CALLMON

A Procedure Call Monitor for OpenVMS Alpha

Version 1.0

24 July 1996

Thierry Lelégard

(lelegard@swthom.enet.dec.com)

Abstract

CallMon is a low-level debug utility for OpenVMS Alpha. With CallMon, an application can monitor all calls to a selected set of procedures, taken from any shareable image.

CallMon is especially useful to implement higher level tools which need to monitor the activity of some system routines. A typical usage of CallMon is the monitoring of the activity of the virtual memory (intercepting calls to routines like LIB$GET_VM and LIB$FREE_VM).

CallMon is delivered as object library and source files. There is currently no public location for CallMon. To receive a kit, send a mail to the author.

CONTENTS

1. OVERVIEW OF ROUTINE INTERCEPTION 3

2. PREPARING TO USE CALLMON 5

2.1 REBUILDING CALLMON 5
2.2 LINKING AN APPLICATION AGAINST CALLMON 5
2.3 AVOIDING RESIDENT IMAGES 5

3. USING CALLMON FROM C 7

3.1 INITIALIZING CALLMON 7
3.2 INTERCEPTING A ROUTINE 7
3.3 PRE-PROCESSING AND POST-PROCESSING ROUTINES 8
3.4 INTERCEPTING ALL ROUTINES IN AN IMAGE 10
3.5 DUMPING THE LIST OF IMAGES 11
3.6 DUMPING THE LIST OF ROUTINES 11
3.7 DISASSEMBLING MACHINE CODE INSTRUCTIONS 11

4. USING CALLMON FROM ADA 13

4.1 USING CHARACTER STRINGS 13
4.2 SPECIFICATION OF PACKAGE CALLMON 13
4.3 DECLARING CALLBACK ROUTINES 15
4.3.1 Pre and post-processing routines 15
4.3.2 Disassembler output routine 15
4.4 LINKING AN ADA APPLICATION AGAINST CALLMON 16

5. USING CALLMON IN A MULTI-THREADED ENVIRONMENT 17

5.1 REENTRANCY 17
5.2 PROGRAMMING HINTS 17
5.3 PROGRAMMING HINTS WITH ADA TASKING 17

6. IMPLEMENTATION OF ROUTINE INTERCEPTION 18

6.1 OVERVIEW OF OPENVMS ALPHA IMAGES 18
6.2 OVERVIEW OF OPENVMS ALPHA CALLING STANDARD 18
6.3 HOW TO INTERCEPT A ROUTINE 19
6.4 MODIFYING ALL REFERENCES TO A ROUTINE 20
6.4.1 Modifying fixups 20
6.4.2 Modifying relocations 20
6.4.2.1 PC-relative branch 21
6.4.2.2 BSR replacement 21
6.4.2.3 BSR depth 22
6.4.2.4 BSR replacement policy 22
6.5 MODIFYING THE ENTRY CODE OF A ROUTINE 23
6.6 PREDEFINED UNINTERCEPTABLE AND UNRELOCATABLE ROUTINES 24
6.6.1 Uninterceptable because of CallMon usage 24
6.6.2 Uninterceptable and unrelocatable because of routine structure 24
6.6.3 Declaring additional routines 24
6.7 TRACING CALLMON OPERATIONS 25

Warning: As you may have (already?) noticed, the author's mother tongue is not English. Consequently, this document is expected to contain lots of mistakes and awkward sentences. Please accept my humble apologies in advance...

1. Overview of Routine Interception

The main feature of CallMon is the interception of routines.

CallMon can intercept any routine which is declared in the symbol vector of any shareable image. To intercept a routine, the application simply calls a CallMon procedure named **callmon$intercept**. After interception, all references to this routine are revectored to a *jacket routine* inside CallMon.

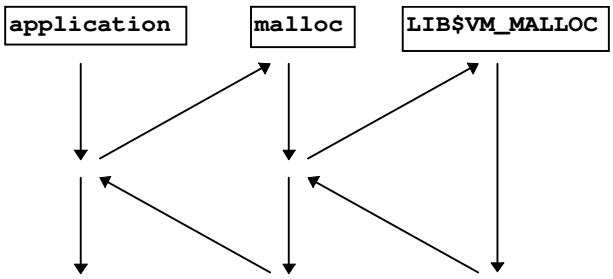
Whenever the intercepted routine is called, the jacket is called instead. This jacket routine performs the following steps:

1. It calls an optional user-written pre-processing routine. This routine was specified by the application when callmon$intercept was called. This user-written routine receives various useful informations such as the name of the intercepted routine on behalf of which it was called and the list or arguments of this intercepted routine.
2. Then, the jacket calls the actual routine which was intercepted with its original arguments.
3. The jacket calls an optional user-written post-processing routine.
4. Finally, the jacket returns to the caller. It returns the result value which was returned by the intercepted routine at step 2.

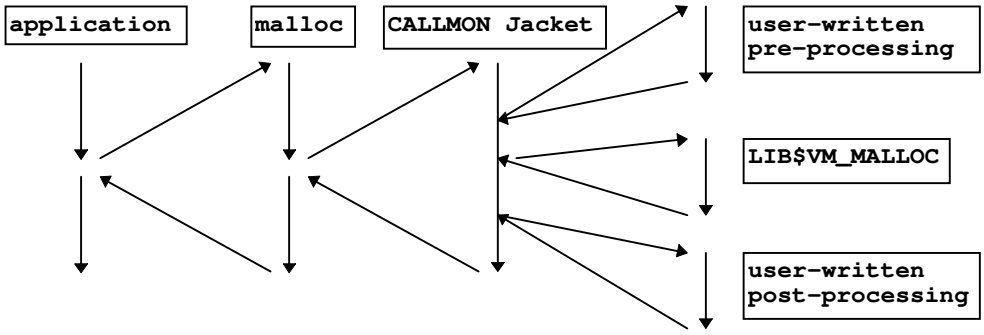
From an application point of view, the interception of the routine is transparent: the routine was executed with the correct arguments and the caller received the correct result. However, some module in the application gained control at that time and could take appropriate actions such as logging the results in a file.

