Chapter 30. Using preinitialization services

You can use preinitialization to enhance the performance of your application. Preinitialization lets an application initialize an HLL environment once, perform multiple executions using that environment, and then explicitly terminate the environment. Because the environment is initialized only once (even if you perform multiple executions), you free up system resources and allow for faster responses to your requests.

This topic describes the Language Environment-supplied routine, CEEPIPI, that provides the interface for preinitialized routines. Using CEEPIPI, you can initialize an environment, invoke applications, terminate an environment, and add an entry to the Preinitialization table (PreInit table). (The PreInit table contains the names and entry point addresses of routines that can be executed in the preinitialized environment.)

This topic also describes reentrancy considerations for a preinitialized environment, XPLINK considerations, user exit invocation, stop semantics, service routines, and an example of CEEPIPI invocation.

Before the introduction of a common runtime environment, introduced with Language Environment, some of the individual languages had their own form of preinitialization. This older form of preinitialization is supported by Language Environment, but it is not strategic. The following is a list of these older forms of preinitialization and some considerations for their use:

- C

  Language Environment supports the prior form of C preinitialization, through the use of an extended parameter list. For more information about this interface, see *z/OS XL C/C++ Programming Guide*.
- C++

  There is no prior form of preinitialization for C++.
- COBOL

  Language Environment supports the prior form of COBOL preinitialization, RTEREUS, ILBOSTP0, and IGZERRE. For more information about these interfaces, see the Enterprise COBOL for z/OS library (http://www-01.ibm.com/support/docview.wss?uid=swg27036733). This prior form of COBOL preinitialization cannot be used at the same time that Language Environment preinitialization is used.
- Fortran

  There is no prior form of preinitialization for Fortran.
- PL/I

  Language Environment supports the prior form of PL/I preinitialization, through the use of an Extended Parameter List. For more information about this interface, see the IBM Enterprise PL/I for z/OS library (http://www.ibm.com/support/docview.wss?uid=swg27036735). This prior form of PL/I preinitialization does not support PL/I multitasking applications.

Using preinitialization

From a non-Language Environment-conforming driver (such as assembler) you can use Language Environment preinitialization facilities to create and initialize a common run-time environment, execute applications written in a Language Environment-conforming HLL multiple times within the preinitialized environment, and terminate the preinitialized environment. Language Environment provides a preinitialized interface to perform these tasks.

In the preinitialized environment, the first routine to execute can be treated as either the main routine or a subroutine of that execution instance. Language Environment provides support for both of these types of preinitialized routines:

- Executing one main routine multiple times
- Executing subroutines multiple times

Language Environment preinitialization is commonly used to enhance performance for repeated invocations of an application or for a complex application where there are many repetitive requests and where fast response is required. For instance, if an assembler routine invokes either a number of Language Environment-conforming HLL routines or the same HLL routine a number of times, the creation and termination of that HLL environment multiple times is needlessly inefficient. A more efficient method is to create the HLL environment only once for use by all invocations of the routine.

The interface for preinitialized routines is a loadable routine called CEEPIPI. This routine is loaded as an RMODE(24) / AMODE(ANY) routine and returns in the AMODE of its caller when the request is satisfied.

CEEPIPI handles the requests and provides services for environment initialization, application invocation, and environment termination. All requests for services by CEEPIPI must be made from a non-Language Environment environment. (“Preinitialization interface” on page 486 contains a detailed description and information about how to invoke each of these services.) The parameter list for CEEPIPI is an OS standard linkage parameter list. Each request to CEEPIPI is identified by a function code that describes the CEEPIPI service and that is the first parameter in the parameter list. The function code is a fullword integer (for example, 1 = init_main, 2 = call_main).

The preinitialization services offered under Language Environment are listed in Table 68 on page 486. Preinitialization services do not support PL/I multitasking applications.

An example assembler program in “An example program invocation of CEEPIPI” on page 519 illustrates invocation of CEEPIPI for the function codes `init_sub`, `call_sub`, and `term`.

Using the PreInit table

Language Environment uses the PreInit table to identify the routines that are candidates for execution in the preinitialized environment, as well as optionally to load the routine when it is called. It is possible to have an empty PreInit table with no entries. The PreInit table contains the names and the entry point addresses of each routine that can be executed within the preinitialized environment. Candidate routines can be present in the table when the `init_main` or `init_sub` functions are invoked, or can be added to the table using (*add_entry*).

When the entry point address is supplied either as an entry in the initial PreInit table provided with initialization functions, or as specified on the add_entry function, the high order bit of the address must be set to indicate the addressing mode for the routine. If the high order bit is OFF the routine is called in 24 bit addressing mode and the address must be a valid 24 bit address. If the high order bit is ON the routine is called in 31 bit addressing mode and the address must be a valid 31 bit address.

C considerations

C routines that are the target of (*call_main*) or (*call_sub*) must be z/OS C routines.

- C main routines must be initialized with (init_main).
- C routines that are the target of (call_main) must contain a main().

C++ considerations

The preinitialization routines (*call_main*) or (*call_sub*) can support C++ applications.

- C++ main routines must be initialized with (init_main).
- C++ routines that are the target of (call_main) must contain a main().

COBOL considerations

COBOL programs that are the target of (*call_main*) or (*call_sub*) must be Enterprise COBOL for z/OS, COBOL for OS/390 & VM, COBOL for MVS & VM, or COBOL/370 programs.

Fortran considerations

Fortran routines cannot be the target of a CEEPIPI call.

PL/I considerations

PL/I routines that are the target of (call_main) or (call_sub) must be Enterprise PL/I for z/OS or PL/I for MVS & VM routines. OS PL/I Version 1 and OS PL/I Version 2 routines can run in the preinitialized environment only when called from PL/I routines that are the target of (call_main) or (call_sub).

Macros that generate the PreInit table

Language Environment provides the following assembler macros to generate the PreInit table for you: CEEXPIT, CEEXPITY, and CEEXPITS.

CEEXPIT: CEEXPIT generates a header for the PreInit table.

Syntax

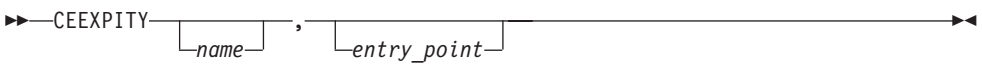
table_name
Assembler symbolic name assigned to the first word in the PreInit table. The address of this symbol should be used as the *ceexptbl_addr* parameter in a (init_main) or a (init_sub) call.

NOSTOR=ABEND
Indicates that the system is to issue an abend if it cannot obtain storage for the preinitialization environment. This is the default.

NOSTOR=RC
Indicates that the system is to issue a return code if it cannot obtain storage for the preinitialization environment.

CEEXPITY: CEEXPITY generates an entry within the PreInit table.

Syntax



name
The first eight characters of the load name of a routine that can be invoked within the Language Environment preinitialized environment.

entry_point
The address of the load module that is to be invoked, or 0, to indicate that the module is to be dynamically loaded.

