reference for more information on the SERVICE option.

- **Requesting a CEEDump**
  If you want to generate a dump during your program's logic, such as in a conditional recovery routine, you can do one of the following:
  - Call the C/C++ *cdump*, *csnap* or *ctrace* routines. Each generates a "generic" CEEDump, where the most detailed dump is generated using *cdump*[2]. *csnap* generates a "condensed storage dump" with fewer CEE3DMP options, while *ctrace* generates a CEEDump containing only a traceback. Refer to *OS/390 C/C++ Run-time Library Reference* for more details about the syntax of these functions and the content of the resulting dumps. The following is an example of a cdump invocation:

    ```
    #include <ctest.h>
    int cdump(char *myDumpTitle);
    ```

  - Call the CEE3DMP callable service, which allows you greater flexibility in requesting a CEEDump with specific types of information. The format of CEE3DMP is:
    `CEE3DMP(80-byte-title, 255-byte-options-string, 12-byte-feedback-token)`
    Refer to *Language Environment Debugging Guide and Runtime Messages* for a functional description and the syntax of the CEE3DMP service.

- **Making specific symbol tables available to a CEEDump or debugger with an optimized compile**
  If you do not compile your program with the test option, program variable information is not available in the resulting object code. However, a technique can be used to ensure that key data areas or objects are defined such that they are available in the CEEDump, as well as available to a debugging program (e.g., DBX). By specifying #pragma primdbg (an internal-use option) in a separate program, compiled with opt(0) and TEST, and linked with your optimized compile, you can indicate specific data areas to be recognized. Specify:

  ```
  #pragma primdbg (<name>, s_external)
  ```

---

[2] Invoking *cdump* is equivalent to calling CEE3DMP with the option string: TRACEBACK BLOCKS VARIABLES FILES STORAGE STACKFRAME(ALL) CONDITION ENTRY

where <name> is the name of a variable or data area. For example, to reference the variable called 'regarea', specify:

```
#pragma primdbg (regarea, s_external)
```

Runtime debug analysis tools

The following debug and analysis tools are available to the ***application programmer*** to use in debugging problems in the application's flow.

- **mutex trace**
  LE provides a trace facility that is controlled by specifying the TRACE runtime variable. For example: TRACE(ON,4K,DUMP,LE=2) . By specifying LE=2, mutex init/destroy and locks/unlocks from LE member libraries are recorded. When you specify LE=3, both the entry/exit trace and the mutex trace are activated. In all cases, the trace is captured in a CEEDump. For more information, refer to the *OS/390 Language Environment Programming Reference.*

- **Heap Checker**:
  A common problem in C/C++ programming is accidental overlay of the heap, the storage area provided to you for malloc'd storage. The LE heap checker can be used to isolate the cause of a heap overlay, or other forms of damage to the heap. OS/390 R4 provides the LE *heapchk* run-time option, which you can invoke without changes to your program. *heapchk* is available via APAR back to OS/390 R1. By setting run-time variables, you can control the frequency and starting point of the LE HEAP analysis, which checks the integrity of the heap area whenever storage is obtained. You can activate *heapchk* with the following run-time option:

  ```
  HEAPCHK(ON, frequency, delay)
  ```

  where:

  - ***'frequency'*** is the frequency at which heap checks are performed. Specify the frequency value as n, nK or nM. A value of 1 (the default) causes the heap to be checked at each call to a LE heap storage management service. A value of n causes the heap to be checked *at every nth call to the service.*
  - ***'delay'*** is the delay before heap checks are performed. It is the number of calls to the heap management service that are skipped before activating the heap check function. The delay is specified in terms of n, nK or nM. A value of 0 (zero, the default) causes the heap to be checked from the first call to the LE management service. A value of n causes the heap to be checked *following the nth call to the service.*

  The IBM default for this value is HEAPCHK(OFF,1,0). If a problem is detected during heapchk processing, the following are produced:

  - An error message for each problem found in the heap...for example:
    - CEE3701 Heap damage found by HEAPCHK Run-time option
    - CEE3709I Either the Right pointer or length is damaged in the Free Tree at xxxxxxxx in Heap Segment beginning at yyyyyyyy.
  - The application is ended with an Abend 4042

- **Detecting the storing of data into freed storage**.
  When using the heapchk option, the use of `HEAPCHK(ON,delay,frequency)` with `STORAGE(,heap_free_value)` results in checking the free areas of the heap. LE sets the storage area to a repeating value of choice, like hex ‘FF’. If your program stores into a free area, an error is signaled. The intent is to ensure that the freed storage areas contain the “free value” in them and to verify that your program frees the correct storage area.
  Example: `#pragma runopts(TERMTHDACT(...),HEAPCHK(ON,1,0),STORAGE(,FF))`

- **Identifying runaway malloc’s.**
  The *__heaprpt()* function is a C run-time function you can place in your code to show heap use counts. The counts are reported each time *heaprpt* is used. Use *heaprpt* in areas of your program that you suspect may be causing storage leaks, and look for spikes in the counts, possibly indicating runaway malloc logic. The function returns the current heap storage counts which you can print to a side file for subsequent analysis. Additional information is in the *OS/390 C/C++ Run-time Library Reference*. To use this function, the calling program needs to obtain storage where the heap storage report will be stored. The address of this storage is passed as an argument to *__heaprpt()*.