Example:

Consider an application written in C which allocates memory using malloc. The malloc routine (actually called DECC$MALLOC in image DECC$SHR.EXE) calls a routine named LIB$VM_MALLOC in image LIBRTL.EXE. This call flow can been drawn that way:



If we want to trace the memory allocations, we use CallMon to intercept LIB$VM_MALLOC. CallMon inspects all images in the process (including DECC$SHR.EXE) and replaces all references to LIB$VM_MALLOC by a reference to a jacket routine inside CallMon. Once the interception is completed, the call flow becomes:



The user-written pre-processing and post-processing routines gain control between the calls to malloc and LIB$VM_MALLOC. The user-written routines both have acces to the arguments which were passed by malloc to LIB$VM_MALLOC. The post-processing routine has access to the returned value of LIB$VM_MALLOC.

2. Preparing to use CallMon

CallMon is delivered as zip-compressed file containing all source files and an object library. You may choose either to use the provided library or to rebuild CallMon.

The API of CallMon is described in definition files for the C and Ada languages. However, CallMon can be used from any other language, provided that you write the appropriate definition file if the language needs one.

2.1 Rebuilding CallMon

To rebuild CallMon, you need the following tools:

- DEC C compiler. Version 5.3 or higher is recommended.
- MACRO-64 native Alpha assembler. Note that this is a layered product which must be separately ordered. On OpenVMS Alpha, the built-in MACRO command invokes the MACRO-32 compiler, not the native assembler.
- MMS (Module Management System), available as a single product or as a component of DECset. MMS is not really a requirement for CallMon.

If you have MMS installed on your system, simply use the provided MMS description file:

```
$ MMS /DESCRIPTION=CALLMON.MMS
```

If you want to rebuild CallMon with the debug options, use the macro DEBUG with MMS:

```
$ MMS /DESCRIPTION=CALLMON.MMS /MACRO=DEBUG=1
```

If you do not have MMS on your system, use the DCL command procedure named CALLMON$BUILD.COM which is provided with CallMon.

2.2 Linking an application against CallMon

Since CallMon references a couple of OpenVMS internal definitions, any application using CallMon must be linked using the /SYSEXE qualifier.

Example:

```
$ LINK APPLI.OBJ, CALLMON.OLB/LIBRARY/SYSEXE
```

2.3 Avoiding resident images

CallMon will try to modify the code and linkage sections of the shareable images which are activated in the process. This is possible only if these images are mapped in the process space (P0/P1/P2). If an image has some code or linkage sections inside the system space (S0/S1/S2), CallMon will not modify it. This will not break the behaviour of the application but this will prevent some calls from being traced.

If a shareable image is installed with the /RESIDENT option, its code sections are mapped in system space. Additionally, if /SHARE=ADDRESS_DATA option was used, its linkage sections are also mapped into system space. All frequently used images like LIBRTL, DEC C run-time library and DECwindows/Motif images are installed that way.

To allow CallMon to trace all calls, the application should not use the system mapping of these images and map a private copy in its process space. This can be done very easily by defining a logical name for each of these images, like:

```
$ DEFINE LIBRTL SYS$SHARE:LIBRTL.EXE;
```

Note the trailing semi-colon at the end of the definition. It specifies *use the highest version* and thus prevent from using the installed version (even if the installed version is the highest one).

A command procedure named CALLMON$RESIDENT.COM is provided with CallMon. It finds all resident images in the system and defines a logical name for each of them in the LNM$PROCESS table.

This command procedure should be run once in each process which will run an application using CallMon. If this command is not run, the application will work properly, however calls to and from resident images will not be traced.

3. Using CallMon from C

All public definitions of the API of CallMon are provided in a header file named callmon.h:

```
#include "callmon.h"
```

Most CallMon routines return a status code. The status codes of CallMon are defined in a header file named callmonmsg.h. However, there is no need for the application to include this header; this is already done in callmon.h.

3.1 Initializing CallMon

There is no need to explicitely initialize CallMon. It initializes itself the first time it is called. However, since this initialization takes some time, the application may choose to initialize it at a proper time, before the first call, using the following routine:

```
void callmon$initialize (void);
```

This routine may be safely called several times. The actual initialization is performed only the first time.

Initialization operation

When CallMon initializes itself, it reads the names of all routines from all active images in the process. This includes the main executable image all all shareable images.

If another shareable image is dynamically merged later (as a result of LIB$FIND_IMAGE_SYMBOL for instance), this image will not be known to CallMon. First, the application will not be able to intercept routines from this image. Second, if a routine is intercepted, it will be not not intercepted when called from the dynamically merged shareable image.

If such a situation occurs, simply explicitely relink the application against this shareable image. This should not be a problem since using CallMon already imposes to relink the application.

3.2 Intercepting a routine

To intercept a routine, you must provide its name (a nul-terminated character string). There is no need to specify the image the routine belongs to since CallMon has read the names of all routines during initialization.

To intercept a routine, call the following procedure:

```
uint32 callmon$intercept (
    char*               routine_name,
    callmon$intercept_t pre_routine,
    callmon$intercept_t post_routine);
```

This routine returns a status code and receives the following parameters:

Parameter	Usage
routine_name	Name of the intercepted routine. This is a nul-terminated character string.
pre_routine	User-written pre-processing routine. See the actual profile of this routine below. The parameter pre_routine may be the NULL pointer, in which case no user-written routine is called before the actual intercepted routine.
post_routine	User-written post-processing routine. See the actual profile of this routine below. Like pre_routine, the parameter post_routine may be the

Parameter	Usage
	NULL pointer.

If callmon$intecept is called several times for the same intercepted routine, each call replaces the action of the previous one. There can be only one pre-processing procedure and only one post-processing procedure at a time for a given intercepted routine.

To revert the effect of the routine interception, simply call callmon$intercept specifying the NULL pointer for both the pre-processing and post-processing routines. Note, however, that the routine remains intercepted and redirected to CallMon. But CallMon acts as a transparent filter in that case.

3.3 Pre-processing and post-processing routines

The profile of the pre-processing and post-processing routines is defined in callmon.h as follow:

```
typedef void (*callmon$intercept_t) (
    callmon$arguments_t* arguments,
    uint32               caller_invo_handle,
    char*                routine_name,
    uint64               (*intercepted_routine)(),
    uint64               (*jacket_routine)());
```

The pre-processing and post-processing routines receive the following parameters:

Parameter	Usage
`arguments`	This structure describes the conditions into which the intercepted procedure was called. See details below.
`caller_invo_handle`	This longword is the *invocation handle* of the routine which called the intercepted routine. This handle may be used to get the address of the caller or the whole traceback. See the *OpenVMS Calling Standard* for more details about invocation handles.
`routine_name`	Name of the intercepted routine. This is a nul-terminated character string. It must not be modified by the user-written routine.
`intercepted_routine`	Procedure value of the original routine, before it was revectored to the jacket routine of CallMon. This procedure is normally called by CallMon after the pre-processing routine and before the post-processing routine.
`jacket_routine`	Procedure value of the replacement routine inside CallMon. This is the routine which is actually called wherever the original routine would have been called.