The high-order bit of the entry_point address must be set to indicate the addressing mode for the routine. If the high-order bit is OFF, the routine is called in 24 bit addressing mode and the address must be a valid 24 bit address. If the high-order bit is ON, the routine is called in 31 bit addressing mode and the address must be a valid 31 bit address.

You have the option of specifying either, both, or neither of the parameters:

- If *name* is omitted and *entry_point* is present, the comma must be present.
- If both parameters are omitted, the entry is a candidate for assignment to the PreInit table by a call to (add_entry).
- If both parameters are present, *name* is ignored and *entry_point* is used as the start of the routine.

Each invocation of the CEEXPITY macro generates a row in the PreInit table. The first entry is row 0, the second is row 1, and so on.

CEEXPITS: CEEXPITS identifies the end of the PreInit table. This macro has no parameters.

Syntax

Reentrancy considerations

You can make multiple calls to main routines by invoking CEEPIPI services and making multiple requests from a single PreInit table. In general, you should specify only reentrant routines for multiple invocations, or you might get unexpected results.

For example, if you have a reentrant C main program that is invoked using (*call_main*) and that uses external variables, then when your routine is invoked again, the external variables are re-initialized. Multiple executions of a reentrant main routine are not influenced by a previous execution of the same routine.

However, if you have a nonreentrant C main program that is invoked using (*call_main*) and that uses external variables, then when your routine is invoked again, the external variables can potentially contain last-used values. Local variables (those contained in the object code itself) might also contain last-used values. If main routines are allowed to execute multiple times, a given execution of a routine can influence subsequent executions of the same routine.

If you are calling `init_sub`, `init_sub_dp`, or `add_entry` for C/C++, the routines can either be naturally reentrant or may be compiled RENT and made reentrant by using the z/OS C Prelinker Utility. If the subroutine is made reentrant using the z/OS C Prelinker Utility, multiple instances of the same subroutine are influenced by the previous instance of the same subroutine.

If you have a nonreentrant COBOL program that is invoked using (*call_main*), condition IGZ0044S is signaled when the routine is invoked again.

PreInit XPLINK considerations

Language Environment preinitialization services (PreInit) support programs that have been compiled XPLINK. Specifically, it allows programs and subroutines that have been compiled XPLINK to be defined in the PreInit table. The following guidelines are provided for this new option:

- XPLINK CEEPIPI subroutines must be fetchable. For C programs, this is done using the #pragma linkage (fetchable) statement. For more details on fetchable subroutines, refer to the documentation on fetch() in *z/OS XL C/C++ Runtime Library Reference*.
- Non-XPLINK PreInit programs can run in an XPLINK PreInit environment, but there may be performance degradation since non-XPLINK programs will be required to execute linkage-switching glue code. If possible, consider having separate PreInit environments for running XPLINK and non-XPLINK programs.
- If a PreInit environment has been initialized as a non-XPLINK environment and either the main() function is XPLINK or the XPLINK(ON) runtime option has been specified, then the PreInit environment will be rebuilt as an XPLINK environment. This is a one-time occurrence that can not be undone.

Creating an XPLINK environment versus a non-XPLINK environment

When initializing a PreInit environment, you can select to create an XPLINK or a non-XPLINK environment. There are four methods used to initialize a PreInit environment; *init_main*, *init_main_dp*, *init_sub*, and *init_sub_dp*. In each case, a token of the preinitialized environment is passed back to the customer PreInit driver program. This token ID is used and passed as input when executing PreInit programs. The following rules will determine if the initialized PreInit environment will be XPLINK or non-XPLINK. You can make a one-time dynamic change in the PreInit environment from non-XPLINK to XPLINK by using (call_main) to an XPLINK main().

init_main: (Input: PreInit table pointer, no runtime options are passed as input)

- If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.

- If the first program in the PreInit table is a non-XPLINK program, then a non-XPLINK environment will be initialized.
- If the PreInit table is empty at initialization time, then a non-XPLINK environment will be initialized.

init_main_dp: (Input: PreInit table pointer, no runtime options are passed as input)

- If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.
- If the first program in the PreInit table is a non-XPLINK program, then a non-XPLINK environment will be initialized.
- If the PreInit table is empty at initialization time, then a non-XPLINK environment will be initialized.

init_sub: (Input: PreInit table pointer, and runtime options)

- If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.
- If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.
- If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.

Note:

1. The runtime options you specify will apply to all of the subroutines that are called by (call_sub) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.
2. If this is a non-XPLINK sub environment, then do not allow an XPLINK subroutine to be added to the table.

init_sub_dp: (Input: PreInit table pointer, and runtime options)

- If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.
- If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.
- If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.

Note: The runtime options you specify will apply to all of the subroutines that are called by (call_sub) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.

User exit invocation

User exits are invoked for initialization and termination during calls to CEEPIPI as shown in Table 67 on page 485.

Table 67. Invocation of user exits during process and enclave initialization and termination

Function	When invoked
Assembler user exit for first enclave initialization	• (*init_sub*) • (*init_sub_dp*) • (*call_main*) • (*call_sub*) or (*call_sub_addr*) or (*call_sub_addr*) ended with stop semantics (see "Stop semantics")
HLL user exit	• (*init_sub*) • (*init_sub_dp*) • (*call_main*) • (*call_sub*) or (*call_sub_addr*) or (*call_sub_addr*) ended with stop semantics
C `atexit()` functions	• (*call_main*) • (*call_sub*) or (*call_sub_addr*), which ended stop semantics. • (*term*) for environment created with (*init_sub*) or (*init_sub_dp*), if the last (*call_sub*) or (*call_sub_addr*) did not end with stop semantics
Assembler user exit for first enclave termination	• (*call_main*) • (*call_sub*) or (*call_sub_addr*), which ended stop semantics • (*term*) for environment created with (*init_sub*) or (*init_sub_addr*) if the last (*call_sub*) or (*call_sub_addr*) did not end with stop semantics
Assembler user exit for process termination	• (*term*)

For main environments:

The CEEBXITA assembler user exit and CEEBINT HLL user exit that are used with the environment are taken from the main routine being called.

For sub environments:

The CEEBXITA assembler user exit and CEEBINT HLL user exit that are used with the environment are taken from the first entry in the PreInit table. Any occurrences of CEEBXITA or CEEBINT in any other PreInit table entries, or in load modules used for call_sub_addr-type calls, are ignored.

See Chapter 28, "Using runtime user exits," on page 419 for more information about user exits.

Stop semantics

When one of the following is issued within the preinitialized environment for subroutines:

- C `exit()`, `abort()`, or signal handling function specifying a normal or abnormal termination
- COBOL STOP RUN statement
- PL/I STOP or EXIT

or when an unhandled condition causes termination of the (only) thread, the logical enclave is terminated. The process level of the environment is retained. Language Environment does *not* delete those entries that were loaded explicitly by Language Environment during the preinitialization processing.