  You can activate the LE trace to only capture records related to malloc and free, to provide data allowing you to diagnose storage leaks. Specify the following runtime variable: `TRACE(ON,table_size,DUMP,LE=8)` where DUMP indicates that the trace table should be dumped to the CEEDump location when done. Running with this trace active will have some effect on the performance of your application.

- **Obtaining LE formatting and analysis for unformatted dumps**
  When invoking DB2, TCP/IP or other operating-system specific functions, an unformatted dump is usually required for diagnosing problems, to allow access to dumped storage that is not ordinarily formatted in the CEEDump or SYSUDUMP. An unformatted dump can be obtained using the TERMTHDACT UADUMP or UAIMM option, or with the use of the MVS DUMP or SLIP operator commands.

  IPCS (Interactive Problem Control System) is a dump formatting tool that comes with OS/390. It is used primarily by IBM Service and knowledgeable system programmers, as well as ISV service providers, but is available to anyone at a customer installation to use (provided they have access to the appropriate libraries). When viewing the dump using IPCS, a formatter (LEDATA) is available for interpretation of the LE data. LEDATA, an IPCS VERBX (or *verbexit*) formatter, was introduced in OS/390 R4 and continues to be enhanced to provide much of the information required to debug system functions written in C, C++, COBOL, PL/I or other use of LE services. The LEDATA report shows data like the runtime options, the traceback, storage management areas, condition management areas C/C++ information, as well as a summary of the overall Language Environment at the time of the dump. You invoke it from the IPCS command line as follows:

VERBEXIT LEDATA ‘parameter,parameter,...’

Particularly useful parameters include:

- **SUM** - Summary of the LE environment at the time of the dump
- **CEEDUMP** - A CEEDUMP-like report, including the traceback, the LE trace and thread synchronization data areas at the enclave, process and thread levels.
- **SM** - a report of the storage management areas related to HEAP or STACK storage. Individual reports can be obtained by specifying just HEAP or STACK.
- **ALL** - all reports, plus a C/C++ report

Note that quotes must be specified around the parameter string, as shown above.

Other LEDATA parameters are available, but are typically used by IBM Service. LEDATA. The output generated is described in *LE Debug and Messages*.

- Based on a run-time variable, you can **remove LE’s ESPIE** from its recovery path. The ESPIE is an MVS recovery construct that allows fast recovery for program checks in unauthorized environments. However, in doing so, it prevents the activation of POSIX signal handling semantics for abends and program checks. An active ESPIE also prevents the MVS SLIP command from getting control at the time of an abend. Therefore, in some cases the installation or system component will want to specify the **TRAP(ON,NOSPIE)** runtime variable to turn off the operation of LE’s ESPIE.

  The resulting recovery in this environment is provided by LE’s MVS ESTAE environment. This may result in additional overhead when handling program checks in business applications, but it greatly improves the ability to gather diagnostic information in application failure conditions. ***This run-time environment is highly recommended for system functions that are too difficult to debug using standard business application diagnostics.***

- With OS/390 V2R7, the **_BPX_MDUMP** environment variable allows a user to specify where a SYSMDUMP will be written to. Allowable values for **_BPX_MDUMP** are:
  - **OFF**
    Request the dump to be written to the current working directory. This is the default.
  - **MVS data set name**
    Request the dump to be written to a data set. The data set name must be a fully qualified data set name and can be up to 44 characters. The name can be specified in upper and/or lower case and will be folded to uppercase.
  - **HFS file name**
    Request the dump to be written to an HFS file. The file name can be up to 1024 characters. The HFS file name must begin with a slash. The slash refers to the root directory, and the file will be created in that directory.

How to Generate and Interpret a System Dump (for applications started at the UNIX shell)

A system dump can be obtained when the application is started from the OS/390 UNIX shell.

Prior to OS/390 V2R7, the shell user can indicate the need for a dump by allocating a SYSMDUMP data set for the TSO/E session. For example:

```
ALLOCATE DDNAME(SYSMDUMP) DSNAME(MYDUMP.DATASET) LRECL(4160) OLD
```

Upon recognizing the dump dataset allocation, the system:

- Creates a file in the user's working directory
- Names it `coredump.pid`, where *pid* is the process ID for the process being dumped.
- Writes a core dump in the file. The core dump is a SYSMDUMP dump.

To use the core dump, do the following:

1. Copy the file into an MVS data set with a record length (LRECL) of 4160. You can use the TSO/E OGET command to do this.
2. Analyze the dump, using IPCS and specifying the dataset name. Specify
   `VERBX LEDATA 'CEEDUMP'`
   If the process is not recognized as the primary address space and task, you will need to specify the TCB: `VERBX LEDATA 'CEEDUMP,ASID(asid),TCB(tcbaddr)'`
   You can also use the IPCS subcommands STATUS and SUMMARY FORMAT CURRENT to obtain a general report of the failed environment system.