The *arguments* structure is described in callmon.h as follow:

```
typedef struct {
    uint32 post_processing;
    uint32 call_it;
    uint64 result_r0;
    uint64 result_r1;
    uint64 result_f0;
    uint64 result_f1;
    uint64 arg_count;
    uint64 arg_list [];
} callmon$arguments_t;
```

Field	Description
post_processing	This is a boolean. It is set to zero when the routine is called in the pre-processing phase and set to one when the routine is called in the post-processing phase. This boolean is only useful if the same routine is used in both cases.
call_it	This is a boolean. In the context of the pre-processing routine it gives the opportunity to suppress the call to the intercepted routine. The initial value of this boolean is set to one (TRUE) by CallMon before calling the pre-processing routine. This means that the intercepted routine will be actually called. If the pre-processing routine sets this boolean to zero (FALSE), the intercepted routine will not be called. Thus, the pre-processing routine can mimic the behaviour of the intercepted routine, reading the argument list and writing the result in the appropriate result_r0 to f1 fields. In the context of the post-processing routine, this boolean indicates whether the intercepted routine was actually called or not.
result_r0 result_r1 result_f0 result_f1	Returned values. Depending on the type of result, the OpenVMS Alpha Calling Standard specifies that the returned value should be passed either in R0, both R0 and R1, F0, both F0 and F1. The post-processing routine must know the type of the result for the intercepted routine and selects the appropriate fields.
arg_count arg_list	Argument list which was passed to the original routine. The first field is the number of arguments, followed by the array of arguments. Note that each argument is 64 bits wide.

One can (and should!) wonder of the following situation: Consider two intercepted routines A and B. Whenever A gets called, the user-written pre-processing procedure is called. What does happen if this pre-processing routine calls B, or even calls A again? Is the same pre-processing procedure called again, calling A again and ending in a wild recursion loop? Of course not! CallMon always determines if an intercepted routine is called in the context of a pre-processing or post-processing procedure. If this is the case, the intercepted routine is transparently called without pre or post-processing.

Caveat 1: Testing the address of the original routine

If the same pre or post-processing procedure is used to intercept different routines, but should take different actions depending on the intercepted routine, it is faster to test the routine address than the routine name. However, carefully write the way you do this.

Let's take an example: a user-written routine named *interceptor* is used as pre-processing routine for two procedures: *proc_a* and *proc_b*. One could discriminate the intercepted routine the following way:

```
extern uint64 proc_a ();
extern uint64 proc_b ();

void interceptor (
    callmon$arguments_t* arguments,
    uint32               caller_invo_handle,
    char*                routine_name,
    uint64               (*intercepted_routine)(),
    uint64               (*jacket_routine)())
{
    if (intercepted_routine == proc_a) {     /* WRONG !!!! */
        /* interpret arguments of proc_a */
    }
    else {
        /* interpret arguments of proc_b */
    }
```

However, as the comments tends to indicate, this is the wrong method. When CallMon has modified all references to procedure *proc_a*, it has also modified the reference in the test:

```
if (intercepted_routine == proc_a) {
```

Consequently, the value of expression ***proc_a*** is now the address of the jacket routine and no longer the address of the original routine. As you have already guessed, the previous test must be rewritten as:

```
if (jacket_routine == proc_a) {     /* GOOD !!!! */
    /* interpret arguments of proc_a */
}
```

Caveat 2: Reading floating-point arguments and results

In the Alpha architecture, floating point data may be represented in a number of formats: IEEE formats (S-float, T-float) and VAX formats (F-float, G-float, D-float).

Each type of floating point data has a *register format* and a *memory format.* IEEE floating point data have identical register and memory formats. However, VAX floating point data have different formats.

- In the argument list (`arguments->arg_list` array), all floating point data are represented in their *memory format.* Use the appropriate instruction (LDT or LDG) to load the data in a register.
- In the result values (`arguments->result_f0` and `arguments->result_f1` fields), all floating point data are represented in their *register format.* Always use a LDT instruction to load the data, even for VAX floating point data types.

3.4 Intercepting all routines in an image

The following routine intercepts all public routines in a shareable image:

```
uint32 callmon$intercept_all (
    char*               image_name,
    callmon$intercept_t pre_routine,
    callmon$intercept_t post_routine,
    callmon$boolean_t   override);
```

This routine is equivalent to a repeated sequence of calls to callmon$intercept on all routines in the image.

Parameter	Usage
`image_name`	Logical name of the image. The logical name of an image does not contain any device or directory specification. « LIBRTL » and « DECC$SHR » are correct image logical names. « LIBRTL.EXE » or « SYS$SHARE:DECC$SHR » are not.
`pre_routine`	User-written pre-processing routine. Same usage as with callmon$intercept.
`post_routine`	User-written post-processing routine. Same usage as with callmon$intercept.
`override`	If set to FALSE, the routines which were already intercepted are not modified. They keep their previous pre and post-processing routines. Note that an intercepted routine with both a NULL pre and post-processing routine is considered as non-intercepted. If the override parameter is set to TRUE, all routines are intercepted with the new pre and post-processing routines.

A definition of the boolean data type is proposed in callmon.h:

```
typedef enum {
    CALLMON$C_FALSE = 0,
    CALLMON$C_TRUE  = 1
} callmon$boolean_t;
```

3.5 Dumping the list of images

The following routine dumps on the standard output a list of all images in the process. Various informations such as the base and end addresses are displayed.

```
uint32 callmon$dump_image_tree (void);
```

3.6 Dumping the list of routines

The following routine dumps on the standard output a list of routines. For each routine, CallMon displays its name, its procedure value and code entry. If the procedure is intercepted, it also displays the procedure value and code entry of the jacket routine which replaces the original routine.

```
uint32 callmon$dump_routine_tree (callmon$dump_routine_t rclass);
```

The parameter *rclass* selects which routines should be dumped. Its type is defined in callmon.h as follow:

```
typedef enum {
    CALLMON$C_ALL_ROUTINES          = 0,
    CALLMON$C_INTERCEPTED_ONLY      = 1,
    CALLMON$C_UNINTERCEPTABLE_ONLY = 2,
    CALLMON$C_UNRELOCATABLE_ONLY    = 3
} callmon$dump_routine_t;
```

Value	Usage
CALLMON$C_ALL_ROUTINES	All routines, from all images are displayed.
CALLMON$C_INTERCEPTED_ONLY	Displays only the routines which are currently intercepted.
CALLMON$C_UNINTERCEPTABLE_ONLY	Displays only the routines which are not allowed to be intercepted. CallMon prevents some low-level predefined routines from being intercepted because this would interfere with the correct behaviour of the application.
CALLMON$C_UNRELOCATABLE_ONLY	Displays only the unrelocatable routines. See the section entitled « *Modifying relocations* » for a definition of *unrelocatable* routines.