Attention: If the first entry in the PreInit table is either different or deleted from when the enclave was last initialized, the assembler user exit (CEEBXITA), HLL user exit (CEEBINT), or programmer default runtime options (CEEUOPT) used during either an enclave reinitialization or enclave termination will either be different or not available. This will result in unpredictable results. Therefore, when using PreInit subroutine environments and in order to keep consistent enclave initialization and termination behavior, users need to ensure the first valid entry in the PreInit table does not change, especially when it contains the aforementioned external references.

Preinitialization interface

The following section describes how to invoke the PreInit interface, CEEPIPI, to perform the following tasks:

- Initialization
- Application invocation
- Termination
- Addition of an entry to the PreInit table
- Deletion of a main entry from the PreInit table
- Identification of an entry in the PreInit table
- Access to the CAA user word

The PreInit services offered under Language Environment using CEEPIPI are listed in Table 68.

Table 68. Preinitialization services accessed using CEEPIPI

Function code	Integer value	Service performed
Initialization		
init_main	1	Create and initialize an environment for multiple executions of main routines.
init_main_dp	19	Create and initialize an environment for multiple executions of main routines.
init_sub	3	Create and initialize an environment for multiple executions of subroutines.
init_sub_dp	9	Create and initialize an environment for multiple executions of subroutines.
Application invocation		
call_main	2	Invoke a main routine within an already initialized environment.
call_sub	4	Invoke a subroutine within an already initialized environment.
start_seq	7	Start a sequence of uninterruptable calls to a number of subroutines.
call_sub_addr	10	Invoke a subroutine by address within an already initialized environment.
Termination		

Table 68. Preinitialization services accessed using CEEPIPI (continued)

Function code	Integer value	Service performed
term	5	Explicitly terminate the environment without executing a user routine.
end_seq	8	Terminate a sequence of uninterruptable calls to a number of subroutines.
Addition of an entry to PreInit table		
add_entry	6	Dynamically add a candidate routine to execute within the preinitialized environment.
Deletion of an entry from PreInit table		
delete_entry	11	Delete an entry from the PreInit table, making it available for subsequent *add_entry* functions.
Identification of a PreInit table entry		
identify_entry	13	Identify the programming language of an entry in the PreInit table.
identify_attributes	16	Identify the attributes of an entry in the PreInit table.
Identification of the environment		
identify_environment	15	Identify the environment that was preinitialized.
Access to the CAA user word		
set_user_word	17	Set value to be used to initialize CAA user word.
get_user_word	18	Get value to be used to initialize CAA user word.

Initialization

Language Environment supports four forms of preinitialized environments. The first supports the execution of main routines. The second is a special form of the first, that allows multiple preinitialized environments, for executing main routines. to be created within the same address space. The third supports the execution of subroutines. The fourth is a special form of the third, that allows multiple preinitialized environments, for executing subroutines, to be created within the same address-space.

The primary difference between these environments is the amount of Language Environment initialization (and termination) that occurs on each application invocation call. With an environment that supports main routines, most of the application's execution environment is reinitialized with each invocation. With an environment that supports subroutines, very little of the execution environment is reinitialized with each invocation. This difference has its advantages and disadvantages.

For the **main environment**, the advantages are:

- A new, pristine environment is created.
- Runtime options can be specified for each application.

and the disadvantages are:

- Poorer performance.

For the **subenvironment**, the advantages are:

- Best performance.

and the disadvantages are:

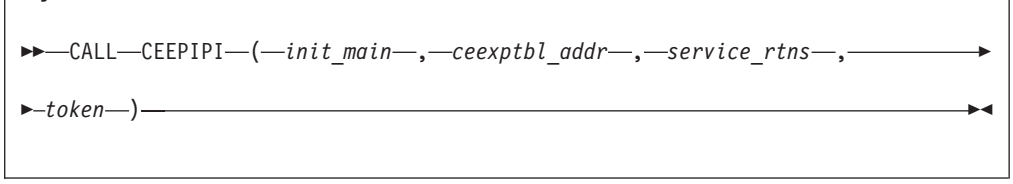
- The environment is left in what ever state the previous application left it in.
- Runtime options cannot be changed.

(init_main) — initialize for main routines

The invocation of this routine:

- Creates and initializes a new common run-time environment (process) that allows the execution of main routines multiple times
- Sets the environment to dormant so that exceptions are percolated out of it
- Returns a token identifying the environment to the caller
- Returns a code in register 15 indicating whether an environment was successfully initialized

Syntax



init_main **(input)**
A fullword function code (integer value = 1) containing the init_main request.

ceexptbl_addr **(input)**
A fullword containing the address of the PreInit table to be used during initialization of the new environment. Language Environment does not alter the user-supplied copy of the table. If an entry address is zero and the entry name is non-blank, Language Environment searches for the routine (in the LPA, saved segment, or nucleus) and dynamically loads it. Language Environment places the entry address in the corresponding slot of a Language Environment-maintained table.

Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, Language Environment uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns **(input)**
A fullword containing the address of the service routine vector or 0, if there is no service routine vector. See “Service routines” on page 512 for more information.

token **(output)**
A fullword containing a unique value used to represent the environment. The *token* should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

Return codes: Register 15 contains a return code indicating if an environment was successfully initialized. Possible return codes (in decimal) are:

0 A new environment was successfully initialized.

4 The function code is not valid.

8 All addresses in the table were not resolved. This can occur if a LOAD

failure was encountered or a routine within the table was generated by a non-Language Environment-conforming HLL.

12 Storage for the preinitialization environment could not be obtained.

16 CEEPIPI was called from an active environment.

32 An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

Usage notes:

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the main routine being called through call_main.
- If a program in the PreInit table failed to load (return code 8), the *identify_attributes* CEEPIPI function can be used to help determine what table entry address did not resolve.

XPLINK considerations:

- If the environment being initialized is to be an XPLINK environment then the first program in the PreInit table must be an XPLINK module.
- If there is no entry in the PreInit table or if the first module is a non-XPLINK program, a non-XPLINK environment will be initialized.
- It is possible to change the environment from a non-XPLINK to an XPLINK environment when doing a call_main. For more details, see call_main.

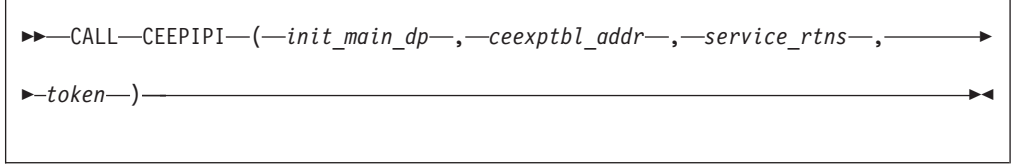
(init_main_dp) — initialize for main routines (multiple environment)

The invocation of this routine:

- Creates and initializes a new common run-time environment (process) that allows the execution of main routines multiple times.
- Sets the environment dormant so that exceptions are percolated out of it.
- Returns a token identifying the environment to the caller.
- Returns a code in register 15 indicating whether an environment was successfully initialized.
- Ensures that the environment tolerates the existence of multiple Language Environment processes or enclaves.