For OS/390 V2R7 and beyond, to obtain a system dump, follow these steps:

1. Specify where the dump should be written. If the dump is to be written to an MVS dataset, specify the following shell command: `export _BPXK_MDUMP=`*`filename`* where *filename* is a fully-qualified dataset name, allocated with the following information: LRECL=4160, BLKSIZE=4160, and RECFM=FBS. To have the dump written to an HFS file, specify `export _BPXK_MDUMP=`*`filename`* where *filename* is a fully-qualified HFS filename. For example: `ALLOCATE DDNAME(DUMPDS) PATH('/tmp/mydump.dmp') ...`

2. Specify LE runtime variable TERMTHDACT(UADUMP) or TERMTHDACT(UAIMM). If UAIMM is specified, be sure to also specify TRAP(ON,NOSPIE). With UADUMP, if you want the CEEDump to be to be written to a specific directory, specify the following environment variable: _CEEDUMP_DIR(directory) .

3. Run the program. If a program check occurs, a SYSMDUMP is written to the specified file.

4. If the dump is written to an HFS file, use the "OGET" TSO command to copy the dump from the HFS to a pre-allocated MVS dataset. For example:
   ```
   OGET /user1/mvsdump.dmp  'USER1.MVSDUMP.MYPROGRAM'
   ```
   Specifying an MVS dataset with _BPXK_MDUMP avoids the need for this copy step.

5. Use IPCS, specifying the dataset name. Use the IPCS subcommands STATUS and SUMMARY FORMAT CURRENT to obtain a general report of the failed environment system. Specify: `VERBX LEDATA 'CEEDUMP'`
   If the process is not recognized as the primary address space and task, you will have to specify the TCB: `VERBX LEDATA 'CEEDUMP,ASID(asid),TCB(tcbaddr)'`

Part 4. OS/390-Specific Implementation Suggestions

Part 2 provided an initial description of guidelines that should be included when porting the programs to OS/390, and could be followed when developing the product on non-OS/390 platforms. This section includes several implementation suggestions organized in the same order as listed in Part 2. All suggestions described in this section are further-documented in LE, C/C++, and OS/390 publications.

1. Minimize the impact of a failure, and improve the availability characteristic of the application.

- Eliminate single points of failure where possible. If your application provides critical support in a thread, and that thread fails, allow for the ability to queue up requests, clean up any prior resources and restart the function. Such recovery is usually easy to do if designed in, and prevents your users from thinking that your process is "down".
- Verify the recovery logic, and provide recovery logic where missing.
- In your signal handler, determine which signals should result in termination of the process (the default) or just termination of the thread. If terminating the thread, communicate with another thread of your application to restart another thread to take over the failing function. See Appendix 1 for an example of signals that can be restarted.
- If your application is driven as a started task, you can use the OS/390 Automatic Restart Manager (ARM) to restart the application upon a failure. ARM macros can be used to "register" the application (the address space), indicate that it is ready to accept work, and "deregister" when the application is done and you don't want it restarted. More information on ARM can be found in *MVS Programming: Sysplex Services Reference*, GC26-1772.

2. Ensure that clear messages are issued about major events and errors that occur.

- Set unique message IDs, reason codes, and texts in all messages, abends and return codes
- Where possible, surface the reason code value when identifying an error by its "errno" value. This MVS-specific value allows problem areas to be identified more specifically than the generic "errno" value. You obtain the reason code value through a call to the __errno2() C/C++ function. The reason codes are documented in *OS/390 UNIX Messages and Codes*.
- Ensure that all messages follow the guidelines listed in Appendix 2, including having unique message ids.
- Record all errors to a log. This may be a private error log, the UNIX System Services syslog daemon, or one of the MVS system logs (e.g., hardcopy-only WTO, Logrec). If a private error log is maintained, ensure that it contains a valid time stamp, and program and thread information for each entry, to aid in correlation. This requires that the program contain "catch" logic to intercept the error, in order to surface it.
- If errors are reported in a private error log, determine which of them would be important to automate, and issue a system message for them as well.
- If your program provides a function for a caller, notify the caller of your API via a unique return code and reason code. In some architectures, the error may be represented as a thrown exception.

- Don't use WTOR (MVS operator message with ability to reply with a response) with unless you really need to. If you do, ensure the operator action is intuitive, especially where the results of the reply are consequential.