3.7 Disassembling machine code instructions

The following routine is not really related to routine interception. It is used internally by CallMon to generate debug information and is provided as public routine for general purpose:

```
void callmon$disassemble (
    void*                          address,
    size_t                         size_in_bytes,
    void*                          displayed_pc,
    callmon$disassemble_output_t output_routine,
    void*                          user_data);
```

This routine disassembles a section of Alpha machine code instructions.

Parameter	Usage
address	Address were the first instruction is stored.
size_in_bytes	Size of the code section to disassemble. This size is given in bytes. Each Alpha instruction is 4 bytes long.
displayed_pc	Starting Program Counter of the code section. This value is normally equal to the *address* argument. The *displayed_pc* argument is only useful if the code section has been copied in a different place.
output_routine	Routine to call to perform the output. If this argument is the NULL pointer, each instruction is displayed on the standard output. Otherwise, the specified user-written routine is called for each instruction. The required profile of this routine is described below.
user_data	A user-specific parameter to pass to the output routine.

The profile of the output routine is described in callmon.h as follow:

```
typedef void (*callmon$disassemble_output_t) (
    void* pc,
    char* opcode,
    char* operands,
    void* user_data);
```

Parameter	Usage
pc	Value of the Program Counter for this instruction. When the output routine is called for the first instruction, the value of *pc* is the *displayed_pc* which was given to callmon$disassemble.
opcode	A character string containing the opcode of the instruction.
operands	A character string containing the operands of the instruction. In branch instructions, the target address is computed and displayed according to the *displayed_pc* argument. If the target address is equal to the code entry of a known routine, the name of the routine is displayed instead.
user_data	The user-specific parameter which was given to callmon$disassemble.

4. Using CallMon from Ada

CallMon can be used in Ada as easily as in C. In the distribution kit of CallMon, a file named CALLMON_.ADA is provided. This file contains the specification of a package named CALLMON where the data types and routines of CallMon are declared.

The following sections only describe the Ada-specific syntaxes for CallMon. For a description of each routine and type, please refer to the section named « *Using CallMon from C* ».

The definition of CallMon status codes are included in another package named CALLMONMSG. The specification of this package may be found in file CALLMONMSG_.ADA.

4.1 Using character strings

Each data type and routine which were described in the previous sections have a corresponding Ada declaration in the specification of package CALLMON.

However, the careful reader will notice that the import pragmas in file CALLMON_.ADA do not interface with the normal C routines. For instance, the Ada routine *Callmon.Intercept* is bound to the external symbol *callmon$intercept_dsc* (instead of the expected *callmon$intercept*).

The Ada interface of CallMon actually uses alternate entry points. This is due to the differences in the way Ada and C use character strings. Handling C-like nul-terminated strings in Ada is not very elegant (although the DEC Ada predefined package *C_Types* makes things easier than it used to be). Consequently, CallMon contains alternate entry points (with names ending in « *_dsc* ») which use string descriptors instead of nul-terminated strings.

Moreover, wherever the « *_dsc* » entry point was used, the corresponding callback also receives a string descriptor instead of a nul-terminated string.

Let's take an example: the application declares a pre-processing routine for *Routine-1* and *Routine-2*, respectively using *callmon$intercept* and *callmon$intercept_dsc*. When *Routine-1* gets called, its pre-processing routine receives a nul-terminated string (becausse *Routine-1* was intercepted using *callmon$intercept*). However, when *Routine-2* gets called, its pre-processing routine receives a string descriptor (becausse *Routine-2* was intercepted using *callmon$intercept_dsc*).

These conventions make things much easier with Ada because the predefined type String may be used everywhere.

4.2 Specification of package CALLMON

This section lists the specification of package CALLMON, as found in file CALLMON_.ADA. For readability, the interface pragmas and representation clauses were removed from the following listing.

You may notice that all routines which return a status code have two overloaded definitions. The first one includes the status code as output parameter. The second declaration omits the status code. This allow an application to ignore the status code (what a horrible thing!) more easily.

```
with Lib;
with System;
with Condition_Handling;

package Callmon is

    subtype Cond_Value_Type is Condition_Handling.Cond_Value_Type;

    subtype Address is System.Address;

    subtype Integer_8  is System.Integer_8;
    subtype Integer_16 is System.Integer_16;
    subtype Integer_32 is System.Integer_32;
```

```
subtype Integer_64 is System.Integer_64;

subtype Invo_Handle_Type is Lib.Invo_Handle_Type;

--
-- Procedure argument list.
-- According to OpenVMS Alpha Calling Standard, an argument list
-- is made of 0 to 255 64-bits arguments.
--

subtype Argument_Count is Natural range 0..255;
subtype Argument_Index is Argument_Count range 1..Argument_Count'Last;

type Argument_List_Type is array (Argument_Index range <>) of Integer_64;

subtype Maximum_Argument_List_Type is
    Argument_List_Type (Argument_Index);

--
-- Argument to pre-processing and post-processing procedure.
--

type Arguments_Type is
record
    Post_Processing : Boolean;
    Call_It         : Boolean;
    Result_R0       : Integer_64;
    Result_R1       : Integer_64;
    Result_F0       : Integer_64;
    Result_F1       : Integer_64;
    Arg_Count       : Integer_64;
    Arg_List        : Maximum_Argument_List_Type;
end record;

--
--  CALLMON Initialization (optional, automatically done otherwise)
--

procedure Initialize (Status : out Cond_Value_Type);
procedure Initialize;

--
--  Intercept a routine.
--

procedure Intercept (
    Status       : out Cond_Value_Type;
    Routine_Name : in  String;
    Pre_Routine  : in  Address := System.Null_Address;
    Post_Routine : in  Address := System.Null_Address);

procedure Intercept (
    Routine_Name : in  String;
    Pre_Routine  : in  Address := System.Null_Address;
    Post_Routine : in  Address := System.Null_Address);

--
-- Intercept all routines in an image.
--

procedure Intercept_All (
    Status       : out Cond_Value_Type;
    Image_Name   : in  String;
    Pre_Routine  : in  Address := System.Null_Address;
    Post_Routine : in  Address := System.Null_Address;
    Override     : in  Boolean := True);

procedure Intercept_All (
    Image_Name   : in  String;
    Pre_Routine  : in  Address := System.Null_Address;
    Post_Routine : in  Address := System.Null_Address;
    Override     : in  Boolean := True);

--
-- Disassemble a range of instructions.
--
```

```
        procedure Disassemble (
            Start_Address  : in Address;
            Size_In_Bytes  : in Natural;
            Displayed_PC   : in Address;
            Output_Routine : in Address := System.Null_Address;
            User_Data      : in Address := System.Null_Address);

        --
        -- Dumping the list of active images.
        --

        procedure Dump_Image_Tree (Status : out Cond_Value_Type);
        procedure Dump_Image_Tree;

        --
        -- Dumping the list of known routines.
        --

        type Dump_Routine_Type is (
            All_Routines,
            Intercepted_Only,
            Uninterceptable_Only,
            Unrelocatable_Only);

        procedure Dump_Routine_Tree (
            Status : out Cond_Value_Type;
            Class  : in  Dump_Routine_Type);

        procedure Dump_Routine_Tree (
            Class  : in  Dump_Routine_Type);

    end Callmon;
```