Note: Multiple main environments can be established by using (init_main_dp), as opposed to using (init_main), which can establish only a single environment.

Syntax



init_main_dp **(input)**
A fullword function code (integer value = 19) containing the (init_main_dp) request.

ceexptbl_addr **(input)**
A fullword containing the address of the PreInit table to be used during initialization of the new environment. A user-supplied copy of the table is not altered. If an entry address is zero and the entry name is non-blank, a search is performed for the routine (in the LPA, saved segment, or nucleus) and the routine is dynamically loaded. An entry is placed in the corresponding slot of a Language Environment-maintained table.

The high-order bit of the entry address determines what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, the AMODE returned by the system loader is used. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns **(input)**
A fullword containing the address of the service routine vector or 0, if there is no service routine vector. See "Service routines" on page 512 for more information.

token **(output)**
A fullword containing a unique value used to represent the environment. The *token* should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

Return codes: Register 15 contains a return code indicating if an environment was successfully initialized. Possible return codes (in decimal) are:

0 A new environment was successfully initialized.

4 The function code is not valid.

8 All addresses in the table were not resolved. This can occur if a LOAD failure was encountered or a routine within the table was generated by a non-Language Environment-conforming HLL.

12 Storage for the preinitialization environment could not be obtained.

16 CEEPIPI was called from an active environment other than a CEEPIPI main_dp environment.

32 An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

Usage notes:

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the main routine being called through (call_main).
- If a program in the PreInit table failed to load (return code 8), the (identify_attributes) CEEPIPI function can be used to help determine what table entry address did not resolve.
- If the process ID needs to be the same for all programs called by (call_main), the preinitialization driver program should pre-dub the task (TCB) before performing (init_main_dp).
- MSGFILE output can be directed to either a spool or to a unique file.
- Language Environment resources are not shared across multiple environments.
- C memory files are not shared across multiple environments.
- Calling POSIX(ON) programs in an (init_main_dp) environment is not supported.

XPLINK considerations:

- If the environment being initialized is to be an XPLINK environment then the first program in the PreInit table must be an XPLINK module.
- If there is no entry in the PreInit table or if the first module is a non-XPLINK program, a non-XPLINK environment will be initialized.
- It is possible to change the environment from a non-XPLINK to an XPLINK environment when using (call_main). For more information, see "(call_main) — invocation for main routine" on page 496.

Nested main_dp environment considerations:

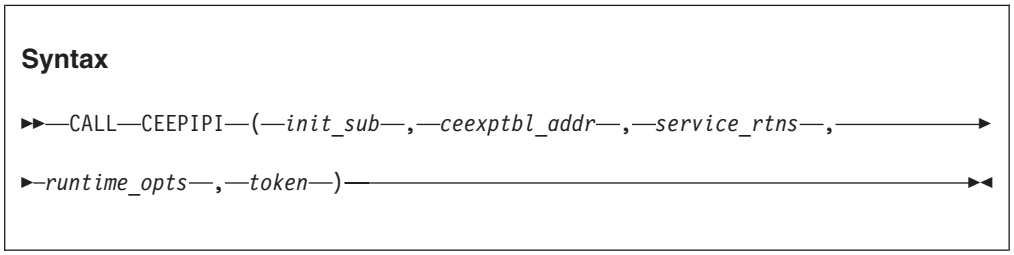
- Main_dp environments can be initialized by calling CEEPIPI(init_main_dp) from an active main_dp environment. From an active main_dp environment, nested calls to CEEPIPI can be made with a token returned from (init_main_dp) to perform certain other functions:
  - (call_main)
  - (add_entry)
  - (delete_entry)
  - (term)
  - (set_user_word)
  - (get_user_word)
  - (identify_entry)
  - (identify_environment)
  - (identify_attributes)
- Restrictions for nested main_dp environments:
  - When the calling environment has a user-provided @EXCEPRTN, the nested main_dp environment must also have a user-provided @EXCEPRTN.
  - If the user-written preinitialization driver program has established a SPIE or ESPIE routine, the nested main_dp environment must have a user-provided @EXCEPRTN.
  - All CEEPIPI calls that use a token must be made from the same TCB.
  - The INTERRUPT(ON) runtime option is not supported when using nested main_dp environments under TSO/E.
  - When the TRAP runtime option is used with nested main_dp environments, use of the TSO/E attention key is not supported.
  - If an ABEND (40XX, for example) causes the immediate ending of a nested main_dp environment without orderly Language Environment termination, the user-provided preinitialization driver program cannot be returned to. The calling main_dp environment will also end without orderly Language Environment termination
  - If the ABTERMENC(ABEND) runtime option is in effect and an unhandled condition causes a nested main_dp environment to ABEND, Language Environment will not return to the preinitialization assembler driver program. The calling main_dp environment will also ABEND without orderly Language Environment termination. Consider using ABTERMENC(RETCODE) in nested main_dp environments
  - If a main_dp environment which uses the TRAP(ON,SPIE) runtime option does (call_main) to a nested main_dp environment which uses TRAP(ON,NOSPIE), language environment issues an ESPIE macro to prevent program checks from being passed to any existing ESPIE routine. If this ESPIE call must be avoided, do not call a nested main_dp environment with TRAP(ON,NOSPIE) from a main_dp environment that uses TRAP(ON,SPIE).

(init_sub) — initialize for subroutines

The invocation of this routine:

- Creates and initializes a new common run-time environment (process and enclave) that allows the execution of subroutines multiple times
- Sets the environment dormant so that exceptions are percolated out of it
- Returns a token identifying the environment to the caller
- Returns a code in register 15 indicating whether an environment was successfully initialized
- Ensures that when the environment is dormant, it is immune to other Language Environment enclaves that are created or terminated

Syntax



init_sub **(input)**
A fullword function code (integer value = 3) containing the init_sub request.

ceexptbl_addr **(input)**
A fullword containing the address of the PreInit table to be used during initialization of the new environment. Language Environment does not alter the user-supplied copy of the table. If an entry address is zero and the entry name is non-blank, Language Environment searches for the routine (in the LPA, saved segment, or nucleus) and dynamically loads it. Language Environment then places the entry address in the corresponding slot of a Language Environment-maintained table.

Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, Language Environment uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns **(input)**
A fullword containing the address of the service routine vector. It contains 0 if there is no service routine vector. See “Service routines” on page 512 for more information.

runtime_opts **(input)**
A fixed-length 255-character string containing runtime options (see *z/OS Language Environment Programming Reference* for a list of runtime options that you can specify).