3. Identify and document the appropriate ways to obtain and analyze a dump

- Enable the capture of the right type of dump upon failure of the program. This will allow problem diagnosis to be performed for production-time problems, usually without having to recreate the problem. Options include:
  - LE CEEDump (a formatted dump that you can view with a browser), containing the traceback, and additional information, depending on the runtime variable set (TERMTHDACT = TRACE/DUMP/UADUMP/UAIMM). If a CEEDump is desired, specifying "DUMP" is recommended.
  - LE-driven MVS abend dump (SYSMDUMP) - an unformatted dump of the abending task or address space, in accordance with what the application is authorized to "touch". This can be obtained with TERMTHDACT = UADUMP or UAIMM. Note that UADUMP results in both a CEEDump and a SYSMDUMP being taken.
  - Write signal handler logic to request the CEEDump, and an MVS unformatted dump (via IEATDUMP), in the event of an abend-related signal. Recovery logic should be considered at this point as well. (IEATDUMP is currently only available in assembler or PLX.)
  - Attempt to debug problems using CEEDumps. They may be sufficient if the application is a single process and does not extensively use system services. However, if system functions are used an unformatted dump will be required.
  - If multiple address spaces are expected to be involved in a failure, you will need to investigate the use of an authorized unformatted dump containing multiple address spaces (call an SVC Dump, obtain through the use of the SDUMPX macro). If an unformatted dump is taken, IPCS will need to be used to examine its contents.

4. Provide clear product documentation

- Provide diagnostic information in external publications or via web pages. Clearly define information such as:
  - Diagnostic procedures for common problems
  - Tracing available and how to interpret its content
  - Return/reason codes, abend codes, error messages
  - Major control block structures, and how they are used by the different threads of your application.
  - If the application is a multiprocess environment, show how the resulting OS/390 address spaces relate to each other, and the specific data structures that represent the state of each
  - What to look for in a dump
  - System services used. Refer to IPCS VERBX formatters and analysis routines that report on those system services.

5. Capture the flow of control for an application.

- Provide the ability to dynamically turn on the collection of trace records representing major events within the application, writing the trace records to an in-storage wrapping list, with the ability to write records to an external file.
- ***Any trace implementation needs the ability to be activated dynamically, and allow request filtering by component, at a minimum.*** In many cases, even component-level filtering can result in too many records, and wrapped buffers, so additional filtering by level of detail, or work request (e.g., transaction) being processed is highly recommended
- In OS/390, this can translate to using the MVS Component Trace (CTRACE) structure, which requires the definition of a data gathering and buffering scheme specific to your application. (Reuse code is available for internal IBM use.) An alternative is to use GTF, which provides the GTRACE macro for specifying a trace record. However, when GTF is turned on, it might impact system performance independent of your specific application tracing need. Therefore, CTRACE is the preferred method.
- If errors are recorded to stderr (standard error) , provide defaults that target the message stream to a data set. Do not default to using JES spool, as this will impact overall system availability.
- Your code should exploit the trace to surface exception conditions (which should be logged to an external file as well), basic significant events, and detail (e.g., entry/exit), provided you can filter the selection of each. ***Exceptions should always be traced/logged <u>by default.</u>***

6. Provide the ability to quickly highlight and display key information in an application or system dump.

- Provide IPCS models for formatting major data areas in a dump. IPCS formatters are also needed for trace records. Provide IPCS analysis routines (VERBX programs) to drive analysis of the dump from your application's point of view. For example, determine and report on outstanding requests, serialization held or other logic that would be important to quickly diagnose what may be wrong with your program. Locate and format key control blocks or data areas
- Identify major areas in an unformatted dump: Create MVS "name/token" pairs to represent the names of major data areas and their corresponding addresses. This way system services can be used to locate your blocks in a dump. The "name/token" table is always dumped in an unformatted dump. The name can be used with the IPCS NAMETOKN subcommand to obtain the corresponding token, and use the token as an address reference into the dump. This eliminates the need to build private tables containing similar data to use in your IPCS VERBX routines. It is recommended that you create name/token pairs for malloc'd storage that will be referenced in an unformatted dump.

  Use the IEANTCR callable service to create a name/token pair. The caller passes both a unique name and the token to be associated with it. For example, a program could pass the name of a task/process level control block as the name, and its address as the token. Use the IEANTRT service to retrieve the token for a given name. This could be a “well known” control block name whose address could be retrieved via the IPCS NAMETOKN subcommand.

- When using the *heapchk* run-time option, combining HEAPCHK(ON,delay,frequency) with STORAGE(,heap_free_value) results in checking the free areas of the heap. When the run-time heap checker finds a heap error, it causes a U4042 abend, which should drive an unformatted dump (see the later sections on gathering and formatting unformatted dumps). When viewing the dump using IPCS, you can specify “LEDATA HEAP,ALL” and the resulting dump analysis will locate errors in each heap segment and report them. The heap dump analyzer checks for nodes with bad lengths, node addresses not on doubleword boundary, heap pointers pointing outside the heap segment, and other indicators of problems. If a problem is located, a message is displayed telling the nature of the problem.

7. Other programming practices required for serviceability

- Define a viewable acronym (EBCDIC or ASCII) and version number at the beginning of each major data area (C mapping, etc.).
- Ensure that any requests made to the application are correct, and returned with a specific error indicator if they are not.