4.3 Declaring callback routines

4.3.1 Pre and post-processing routines

In Ada, the user-written pre-processing and post-processing routines must be declared the following way:

```
    procedure Interceptor (
        Arguments           : in out Callmon.Arguments_Type;
        Caller              : in     Callmon.Invo_Handle_Type;
        Routine_Name        : in     String;
        Intercepted_Routine : in     Callmon.Address;
        Jacket_Routine      : in     Callmon.Address);

    pragma Export_Procedure (
        Internal  => Interceptor,
        External  => "",
        Mechanism => (Reference, Value, Descriptor (S), Value, Value));
```

The pragma *Export_Procedure* is required by DEC Ada in that case because the *Address* attribute will be used for this procedure. However, to avoid a pollution of the name space, a empty string is used as external symbol (which directs the compiler to generate no symbol).

To intercept a routine, the following syntax must be used:

```
    Callmon.Intercept (Routine_Name => "LIB$GET_VM",
                       Pre_Routine  => Interceptor'Address,
                       Post_Routine => Interceptor'Address);
```

4.3.2 Disassembler output routine

In Ada, the user-written output routine for the disassembler must be declared the following way:

```
    procedure Output (
        PC        : in Callmon.Address;
        Opcode    : in String;
        Operands  : in String;
        User_Data : in Callmon.Address);
```

```
pragma Export_Procedure (
    Internal  => Output,
    External  => "",
    Mechanism => (Value, Descriptor (S), Descriptor (S), Value));
```

4.4 Linking an Ada application against CallMon

As stated previously, CallMon referencs a couple a system-defined data structures and must be linked against the OpenVMS base image, that is to say using the linker option /SYSEXE.

The command ACS LINK which is used to link Ada applications does not recognize this qualifier. However, since ACS LINK creates a sub-process into which the OpenVMS linker is invoked, it is possible to redefine the DCL command LINK to include this option:

```
$ LINK :== LINK/SYSEXE
$ ACS LINK main-unit-name
```

5. Using CallMon in a Multi-Threaded Environment

5.1 Reentrancy

The API of CallMon is not reentrant. This means that the routines like callmon$intercept or callmon$dump_routines cannot be called simultaneously in different threads. Moreover, during the time a routine is being intercepted (namely, during the execution of callmon$intercept), this routine is unsafe. Calling it in a different thread may produces unpredictable results (in other words, the most predictable result is a crash of the application...)

However, **the jacket routines of CallMon are reentrant**. This means that intercepting a reentrant routine does not break its reentrancy.

Of course, to keep the reentrancy of an intercepted routine, the user-written pre and post-processing routines must be reentrant as well.

5.2 Programming hints

In a multi-threaded application, always establish the routine interception from the main thread, before creating any other thread.

5.3 Programming hints with Ada tasking

On OpenVMS Alpha, Ada tasks are pthreads. Consequently, the above advice also applies to Ada applications with multiple tasks. The simplest way to do this in Ada is gathering all calls to the API of CallMon inside the elaboration part of one package.

Example:

```
with CALLMON;
package body INTERCEPTOR is
    ...
begin
    CALLMON.INTERCEPT (...);
    CALLMON.INTERCEPT (...);
    ...
end INTERCEPTOR;
```

However, it it somewhat difficult to control the elaboration order. If other packages contain static tasks, these tasks are also started during the elaboration of the application. To ensure that these tasks will start after calling the API of CallMon, use a pragma ELABORATE inside all packages which create tasks during their elaboration.

Example:

```
with INTERCEPTOR;
pragma ELABORATE (INTERCEPTOR);

package body WHICH_CREATES_TASKS is

    task WHICH_STARTS_DURING_ELABORATION is
        ...
```

6. Implementation of Routine Interception

This section explains how CallMon intercepts routines. Reading this section is not necessary to use CallMon. If you are not familiar with the structure of executable images and the calling standard of OpenVMS Alpha, you may skip this section.

For more details, you may read the following documents:

- *OpenVMS Calling Standard*, chapter « *OpenVMS Alpha Conventions* ».
- *OpenVMS Linker Utility Manual*, appendix « *Alpha Object Language* ».
- *OpenVMS Alpha Internals and Data Structures*, chapter « *Image Activation and Exit* ».

6.1 Overview of OpenVMS Alpha images

There are two kinds of image files in OpenVMS:

- **Executable images**. There is at most one active executable image in a process. This image is usually not relocatable. When the image was created, the linker assigned a fixed address to each location and resolved all internal references (*intra-image references*). The image is usually based at hexadecimal address 00010000.
- **Shareable images**. There are several active shareable images in a process. Shareable images are usually position-independent, or *relocatable.* They are mapped at different addresses in different processes. When a shareable image is created, the linker cannot resolve intra-image references. Consequently, it generates a *relocation table.* This table contains a list of all intra-image references. When a shareable image is activated inside a process, the *image activator* assigns a base address to the shareable image. Then, it reads the relocation table and modifies all intra-image references according to the base address.