Note:

1. The runtime options you specify will apply to all of the subroutines that are called by the (call_sub) function. This includes options such as POSIX. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.
2. If the Language Environment being initialized is a non-XPLINK environment, then all of your subroutines must be non-XPLINK subroutines.

token **(output)**
A fullword containing a unique value used to represent the environment. The *token* should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

Return codes: Register 15 contains a return code indicating if an environment was successfully initialized. Possible return codes (in decimal) are:

0 A new environment was successfully initialized.
4 The function code is not valid.
8 All addresses in the table were not resolved. This can occur if a LOAD failure was encountered, a routine within the table was not generated by a Language Environment-conforming HLL, or a C or PL/I routine within the table was not fetchable.
12 Storage for the preinitialization environment could not be obtained.
16 CEEPIPI was called from an active environment.
32 An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.
40 An entry in the PreInit table is an XPLINK subroutine and the environment is a non-XPLINK sub environment. This entry is not valid.

Usage notes:

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the first valid entry in the PreInit table. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PreInit table entries are ignored. Unpredictable results will occur if this first entry is deleted or changed.
- If a program in the PreInit table failed to load (return code 8 or 40), the *identify_attributes* CEEPIPI function can be used to help determine what table entry address did not resolve.

XPLINK considerations:

- If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.
- If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.
- If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.

Note:

1. The runtime options you specify will apply to all of the subroutines that are called by (call_sub) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.
2. If this is a non-XPLINK sub environment, then do not allow an XPLINK subroutine to be added to the table.

(init_sub_dp) — initialize for subroutine (multiple environment)

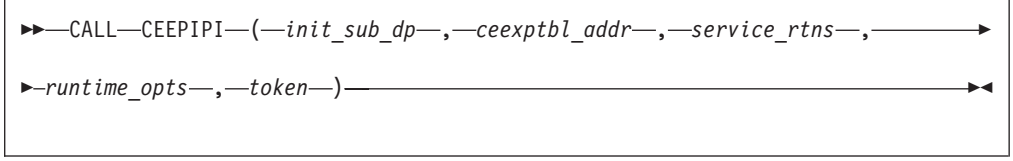
The invocation of this routine:

- Creates and initializes a new Language Environment process and enclave to allow the execution of subroutines multiple times
- Sets the environment dormant so that exceptions are percolated out of it
- Returns a token identifying the environment to the caller

- Returns a code in register 15 indicating whether an environment was successfully initialized
- Ensures that the environment tolerates the existence of multiple Language Environment enclaves
- Ensures that when the environment is dormant, it is immune to other Language Environment enclaves that are created or terminated

Multiple environments can be established only by using (init_sub_dp) as opposed to (init_sub), which can establish only a single environment.

Syntax



init_sub_dp **(input)**
A fullword function code (integer value = 9) containing the *init_sub_dp* request.

ceexptbl_addr **(input)**
A fullword containing the address of the PreInit table to be used during initialization of the new environment. Language Environment does not alter the user-supplied copy of the table. If an entry address is zero and the entry name is non-blank, Language Environment searches for the routine (in the LPA, saved segment, or nucleus) and dynamically loads it. Language Environment then places the entry address in the corresponding slot of a Language Environment-maintained table.

Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the entry address is zero, and the entry name is supplied, Language Environment uses the AMODE returned by the system loader. If the entry address is supplied, you must provide the AMODE in the high-order bit of the address.

service_rtns **(input)**
A fullword containing the address of the service routine vector. It contains 0 if there is no service routine vector. See “Service routines” on page 512 for more information.

runtime_opts **(input)**
A fixed-length 255-character string containing runtime options (see *z/OS Language Environment Programming Reference*for a list of runtime options that you can specify).

Note:

1. The runtime options you specify will apply to all of the subroutines that are called by the (call_sub) function. This includes options, such as POSIX. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.
2. If you want to run XPLINK routines in a PreInit sub environment, you must specify the XPLINK(ON) runtime option field when you create the sub environment by calling CEEPIPI(init_sub). You can not run XPLINK routines in a sub environment when runtime option XPLINK(OFF) is in effect.

token **(output)**
A fullword containing a unique value used to represent the environment. The *token* should be used only as input to additional calls to CEEPIPI, and should not be altered or used in any other manner.

Return codes: Register 15 contains a return code indicating if an environment was successfully initialized. Possible return codes (in decimal) are:

0 A new environment was successfully initialized.
4 The function code is not valid.
8 All addresses in the table were not resolved. This can occur if a LOAD failure was encountered or a routine within the table was not generated by a Language Environment-conforming HLL.
12 Storage for the preinitialization environment could not be obtained.
32 An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.
40 An entry in the PreInit table is an XPLINK subroutine and the environment is a non-XPLINK sub environment. This entry is not valid.

Usage notes:

- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the first valid entry in the PreInit table. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PreInit table entries are ignored. Unpredictable results will occur if this first entry is deleted or changed.
- COBOL, PL/I, and C routines must be compiled RENT to participate in this environment
- You can direct MSGFILE output to either a spool or to a unique file.
- C memory files are not shared across multiple environments.
- If the (init_sub_dp,...) interface is used to create additional environments, neither the existing environment, nor the one trying to be created can be POSIX(ON).
- If a program in the PreInit table failed to load (return code 8 or 40), the *identify_attributes* CEEPIPI function can be used to help determine what table entry address did not resolve.

XPLINK considerations:

- If the first program in the customer PreInit table is an XPLINK program, then an XPLINK environment will be initialized.
- If the runtime options are passed as input and the XPLINK option is specified as XPLINK(ON), then an XPLINK environment will be initialized.
- If neither of the above are true (the first program in the customer PreInit table is a non-XPLINK program and the XPLINK runtime option is off or not specified), then a non-XPLINK environment will be initialized.

Note: The runtime options you specify apply to all of the subroutines that are called by (call_sub_dp) function. This includes options such as XPLINK. Therefore, all of your subroutines must have the same characteristics and requirements needed for these runtime options.

Application invocation

Language Environment provides facilities to invoke either a main routine or subroutine. When invoking main routines, the environment must have been

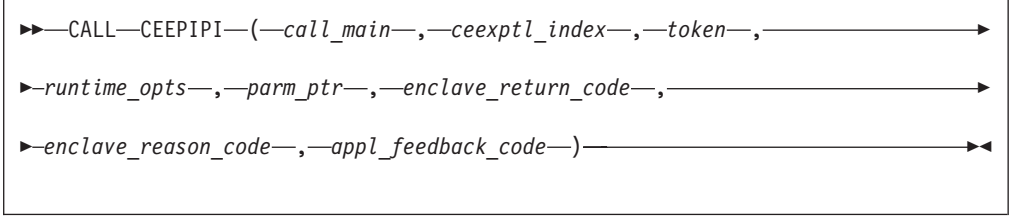
initialized using the init_main or init_main_dp function code. Similarly, when invoking subroutines, the environment must have been initialized with the init_sub or init_sub_dp function codes.

(call_main) — invocation for main routine

This invocation of CEEPIPI invokes as a main routine the routine that you specify. The common execution environment identified by *token* is activated before the called routine is invoked, and after the called routine returns, the environment is dormant.