For additional information, please refer to *OS/390 System Application RAS Checklist*, available from Bob Abrams. In addition, the following references may be useful:

- *OS/390 C/C++ Run-time Library Reference,* SC28-1633
- *OS/390 MVS Programming: Assembler Services Guide,* GC28-1762.
- *OS/390 MVS Programming: Assembler Services Reference,* GC28-1910.
- *OS/390 Authorized MVS Programming: Assembler Services Guide,* GC28-1763.
- *OS/390 Interactive Problem Control System User's Guide*, GC28-1756.
- *OS/390 Interactive Problem Control System Commands*, GC28-1754.
- *OS/390 Diagnosis: Tools and Service Aids*, LY28-1085.
- *OS/390 MVS Programming: Sysplex Services Reference,* GC26-1772.
- *Language Environment for OS/390 and VM Debugging Guide and Runtime Messages,* SC28-1942.
- *Language Environment for OS/390 and VM Programming Reference,* SC28-1940.
- *Language Environment for OS/390 and VM Programming Guide,* SC28-1939.
- *OS/390 UNIX System Services Command Reference,* SC28-1892.
- *IBM C/C++ for MVS/ESA Library Reference,* SC23-3881.
- *IBM C/C++ for MVS/ESA Programming Guide,* SC09-2164.
- *"MVS Dump Suppression: DAE provides Benefits"*, by R.M. Abrams and D. Williamson, Enterprise Systems Journal, September 1994.

Appendix 1. Example of Determining the Recovery from Certain Signals

Your application should be able to recover from certain types of signals issued by the UNIX System Services. The following table shows an example (taken from Component Broker/390) of which signals it makes sense to retry in your application, instead of taking the default action -- failing the process. The base information in this table is taken from *C/C++ Library Reference* SC28-1663, Table 32. The table contains all of the signals delivered by UNIX System Services.

Signal Value	Default Action(*)	Meaning	Suggested Recovery Action in Signal Handler
SIGABND	1	Abend	Attempt retry / terminate thread
SIGABRT	1	Abnormal termination, sent by abort()	Attempt retry / terminate thread
SIGALRM	1	Timeout signal, sent by alarm()	None (don't catch)
SIGBUS	1	Bus error	Attempt retry / terminate thread
SIGFPE	1	Arithmetic exceptions that are not masked, like overflow, division by zero and incorrect operation	Attempt retry / terminate thread
SIGHUP	1	Controlling terminal is suspended, or the controlling process ended	Terminate process
SIGILL	1	Detection of an incorrect function image	Attempt retry / terminate thread
SIGINT	1	Interactive attention	Terminate process
SIGKILL	1	A termination signal that cannot be caught	None (don't catch)
SIGPIPE	1	Write to a pipe that is not being read	Terminate thread
SIGPOLL	1	Pollable event occurred	None (don't catch)
SIGPROF	1	Profiling timer expired	None (don't catch)
SIGQUIT	1	A quit signal for a terminal	Terminate process
SIGSEGV	1	Incorrect access to memory	Terminate thread
SIGSYS	1	Bad system call issued	Attempt retry / terminate thread
SIGTERM	1	Termination request sent to the program	Terminate process
SIGTRAP	1	Internal for use by dbx or ptrace.	None (don't catch)
SIGURG	2	High bandwidth data is available at a socket	None (don't catch)
SIGUSR1	1	Intended for use by user	None (don't catch)

		applications	
SIGUSR2	1	Intended for use by user applications	None (don't catch)
SIGVTALRM	1	Virtual timer has expired	None (don't catch)
SIGXCPU	1	CPU time limit exceeded	Terminate process
SIGXFSZ	1	File size limit exceeded	Attempt retry / terminate thread
SIGCHLD	2	An ended or stopped child.	None (don't catch)
SIGDCE	2	Signal is used by DCE	None (don't catch)
SIGIO	2	Completion of input or output	None (don't catch)
SIGIOERR	2	A serious I/O error was detected	None (don't catch)
SIGWINCH	2	Window size has changed	None (don't catch)
SIGSTOP	3	A stop signal that cannot be caught or ignored	None (don't catch) - cannot catch STOP
SIGTSTP	3	A stop signal for a terminal	Terminate process
SIGTTIN	3	A background process attempted to read from a controlling terminal	Terminate process
SIGTTOU	3	A background process attempted to write to a controlling terminal	Terminate process
SIGCONT	4	If stopped, continue	None (don't catch)

(*) The Default Actions in the above table are:
1. Normal termination of the process.
2. Ignore the signal.
3. Stop the process.
4. Continue the process if it is currently stopped. Otherwise, ignore the signal.