For both kinds of images, another type of reference exists: the *inter-images references.* Such a reference exists when an image calls (or makes any kinds of reference to) another image. Only shareable images may be called. When image A references symbol S in image B, two kinds of data exist:

- **Inside image B**, there is a **definition** of symbol S. This definition is inside a special structure in image B: the *symbol vector.* Note that image B is necessarily a shareable image.
- **Inside image A**; there is **reference** to symbol S (more precisely, a reference to the offset of S inside the transfer vector of image B). Image A can be an executable or a shareable image. When the linker creates image A, it cannot resolve references to external images. Consequently, for each image which is used by image A, the linker generates a *fixup table* inside file A.EXE. This table contains, among others, a list of all references to image B. When image A is activated, the image activator first ensures that all images which are used by A are activated. Then, it reads the fixup table of A and modifies all references according to the base address of B.

6.2 Overview of OpenVMS Alpha calling standard

Some parts of the OpenVMS calling standard must be understood before going further.

Alpha is a register-based architecture. All instructions have a fixed size (32 bits). Consequently, an instruction cannot contains an address (64 bits). To access a data at a given location, the address of this data must have been loaded into a register and the instruction contains the name of the register to use, not the address of the data itself.

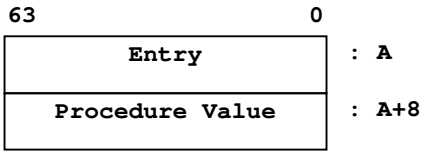
More precisely, data are grouped into sections. When a routine needs to access a data, it loads the base address of the section in a register and uses the offset of the data in the section. Example:

```
; Register R27 contains the base address of the section
; The data is a quadword at offset 64 in the section
LDQ    R0, 64(R27)    ; Load the data into register R0
```

By convention, each routine has a ***linkage section*** were all its useful constant data are stored (usually addresses of other data and routines). The OpenVMS calling standard specifies that the caller of a routine must load in register R27 the address of the linkage section of this routine before calling it. Upon entry, the routine may safely rely on the fact that R27 points to its linkage section and can consequently access any data the address of which is in the linkage section.

The address of the linkage section of a routine is called the ***procedure value*** of this routine. At the beginning of the linkage section, there is always a special structure called a ***procedure descriptor*** which contains various information about the procedure, including the address of its first instruction. This address is called the ***entry***. Since the procedure descriptor is always at the beginning of the linkage section, the procedure value is also the address of the procedure descriptor.

The caller of a routine must know both the entry and procedure value of this routine before calling it. It must load the procedure value into register R27 and branch to the entry. Usually, these two addresses are grouped into a ***linkage pair***:



When procedure P1 calls procedure P2, the linkage section of P1 contains a linkage pair for P2. The calling sequence is typically the following:

```
; assume that register R2 points to linkage section of P1
LDQ    R26, offset (R2)    ; load entry address of P2
LDQ    R27, offset+8 (R2)  ; load procedure value of P2
JSR    R26, (R26)          ; branch to address from R26 and
                           ; save return address in R26
```

6.3 How to intercept a routine

We want to intercept all calls to a given routine, whether this call is performed from the application (that we can rebuild) or from any shareable images (out of our control, like those which ship with OpenVMS).

There are two alternatives:

1. We let the application calls the routine at its normal address but we modify the instructions stream at the beginning of the intercepted routine. The modified sequence branches into CallMon.
   - Advantages: We are absolutely sure to trace all calls to the routine.
   - Drawbacks: When the jacket routine of CallMon calls the original routine, it must somehow rebuild the original instruction stream, which is not an obvious job.
2. We do not modify the executable code of the original routine but we modify all calls to it. The calls are modified such that the jacket routine of CallMon is called instead of the original routine.
   - Advantages: We preserve the integrity of the original code.

- Drawbacks: We must locate every occurrence of a call to this routine in all images in the process, either inside the image containing the routine and inside other images.

None of these methods is 100% safe. Intercepting a routine remains a big hack. In this version, CallMon implements the second method only. In a future release, CallMon will implement both methods and the application will choose the prefered one.

6.4 Modifying all references to a routine

Everywhere a routine is called, there is at least a reference to its procedure value and optionally a reference to its entry code. The last reference is optional because the address of the entry code is also stored in the procedure descriptor and can consequently be derived from the procedure value without direct reference.

In the virtual address space of the process, each time CallMon encounters a reference to either the procedure value or the entry code of the procedure to intercept, it replaces this reference with the procedure value or entry code of its jacket procedure.

The problem is locating all the references to the routine inside the whole virtual address space of the process. CallMon processes inter-image references (*fixups*) and intra-image references (*relocations*) differently.

6.4.1 Modifying fixups

Fixups are solely used for inter-image calls. Since the target address of an inter-image call is not known, neither by the compiler nor by the linker, the address is necessarily loaded from the linkage section of the caller. This location inside the linkage section is declared as a fixup.

Initially, the linkage pair in the linkage section of the caller does not contain any valid address. Instead, it contains an offset inside the symbol vector of the target image.

The caller image contains several fixup records, one for each shareable image which is used. In the fixup record for the target image, there is one entry which points to the linkage pair. When the caller image is activated, the image activator

- reads the fixup record,
- finds the entry,
- looks at the pointed location (ie. the linkage pair),
- interprets its content as an offset in the target symbol vector,
- reads the actual address of the target routine from the symbol vector,
- replaces the content of the pointed location (the linkage pair) by the actual address.

Basically, CallMon exactly behaves the same way. When a routine is going to be intercepted, CallMon assumes that all images have already been activated. Consequently, for all images, except the target image (the one containing the routine to intercept), CallMon reads the fixup records which refer to the target image. Each time a fixup element points to a location containing either the procedure value or the code entry of the target routine, CallMon replaces the content of this location by either the procedure value or code entry of the jacket routine.

Using this method, CallMon is certain to safely intercept 100% inter-image calls to the target routine.

6.4.2 Modifying relocations

Relocations are solely used for intra-image calls. Modifying relocations is much more complicated than modifying fixups. In a way, relocations are handled by the linker and the image activator pretty much the same way as fixups: there are relocations records, containing relocation entries, pointing to locations containing an offset from the start of the image. To intercept a routine which is referenced using relocations, CallMon uses a similar method than for fixups.