At termination, the currently active HLL event handlers are driven to enforce language semantics for the termination of an application such as closing files and freeing storage. The process level is made dormant rather than terminated. The thread and enclave levels are terminated. The assembler user exit is driven with the function code for first enclave termination. (For more information about user exits, see Chapter 28, "Using runtime user exits," on page 419.)

Syntax



call_main **(input)**
A fullword function code (integer value = 2) containing the call_main request.

ceexptbl_index **(input)**
A fullword containing the row number within the PreInit table of the entry that should be invoked. The index starts at 0.

Each invocation of the CEEXPITY macro generates a row in the PreInit table. The first entry is row 0, the second is row 1 and so on. A call to (add_entry) to add an entry to the PreInit table also returns a row number in the *ceexptbl_index* parameter.

token **(input)**
A fullword with the value of the token returned by (init_main) or (init_main_dp) when the common run-time environment is initialized. The *token* must identify a previously preinitialized environment that is not active at the time of the call.

runtime_opts **(input)**
A fixed-length 255-character string containing runtime options. (See *z/OS Language Environment Programming Reference* for a list of runtime options that you can specify.)

parm_ptr **(input)**
A fullword parameter list pointer or 0 (zero) that is placed in register 1 when the main routine is executed. The parameter list that is passed must be in a format that HLL subroutines expect (for example, in an `argc, argv` format for C routines).

enclave_return_code **(output)**
A fullword containing the enclave return code returned by the called routine

when it finished executing. For more information about return codes, see "Managing return codes in Language Environment" on page 151.

enclave_reason_code **(output)**
A fullword containing the enclave reason code returned by the environment when the routine finished executing. For more information about reason codes, see "Managing return codes in Language Environment" on page 151.

appl_feedback_code **(output)**
A 96-bit condition token indicating why the application terminated.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The environment was activated and the routine called.
4 The function code is not valid.
8 If *token* was initialized by (init_main) or (init_sub), CEEPIPI(call_main) was called from a Language Environment-conforming HLL.

If *token* was initialized by (init_main_dp), CEEPIPI(call_main) was called from a Language Environment-conforming HLL that is not running in a (main_dp) environment, or token is already in use for another call to CEEPIPI.
12 The indicated environment was initialized for subroutines. No routine was executed.
16 The *token* is not valid.
20 The index points to an entry that is not valid or empty.
24 The index that was passed is outside the range of the table.
32 An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

The user return code and Language Environment return code modifier are set to zero before invoking the target routine.

Usage notes:

- The NOEXECOPS and CBLOPTS runtime options (see *z/OS Language Environment Programming Reference*) are ignored since the parameter inbound to the application and the runtime options are separated already. Therefore, NOEXECOPS and CBLOPTS do not affect the parameter string format. See "C PLIST and EXECOPS Interactions" on page 563 for more information.
- The assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and programmer defaults (CEEUOPT) that are used to initialize the environment are taken from the main routine being called. Any occurrences of CEEBXITA, CEEBINT, and CEEUOPT in other PreInit table entries are ignored.
- For more information about return codes, see "Managing return codes in Language Environment" on page 151.

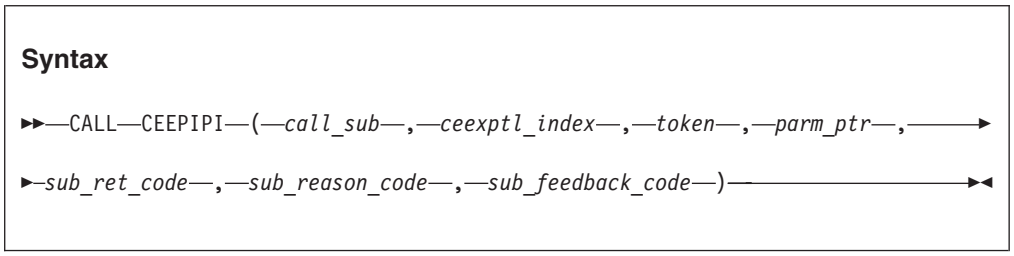
(call_sub) — invocation for subroutines

This invocation of CEEPIPI invokes as a subroutine the routine that you specify. The common run-time environment identified by *token* is activated before the called routine is invoked, and after the called routine returns, the environment is dormant.

The enclave is terminated when an unhandled condition is encountered or a STOP statement is executed. (See "Stop semantics" on page 485 for more information.)

However, the process level is maintained. The next call to (call_sub) initializes a new enclave.

Syntax



call_sub **(input)**
A fullword function code (integer value = 4) containing the call_sub request for a subroutine.

ceexptbl_index **(input)**
A fullword containing the row number of the entry within the PreInit table that should be invoked; the index starts at 0.

Note: If the token pointing to the previously preinitialized environment is a non-XPLINK environment and the subprogram to be invoked is XPLINK, then a Return Code of 40 will be returned because this is not valid.

token **(input)**
A fullword with the value of the token returned when the common run-time environment is initialized. This token is initialized by the (init_sub) or (init_sub_dp). The *token* must identify a previously preinitialized environment that is not active at the time of the call. You must not alter the value of the token.

Note: If the token pointing to the previously preinitialized environment is a non-XPLINK environment and the subprogram to be invoked is XPLINK a Return Code of 40 will be returned because this is not valid.

parm_ptr **(input)**
A parameter list pointer or 0 (zero) that is placed in register 1 when the routine is executed.

C and C++ users need to follow the subroutine linkage convention for C/C++ — assembler ILC applications, as described in *z/OS XL C/C++ Programming Guide*.

sub_ret_code **(output)**
The subroutine return code. If the enclave is terminated due to an unhandled condition, a STOP statement, or EXIT statement (or an exit() function), this contains the enclave return code for termination.

sub_reason_code **(output)**
The subroutine reason code. This is 0 for normal subroutine returns. If the enclave is terminated due to an unhandled condition, a STOP statement, or EXIT statement (or an exit() function), this contains the enclave reason code for termination.

sub_feedback_code **(output)**
The feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an unhandled condition, a STOP statement, or EXIT statement (or an exit() function), this contains the enclave feedback code for termination.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The environment was activated and the routine called.

4 The function code is not valid.

8 CEEPIPI was called from a Language Environment-conforming HLL.

12 The indicated environment was initialized for main routines. No routine was executed.

16 The *token* is not valid.

20 The index points to an entry that is not valid or empty.

24 The index passed is outside the range of the table.

28 The enclave was terminated but the process level persists.

This value indicates the enclave was terminated while the process was retained. This can occur due to a STOP statement being issued or due to an unhandled condition. The *sub_ret_code*, *sub_reason_code*, and *sub_feedback_code* indicate this action.

40 The subprogram was an XPLINK program and the preinitialized environment is non-XPLINK. This is not valid.

Usage notes:

- The enclave terminates if the subroutine issues a STOP statement, EXIT statement (or an exit() function), or if there is an unhandled condition. However, the process level is not terminated. When the enclave level is terminated, any subsequent invocation creates a new enclave by using the same runtime options used in the creation of the first enclave. Language Environment does not delete any user routines that were loaded into the PreInit table.