Appendix 2. System and Application Message Guidelines

External Guidelines

- Systems should be managed, not run.
- The system should be managed by exception.
- Don't display a message unless you are requesting the operator to do something or you're answering a specific query.
- Allow the operator to request only the information he or she is interested in.
- Don't answer a query with more information than the operator asked for.
- Responses to operator queries should only go back to the operator that requested the information.
- Don't ask the operator to do something unless it is important.
- Don't ask the operator to do something unless it truly requires a human to do it (mount a tape, load paper, etc.).
- Don't ask the operator to do something that he or she does not have the expertise to do.
- Only put out choices which the operator can be reasonably expected to understand.
- If you have to interact with the operator, do it in as few interactions as possible. Don't be overly verbose. At the same time, do not use obscure jargon.
- If an operator decision is required, list the options in increasing order of severity.
- If a message must be issued, don't assume that it will be read by a human -- it may be read by various forms of automation.
- Direct the message to the appropriate decision maker, not to the world. Route messages functionally.
- Don't violate the Principle of Least Astonishment: "It didn't work the way I expected it to."
- Consider what the operator has to do to handle your message:
  - Retention/erasure of action messages
  - Retrieval of action messages
- Valid message IDs are typically 7-10 characters log, following a naming convention that identifies the component, subcomponent, and a message number within the subcomponent. Having unique message IDs aids in automation, helps enable automated publication lookup tools, and facilitates easier manual lookup in a book.

OS/390-Specific Guidelines

- Be sure to flag operator responses as operator responses.
- Keep unnecessary messages off of the Master Console.
- Use DOM to delete action messages when the action has been taken or the situation requiring the message no longer exists. This allows any recording related to the message to be cleaned up and allows the message to roll off the MVS console if it is held.
- Multiple line messages should be issued as a single multi-line message.
- Don't use WTORs as a way to do command entry; use the MODIFY/CANCEL/STOP command facilities.
- Don't use Write To Log (WTL); use WTO with "Hardcopy Only" specified.

- Be sure to document which messages are:
  - new
  - changed (and how)
  - deleted
- Command response: MCSFLAG=RESP, Descriptor code 5
- If you have single-line message with inserts, use the TEXT= parameter on the WTO macro to pass the address of the text line. Similar capability exists for multi-line WTOs.
- Use 4-byte console id on command responses
- Propagate CART (Command And Response Token) on all command responses. It is used by automation programs for correlation of command and response(s). Depending on the type of program, the CART can be found in one of the following locations:

• CIB:	CIBXCART	(Started task command information block)
• CMDX:	CMDXCART	(Command exit routine)
• CSCB:	CHCART	(Most operator commands)
• XSA:	XACART	(internal representation of operator commands)
• SSCM:	SSCMCART	(on the command subsystem interface)
• WQE:	WQECART	(for system messages)

Some WTO definitions

The Write To Operator (WTO) macro is used to specify system messages that are to appear on the MVS operator console and in the MVS system log. Messages issued via WTO are also made available to system automation products. WTO can be used by unauthorized programs, and certain features are limited to use by authorized programs. WTO is described in *OS/390 MVS Programming: Assembler Services Reference*.

Rough WTO syntax:

```
WTO '... text ...',ROUTCDE=(16,28),DESC=(4),CONS=(26),
     CART=cart,MCSFLAG=(flagsettings)
```

WTO also accepts the address of a text line, making it easier to build a message with inserts and specify it to the macro: `WTO TEXT=addr, ......`

See the Assembler Macro Reference book for more details on WTO syntax.

- **Operator:** a person whose job is to keep the system operational, which includes responding to key requests, managing jobs, controlling output, etc.
- **Routing code:** a number that identifies the operational area related to the message, used by the system to route the message to one or more consoles requesting to display messages with that routing code (specified as one or more numbers using the ROUTCDE parameter). Routing codes are used by some installations who like to segregate the message traffic across multiple consoles
- **Descriptor code:** a number that instructs the system on how to treat the appearance of a message, and whether it describes an operator action or command response (specified as one or more numbers using the DESC parameter).
- **Command response:** message that describes the final status of a command request. Some message responses are in the form of a multiple-line display of data, called a display response. Specified as DESC=(5).

- **Master console:** primary console used by the installation for displaying messages (specified as ROUTCDE=(2).
- **Hardcopy-only:** when you specify this for a message, that message is not displayed on any console, and it is written directly to the system log (specified as MCSFLAG=(HARDCPY).

Appendix 3 - Introduction to reading a CEEDump

The CEEDump is an application debug dump that is formatted by the OS/390 Language Environment when the runtime option TERMTHDACT is set to the appropriate value. *Language Environment Debugging Guide and Runtime Messages* defines each specific section of a full CEEDUMP in detail. The following are the major segments of the dump:

- Page Heading
- Caller Program and Offset
- Registers on entry to CEE3DMP
- Enclave identifier
- Thread information
- Traceback
- Condition Information for Active Routines
- Arguments, Registers, and Variables for Active Routines
- Control blocks for active Routines
- Storage for active routines
- Control Blocks associated with the Thread
- Enclave Control Blocks
- Enclave Storage
- Process Control Blocks

The following are some hints to help you when examining a CEEDump:

- Many times, when you are looking at a traceback, you would like to **verify the data passed from call to call.** You can do this by looking at the Dynamic Storage Area (DSA), also known as the stack frame, associated with each call. The traceback indicates the address of the DSA associated with each calling routine. The DSAs are listed below the traceback table. If you look at Register 1 in a stack frame, the value in that register represent the address of the parameter list when 'that routine' called 'the next routine' (forward in time ... going up the traceback list). That address will be within the 'caller's' stack frame itself. You can go and look at the parameter list to see the data passed to the called routine. This is very useful for verifying:
  - How many parameters were actually passed? Are they all there?
  - Is the parameter in the correct format ... is it the data or the address of the data!