However, CallMon cannot do that safely on all relocations.

6.4.2.1 PC-relative branch

Unlike fixups, many intra-image calls to a specific target routine are not described by relocations. Since the target routine is located inside the same image, and maybe inside the same module, as the caller routine, the reference can be made using self-relative addressing mode.

We have seen, in a previous section, that most the calls are performed using the two fields of a linkage pair:

```
; assume that register R2 points to linkage section
LDQ     R26, offset (R2)     ; load entry address
LDQ     R27, offset+8 (R2)   ; load procedure value
JSR     R26, (R26)           ; branch to address from R26
```

However, if both routines are in the same module, the compiler can compute the displacement between the call instruction and the target entry code. In that case, it generates a sequence similar to:

```
LDQ     R27, offset+8 (R2)   ; load procedure value from linkage section
BSR     R26, displacement    ; branch to address at known displacement
```

If both routines are in different modules, the compiler generates the first sequence but it also includes in the object file some optimization hints. Using these hints, the linker can modify the object code if it determines that the displacement between the call instruction and the target entry (in two different modules) is small enough to fit into an instruction.

Generated by the compiler:

```
LDQ     R26, offset (R2)
LDQ     R27, offset+8 (R2)
JSR     R26, (R26)
```

Modified by the linker:

```
NOP
LDQ     R27, offset+8 (R2)
BSR     R26, displacement
```

If CallMon blindly modifies all relocations to the target procedure, it will modify the reference in the linkage section which contains the procedure value. This modified value will be used in the two above examples. However, since the branch is taken using a displacement, the result is a call to the code entry of the original routine with register R27 containing the base address of the linkage section of another routine (the jacket routine). This results in various nice effects, from access violation to data corruption.

Consequently, before modifying the relocations which point to a target routine, CallMon carefully examines all image sections containing executable code inside the target image. If any section contains a standard call to the target routine using PC-relative addressing (namely a `BSR R26, disp` instruction), then CallMon marks the routine as ***unrelocatable*** and will not modify any relocation pointing to it.

Consequently, a call to an unrelocatable routine will be traced only it the caller is located into another image. In other words, inter-image calls to unrelocatable routines are traced (because they use fixups) but intra-image calls are not (because they use relocations).

6.4.2.2 BSR replacement

In order to reduce the risk of losing intra-image calls, CallMon attempts to reduce the number of BSR instructions pointing to the target routine. If CallMon manages to remove all these BSR, the routine becomes relocatable again and its relocations may be modified. If at least one BSR pointing the target routine cannot be replaced, the routine remains unrelocatable and no relocation at all is modified.

Each time CallMon encounters a BSR call to the target routine, it attempts to replace it with a JSR using a relocatable reference to the target entry code. In a way, CallMon acts as a deoptimizer wherever the linker has acted as an optimizer.

How does it works? Before a standard call, register R27 must point to the base of the linkage section of the target. Since the beginning of this linkage section always contains the procedure

descriptor of the target routine, the address of the code entry can be fetched from this descriptor using an instruction like:

```
LDQ  R26, 8(R27)   ; entry address is at offset 8 in the procedure descriptor
```

CallMon can always replace the BSR with a JSR because all instructions have the same size. However, CallMon must find some room to insert the LDQ R26 instruction. Consequently, CallMon attempts to find a NOP instruction before the BSR. This is not a foolish quest since most BSR result from a linker optimization and we have seen that this kind of optimization leaves a NOP instruction.

If a NOP is found, depending on the position of the instruction loading something into R27, CallMon performs one of the following replacements.

Before replacement:

```
A1: LDx    R27, offset (Ry)
    ...
A2: NOP
    ...
    BSR    R26, displacement
```

After replacement:

```
A1: LDx    R27, offset (Ry)
    ...
A2: LDQ    R26, 8 (R27)
    ...
    JSR    R26, (R26)
```

CallMon must ensure that R27 has a valid value before using it. If the LDx R27 (LDA or LDQ) is located after the NOP and if register R27 is not used between the NOP and the LDx R27, CallMon moves the LDx R27:

Before replacement:

```
A1: NOP
    ...
A2: LDx    R27, offset (Ry)
    ...
    BSR    R26, displacement
```

After replacement:

```
A1: LDx    R27, offset (Ry)
    ...
A2: LDQ    R26, 8 (R27)
    ...
    JSR    R26, (R26)
```

Of course, in the two cases, if any instruction between address A1 and the BSR is a branch instruction or is the target of another branch instruction somewhere else, the replacement is too risky and CallMon leaves the BSR.

6.4.2.3 BSR depth

Even in the absence of branch target between A1 and the BSR, CallMon limits the number of instructions to seek before the BSR. This number is called the ***BSR depth***. If the number of instructions between address A1 and the BSR exceeds the BSR depth, the replacement is not performed.

By default, CallMon uses a BSR depth of 1000 instructions. This means that the seek is virtually unlimited since the probability to find 1000 contiguous instructions without branch target in between is very low.

The user may arbitrarily limit the BSR depth using the logical name CALLMON$BSR_DEPTH.

Example:

```
$ DEFINE CALLMON$BSR_DEPTH 10
$ RUN APPLICATION
```

In that case, if CallMon is unable to find a NOP in the 10 instructions preceeding a BSR, the BSR will remain in its original form.

6.4.2.4 BSR replacement policy

In some cases, modifying code sections may not be acceptable. For some reason, the application may fail because of this replacement. The BSR replacement policy may be modified using the logical name CALLMON$BSR_REPLACEMENT. Its translation must be a keyword from the following table.

Keyword	Description
OFF	No BSR replacement is performed. CallMon does not modify any code section. This is the safest replacement policy but it reduces the ability to trace intra-image calls. Use it in case of unexpected application failure.
IF_R27	A BSR is replaced only if CallMon can find the instruction which loads the address of the linkage section in R27. If the LDx R27 is not found, this may indicate that the compiler knew that the target routine does not need the address of its linkage section and did not load R27 with anything significant. In such a (rare) case, using 8(R27) to load the code entry would fail. IF_R27 is the default replacement policy.
ALL	A BSR is replaced even if the LDx R27 is not found. In that case, CallMon always relies on the rule of the OpenVMS Calling Standard and assumes that the compiler has already loaded the correct address in R27. If the LDx R27 instruction is not found in the BSR depth or since the last branch target, CallMon assumes that R27 was correctly loaded before. If a NOP is found, the NOP and the BSR are replaced.