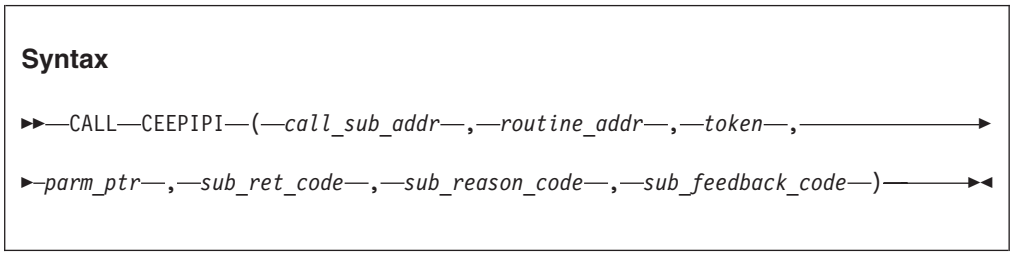
  However, if, the first valid entry in the PreInit table is different than when the enclave was last initialized, the assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and/or programmer default runtime options (CEEUOPT) used during the enclave re-initialization might be different. PreInit subroutine initialization uses these external references only when associated with the first valid entry in the PreInit table. Therefore, when using PreInit subroutine environments and you want consistent enclave initialization behavior across the stop semantics, you need to ensure the first valid entry in the PreInit table does not change, especially when it contains the aforementioned external references.(See “Stop semantics” on page 485.)
- Any subroutine that modifies external data cannot make assumptions about the initial state of that external data. The initial state of the external data is influenced by previous instances of the same subroutine and also by previous instances of any subroutine that caused enclave termination.
- If the first entry in the PreInit table contained a CEEBXITA, CEEBINIT or CEEUOPT when the environment was initialized and is then deleted or changed, the results of subsequent enclave re-initialization or termination is unpredictable. It is the responsibility of the user to ensure the first entry in the PreInit table does not change, especially when it contains the aforementioned external references.

(call_sub_addr) — invocation for subroutines by address

This invocation of CEEPIPI invokes a specified routine as a subroutine. The common run-time environment identified by *token* is activated before the called routine is invoked; after the called routine returns, the environment is dormant.

The enclave is terminated when an unhandled condition is encountered or a STOP or EXIT statement (or an exit() function) is executed. (See "Stop semantics" on page 485 for more information.) However, the process level is maintained; only the enclave level terminates.

Syntax



call_sub_addr **(input)**
A fullword function code (integer value = 10) containing the call_sub request for a subroutine.

routine_addr **(input/output)**
A doubleword containing the address of the routine that should be invoked. The first fullword contains the entry point address.

Note:

1. If this is an XPLINK environment and the second fullword is zero, Preinitialization services will create a new function pointer to call the routine directly. The new function pointer will be returned in the second fullword.
2. If this is an XPLINK environment and the second fullword is a function pointer, the XPLINK subroutine is called directly. This fast path avoids the overhead of translating the routine address to the function pointer.

token **(input)**
A fullword with the value of the token returned by (init_sub) or (init_sub_dp) when the common run-time environment is initialized. The *token* must identify a previously preinitialized environment that is not active at the time of the call. You must not alter the value of the token.

Note: If the token pointing to the previously preinitialized environment is a non-XPLINK environment and the subprogram to be invoked is XPLINK, then a return code of 40 will be returned because this is not valid.

parm_ptr **(input)**
A parameter list pointer or 0 (zero) that is placed in register 1 when the routine is executed.

C and C++ users are advised to follow the subroutine linkage convention for C/C++ — assembler ILC applications, as described in *z/OS XL C/C++ Programming Guide*.

sub_ret_code **(output)**
The subroutine return code. If the enclave is terminated due to an unhandled condition or a STOP or EXIT statement (or an exit() function), this contains the enclave return code for termination.

sub_reason_code **(output)**
The subroutine reason code. This is 0 for normal subroutine returns. If the

enclave is terminated due to an unhandled condition or a STOP or EXIT statement (or an exit() function), this contains the enclave reason code for termination.

sub_feedback_code **(output)**
The feedback code for enclave termination. This is the CEE000 feedback code for normal subroutine returns. If the enclave is terminated due to an unhandled condition or a STOP or EXIT statement (or an exit() function), this contains the enclave feedback code for termination.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The environment was activated and the routine called.

4 The function code is not valid.

8 CEEPIPI was called from a Language Environment-conforming HLL.

12 The indicated environment was initialized for main routines. No routine was executed.

16 The *token* is not valid.

28 The enclave was terminated but the process level persists.

This value indicates the enclave was terminated while the process was retained. This can occur due to a STOP or EXIT statement (or an exit() function) being issued or due to an unhandled condition. The *sub_ret_code*, *sub_reason_code*, and *sub_feedback_code* indicate this action.

40 The subprogram was an XPLINK program and the preinitialized environment is non-XPLINK. This is not valid.

41 Indicates the routine address could not be converted to a function descriptor.

Usage notes:

- The enclave terminates if the subroutine issues a STOP or EXIT statement (or an exit() function), or if there is an unhandled condition. However, the process level is not terminated. When the enclave level is terminated, any subsequent invocation creates a new enclave using the same runtime options used in the creation of the first enclave. Language Environment does not delete any user routines that were loaded into the PreInit table.

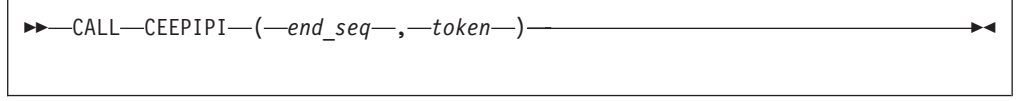
  However, if, the first valid entry in the PreInit table is different than when the enclave was last initialized, the assembler user exit (CEEBXITA), HLL user exit (CEEBINT), and/or programmer default runtime options (CEEUOPT) used during the enclave re-initialization might be different. PreInit subroutine initialization uses these external references only when associated with the first valid entry in the PreInit table. Therefore, when using PreInit subroutine environments and you want consistent enclave initialization behavior across the stop semantics, you need to ensure the first valid entry in the PreInit table does not change, especially when it contains the aforementioned external references.(See “Stop semantics” on page 485.)
- Any subroutine that modifies external data cannot make assumptions about the initial state of that external data. The initial state of the external data is influenced by previous instances of the same subroutine and also by previous instances of any subroutine that caused enclave termination.

- C subroutines that are not naturally reentrant and C++ subroutines can be invoked using call_sub_addr only in an XPLINK environment. In a non-XPLINK environment, they must be invoked using call_sub.
- If the first entry in the PreInit table contained a CEEBXITA, CEEBINIT or CEEUOPT when the environment was initialized and is then deleted or changed, the results of subsequent enclave re-initialization or termination is unpredictable. It is the responsibility of the user to ensure the first entry in the PreInit table does not change, especially when it contains the aforementioned external references.