  Many times this helps narrow down where a problem is occurring,

- When looking at registers in a DSA (stack frame) note that these register are reflective of the environment at the time of the call from 'that' routine' to the next. When a call is made, the registers will represent the following:

  R0 is used by COBOL static call
  R1 is a pointer to a parameter list or 0 if no parameter list is passed
  R2-R11 is unreferenced by LE (Caller's values are passed transparently)
  R12 is a pointer to the Common Anchor Area (CAA) if entry is to an external routine

R13 is a pointer to the caller's stack frame
R14 is a return address
R15 is the address of the called entry point

- **For Heap corruption problems,** you can usually determine where the corruption is by using the LE Storage Management Internals document and check the HPCB, HANC, and the Cartesian Trees in the dump. Usually some the of registers at the time of the dump point to the LE Enclave Storage Management Control Structure. At a minimum, Reg 12 will point to the CAA. Depending upon when LE found a problem in the heap other registers may point to the HPCB or the particular HANC (heap segment) that is corrupted.

- **If you want an SVC dump instead of a CEEDUMP** from a probe situation (since a DCE probe invokes CEE3DMP), you can simply set a slip where the probe is issued. This would of course be second failure data capture... but you may be in a situation where you NEED an SVC dump for a probe instead of the CEEDUMP. The address that you would need to set the slip on (instruction fetch) can be easily determined from the data in the traceback. Take the following example:

```
Traceback:
DSA Addr    Program Unit  PU Addr   PU Offset  Entry              E Addr  ....
04655100    /ibmdce/mvs/full/dw4.xx7/dce/src/svc/MVS/svcprobe.c
                          0228B9B8  +000009EE  dce_svc_probe
                                                                  0228B9B8 ....
046543D8     /ibmdce/mvs/full/dw4.xx7/dce/src/rpc/runtime/rpcaclmg.c
                          0266CF80  +000009CC  rpc_acl_build_default_entry
                                                                  0266CF80 ...
 ...etc...
```

In the above traceback, you can see that it is the rpc_acl_build_default_entry routine which call the probe. In particular the call to the probe occurred at X'0266CF80" + X'9CC' = X'0266D94C'. You can verify this address by looking at the DSA for the rpc_acl_build_default_entry routine. The value in R14 should be the next instruction after the call to probe (BALR) ... and in this case R14 is X'0266D94E' ... which is the return register.

Refer to the SLIP command for more information on setting a slip for an instruction fetch. Be careful that when using the address from the CEEDUMP, you should know that that instruction will occur at the same address. Otherwise the slip won't hit (or will hit for some other instruction.) If the probe is issued from a DLL routine, and the DLL is running in LPA, then the instruction will be the same. If the probe is issued from a fetched module (say from the link list) the module may not load/run from the same place. In that case you could set the slip, but use module name and displacement to indicate when to take the dump.

Appendix 4 - IPCS 101: A short IPCS Tutorial

Getting Started

When setting up IPCS at your installation, you can determine how to invoke the IPCS panels, as an ISPF option from the main ISPF panel. For example, let's say you set up IPCS to be invoked via option 9. Choose option 9 and you will get the main IPCS panel, which looks like the following:

```
----------------------- IPCS PRIMARY OPTION MENU  ----------------------
OPTION  ===>
                                                           *******************
   0  DEFAULTS    - Specify default dump and options       * USERID  - ABRAMS
   1  BROWSE      - Browse dump data set                   * DATE    - 99/05/20
   2  ANALYSIS    - Analyze dump contents                  * JULIAN  - 99.143
   3  UTILITY     - Perform utility functions              * TIME    - 15:41
   4  INVENTORY   - Inventory of problem data              * PREFIX  - ABRAMS
   5  SUBMIT      - Submit problem analysis job to batch   * TERMINAL- 3278
   6  COMMAND     - Enter subcommand, CLIST or REXX exec   * PF KEYS - 12
   T  TUTORIAL    - Learn how to use the IPCS dialog       *******************
   X  EXIT        - Terminate using log and list defaults

Enter END command to terminate IPCS dialog



 F1=HELP      F2=SPLIT     F3=END       F4=RETURN    F5=RFIND     F6=RCHANGE
 F7=UP        F8=DOWN      F9=SWAP     F10=LEFT     F11=RIGHT    F12=RETRIEVE
```

To view your dump for the first time, enter your option '0' defaults. On the line that states:

```
Source  ==> NODSNAME
```

Change "NODSNAME" to the name of the dataset containing the dump. For example: `DSNAME('SYS1.DUMP01')` and hit enter. Now exit the "defaults" panel.

Choose option "1" to "Browse" the dump. You will see a panel with your DSNAME already provided (came from the defaults you set). Hit enter. Note that you do not necessarily need to provide the dump dataset name on the defaults panel first - you can provide it on the "browse" panel instead. Furthermore, if you have a default name that appears on the browse panel, you can override it with a new name of a dump dataset to browse.