Example:

```
$ DEFINE CALLMON$BSR_REPLACEMENT OFF
$ RUN APPLICATION
```

In that case, CallMon will keep all code sections intact.

6.5 Modifying the entry code of a routine

Note: This method is not yet implemented in the current version of CallMon.

This method requires the replacement of the first four instructions at the entry of the procedure to intercept. CallMon must first ensure that these four instructions meet the following requirements:

- None of them is a branch instruction.
- None of them is the target of a branch instruction.
- None of them uses register R28. We rely here on the fact that R28 is defined by the OpenVMS Alpha Calling Standard as a volatile scratch register across calls.

If these conditions are met (which is the case for the vast majority of routines), the first four instructions are replaced by the following sequence:

```
Original routine:  BR   R28, label   ; Load PC in R28
           label:  LDL  R28, 8(R28)  ; Load the content of target in R28
                   JMP  R31, (R28)   ; Branch to jacket routine entry
          target:  address-of-jacket-routine-entry (32 bits)
    continuation:  ...               ; 5th original instruction and more...
```

Of course, the original first four instructions were previously saved somewhere else for later execution.

When the intercepted routine is called by the application, the jacket routine is invoked with register R27 containing the address of the linkage section of the intercepted routine. Consequently, the jacket must first load in R27 the address of its own linkage section and performs its normal work, just as if it had been invoked by the caller of the intercepted routine.

When the jacket routine calls the original routine, it uses the following sequence:

```
                    LDQ  R27, procedure value of intercepted routine
```

```
                    LDQ  R28, address of 5th instruction in original routine
                    LDQ  R26, address of save area
                    JSR  R26, (R26)
```

This sequence branches into the area were the original first four instructions were saved:

```
          save-ares: original-instruction-1
                     original-instruction-2
                     original-instruction-3
                     original-instruction-4
                     JMP  R31, (R28)   ; branch to original-instruction-5
```

Note that we previously checked that the original first four instructions did not use the register R28, contained no branch instructions and were the target of no other branch instruction.

6.6 Predefined uninterceptable and unrelocatable routines

Some routines are uninterceptable or unrelocatable *by definition.*

6.6.1 Uninterceptable because of CallMon usage

Some routines cannot be intercepted because this would interfere with the normal behaviour of CallMon. The list of these routines is very short:

- LIB$GET_CURR_INVO_CONTEXT
- LIB$GET_PREV_INVO_CONTEXT
- LIB$GET_INVO_HANDLE
- OTS$DIV_UI

6.6.2 Uninterceptable and unrelocatable because of routine structure

Some routines, such as *strlen* in the DEC C Run-Time Library, do not use their linkage section. If the compiler knows that, it does not load the procedure value in R27 before calling the routine.

In the case of *strlen*, the header file *string.h* contains a pragma linkage which specifies that *strlen* does not need its argument information register (R25) and its linkage pointer (R27). When the DEC C compiler generates a call to *strlen*, it does not load anything in register R27.

If we attempt to replace the BSR to such a routine, we would load in register R26 a value at 8(R27). But, in that case, CallMon cannot safely assume that R27 contains something significant.

If we attempt to replace the relocations, the fixups or the first instructions of such a routine, the jacket routine in CallMon would be called without linkage section address in R27. Since the jacket routine needs its linkage section, this will not work.

The following routines were identified as generating non-standard calls, without R27. They are both predeclared as uninterceptable and unrelocatable:

- DECC$STRCMP (known as *strcmp* in the DEC C Run-Time Library).
- DECC$STRCPY (known as *strcpy* in the DEC C Run-Time Library).
- DECC$STRLEN (known as *strlen* in the DEC C Run-Time Library).
- DECC$STRNLEN (known as *strnlen* in the DEC C Run-Time Library).

6.6.3 Declaring additional routines

It is possible to declare additional uninterceptable routines using the logical name CALLMON$NO_INTERCEPTION. This logical name is a search list. Each element of the list is the name of a routine.

It is also possible to declare unrelocatable routines using another search list named CALLMON$NO_RELOCATION.

Example:

```
$ DEFINE CALLMON$NO_RELOCATION LIB$GETDVI, LIB$GETJPI
$ DEFINE CALLMON$NO_INTERCEPTION LIB$GETDVI, LIB$GETJPI, LIB$GETSYI
```

6.7 Tracing CallMon operations

In case of problem or to satisfy a legitimate curiosity, you can ask CallMon to display some trace messages. These messages indicate what CallMon is actually doing. They are activated using the logical name CALLMON$TRACE.

The logical name CALLMON$TRACE must be defined as a search list of keywords. Each keyword indicates a class of messages.

Example:

```
$ DEFINE CALLMON$TRACE IMAGE, ROUTINE
$ RUN APPLICATION
```

The list of keywords is summarized in the following table:

Keyword	Description
API	Reports all calls to the Application Programming Interface of CallMon, that is to say callmon$intercept, callmon$intercept_all, etc..
CONFIGURATION	Reports the actual values of the configurable options like the BSR depth or the BSR replacement policy.
DUPLICATE	Reports the duplication of procedure names or values, that is to say two procedures with same name in two different images or two names for the same procedure.
FIXUP	Reports all fixup modifications. Warning: this option may produce many trace messages.
INSTRUCTIONS	Reports all instructions processing, branch targets, BSR replacement, etc... For each code section in each image, a text file is generated. This file contains all instructions and replacements in the section. The file name is derived from the image name and code section number. For instance, the image section number 3 in LIBRTL.EXE will produce a file named LIBRTL_003.LIS. By default, the files are created in the current directory. To select another directory, define the logical name CALLMON$TRACE_DIRECTORY. Warning: This option may significantly increase the initialization time.
IMAGE	Reports all images as they are loaded by CallMon.
MEMORY	Reports each modification of the protection and ownership of a page of virtual memory. To modify the fixups, relocations and instructions, CallMon changes many protections.
REFERENCES	Reports a summary of the fixup and relocation modifications.
RELOCATION	Reports all relocation modifications. Warning: like FIXUP, this option may produce many trace messages.
ROUTINE	Reports all routines as they are loaded by CallMon.

Keyword	Description
UNRELOCATABLE	Reports all unrelocatable routines, that is to say the routines which are the targets of some BSR which could not be replaced.
UNRELOCATED	Reports all unrelocatable routines which are actually intercepted. This list may help to understand why a suspected call to an intercepted routine is not processed.