Next, you will get some messages while IPCS is retrieving the dump data. If you get the following message, respond with "Y".

```
BLS18160D May summary dump data be used by dump access? Enter Y to use, N to
bypass
```

This makes the summary portion of the dump available to browse as well.

You should then get to a panel that looks something like this:

```
DSNAME('SYS1.DUMP01') POINTERS  ----------------------------------------
PTR   Address  Address space                                Data type
00001 00000000 ASID(X'0041')                                AREA
```

```
**************************** END OF POINTER STACK ****************************
```

Put your curser on the 00001 value under PTR and enter an 's' to select viewing address space X'0041' (actually the address space ID that's in your dump). Note that if your dump contained more than one address space ID (ASID), that they would be enumerated on this panel, and you could 's' select any one of them to browse.

After entering "s" on the dump pointers panel on the ASID you wish to review, you will start browsing the beginning of the dump - starting with virtual address "0" of the address space. At this point you can start to use the various IPCS commands and subcommands to browse different locations in storage, information in the summary dump, locate strings, etc. Please refer to the IPCS Commands book for a complete list and description of IPCS subcommands.

Common IPCS commands and subcommands

The following are some common IPCS commands for use when browsing a dump:

- **IPcs**: Invoke an IPCS subcommand, for example 'IP STATUS FAILDATA' invokes the status subcommand
- **Locate**: View a particular storage location while in the BROWSE storage panel.
- **Find**: Search dump for a specified value
- **CBFormat**: Format a control block
- **UP RIGHT LEFT DOWN**: Scroll data options

The following are some common IPCS *sub*commands for use when browsing a dump:

- **CBFormat**: Format a control block
- **CTRACE**: Format component trace entries (for example OE CTRACE)
- **Find**: Locate data in a dump
- **FINDMOD**: Locate a module in the dump (searches the Symbol table, active link pack area queue, and link pack area directory)
- **GTFtrace**: Format GTF Trace records (if provided in the dump)
- **List**: Display storage
- **LPAMAP**: List LPA Entry Points
- **RUNChain**: Process a chain of control blocks
- **STatus**: Describe system status...psw, regs, etc.
- **SUMMary**: Produces a variety of reports depending upon specified keyword. For example IP SUMMARY TCBSUMMARY displays and formats a variety of fields in a TCB control block, for those TCBs in the currently browsed ASID.
- **TSO**: Execute a TSO command
- **VERBEXIT**: Run an IBM or Installation provided Verb Exit Routine

The VERBEXIT subcommand must be entered along with a program name or verb name, which represent an exit routine. IBM supplies verbexit routines, and the user/installation may write their own as well. Some commonly used verbexit routines are:

- **LOGDATA**: Format Logrec Buffers
- **SUMDUMP**: Display Summary Dump Data with the SVC dump
- **VSMDATA**: Format Virtual Storage Management data

Formatters of interest

The following are some useful tools for formatting data in a dump:

- OMVSDATA: From browse you can invoke:

  ```
  ip omvsdata detail process
  ```

  The `omvsdata` subcommand can format all processes and threads that OS/390 UNIX manages. This is particularly useful when debugging loops and hangs. You can identify which process/thread information to display,from a set of address spaces with the ASIDLIST parameter. Process/thread summary is just one of four types of data that OS/390 UNIX can report on. The four types are:
  - **Process**
  - **Exception**
  - **Communications**
  - **Storage**

Refer to the *IPCS Commands Reference* for more details on the `omvsdata` IPCS subcommand.

- **VERBX LEDATA:**
  - **HEAP option**: Provides a mapping of LE heap storage...for example:

    ```
    ip verbx ledata 'heap,detail,tcb(tcbaddr)'
    ```

    - Maps all heaps associated with a particular enclave.
    - Note that you can use various options, such 'caa' and 'asid'.
    - Since heaps are associated with an enclave, if you use the tcb option, you must choose the tcb that represents
    - the IPT thread (initial process thread) for your enclave.
    - Specifically maps the free chain, maps all malloc'd and free'd area, and totals free and malloced pieces.
    - **Hint:** If you dumped more that one asid, you must supply the asid as well as the tcbaddr.

  - **CEEDUMP option:** Provides the 'traceback' and 'LE trace' data

    ```
    ip verbx ledata 'ceedump'
    ```

IPCS Publications

Refer to the following OS/390 IPCS publications for a complete description of IPCS functionality, customization and usage:

- *OS/390 MVS IPCS Commands*, GC28-1754
- *OS/390 MVS IPCS Customization*, GC28-1755
- *OS/390 MVS IPCS User's Guide*, GC28-1756

Acknowledgements

I thank the following people who helped me ensure the clarity and accuracy of this paper: Kevin Evans, Don Ewing, Tim Hahn, Kevin Kelley, Roger McKnight,Wayne Rhoten, Carol Rozella, Bart Tague, John Thompson.