(end_seq) — end a sequence of calls

This invocation of CEEPIPI declares that a sequence of uninterrupted calls to subroutines by this driver program has finished.

Syntax



end_seq **(input)**
A fullword function code (integer value = 8) containing the end_seq request

token **(input)**
A fullword with the value of the token returned by (init_sub_dp) when the common runtime environment is initialized.

The *token* must identify a previously preinitialized environment that was prepared for multiple calls by the (start_seq) call.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The environment is no longer prepared for a sequence of calls.

4 The function code is not valid.

8 The indicated environment was already active; no action taken.

16 The *token* is not valid.

20 The *token* was not used in a start_seq call.

Usage notes:

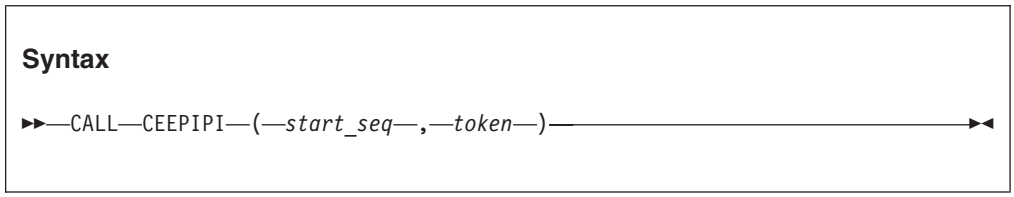
- (end_seq) can be used only in conjunction with a Language Environment environment initialized by an (init_sub_dp) function code. A return code of 4 is set for environments initialized by other than (init_sub_dp).
- Only (call_sub) or (call_sub_addr) invocations are allowed between the (start_seq) and (end_seq) calls.
- The driver program cannot cancel any STAE or ESPIE routines.
- This function can be called from an active environment if the Preinitialization environment indicated by *token* was created with the (init_sub_dp) function.

(start_seq) — start a sequence of calls

This invocation of CEEPIPI declares that a sequence of uninterrupted calls is made to a number of subroutines by this driven program to the same preinitialized

environment. This minimizes the overhead between calls by performing as much activity as possible at the start of a sequence of calls.

Syntax



start_seq **(input)**
A fullword function code (integer value = 7) containing the start_seq request.

token **(input)**
A fullword with the value of the token returned by (init_sub_dp) when the common runtime environment is initialized.

The *token* must identify a previously preinitialized environment for subroutines that are dormant at the time of the call.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The environment was prepared for a sequence of calls.

4 The function code is not valid.

8 The indicated environment was already active; no action taken.

16 The *token* is not valid.

20 Sequence already started using *token*.

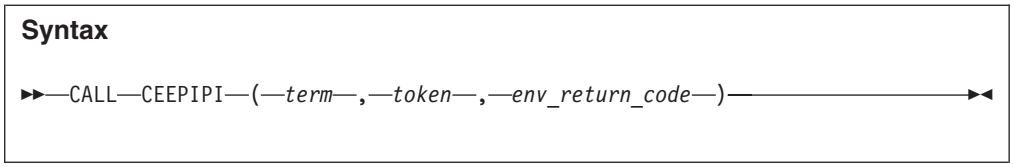
Usage notes:

- (start_seq) can be used only in conjunction with a Language Environment environment initialized by (init_sub_dp) function code. A return code 4 is set for environments not initialized by (init_sub_dp).
- (start_seq) minimizes the overhead between calls by allowing Language Environment to perform as much activity as possible at the start of the sequence of calls.
- Only (call_sub) or (call_sub_addr) invocations are allowed between the (start_seq) and (end_seq) calls.
- The same *token* must be passed for all invocations of (call_sub) or (call_sub_addr) between the (start_seq) and (end_seq) function codes. You can vary the routine invoked.
- During a CEEPIPI call sequence, the user's CEEPIPI driver must insure that the Language Environment recovery routines are never invoked when a program check or abend occurs in the user application code. One way to do this is to run with Trap (ON,NOSPIE), and also establish an ESTAE to handle errors when Language Environment is not active.

(term) — terminate environment

This invocation of CEEPIPI terminates the environment identified by the value given in *token*. This service is used for terminating environments created for subroutines or main routines.

Syntax



term **(input)**
A fullword function code (integer value = 5) containing the termination request.

token **(input)**
A fullword with the value of the token of the environment to be terminated. This token is returned by a (init_main), (init_main_dp), (init_sub), or (init_sub_dp) request during the initialization call.

The *token* must identify a previously preinitialized environment that is dormant at the time of the call.

env_return_code **(output)**
A fullword integer which is set to the return code from the environment termination.

If the environment was initialized for a main routine or a subroutine, and the last (call_sub) or (call_sub_addr) issued stop semantics, the value of *env_return_code* is zero.

If the environment was initialized for a subroutine and the last (call_sub) or (call_sub_addr) did not terminate with stop semantics, *env_return_code* contains the same value as that in *sub_ret_code* from the last (call_sub) or (call_sub_addr).

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The environment was activated and termination was requested.

4 Non-valid function code.

8 If *token* was initialized by (init_main) or (init_sub), CEEPIPI(term) was called from a Language Environment-conforming routine.

If *token* was initialized by (init_main_dp), CEEPIPI(term) was called from a Language Environment-conforming routine that is not running in a (main_dp) environment, or *token* is already in use for another call to CEEPIPI

16 The *token* is not valid.

Usage notes:

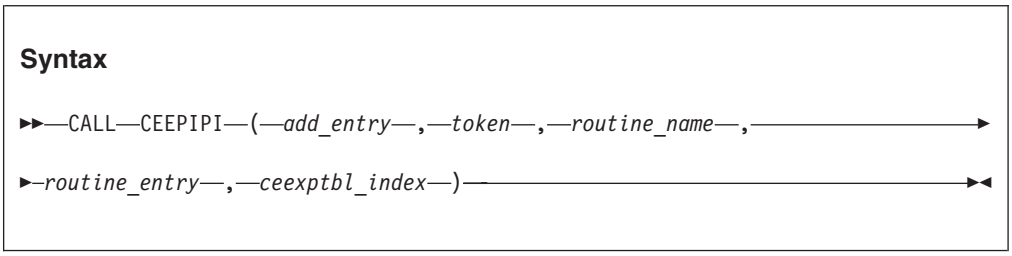
- All resources obtained are released when the environment terminates.
- All routines loaded by Language Environment are deleted when the environment terminates.
- Subsequent references to *token* by preinitialization services result in an error indicating the token is not valid.

(add_entry) — add an entry to the PreInit table

This invocation of CEEPIPI adds an entry for the environment represented by *token* in the Language Environment-maintained table. If a routine entry address is not provided, the routine name is used to dynamically load the routine and add it to

the PreInit table. The PreInit table index for the new entry is returned to the calling routine.

Syntax



add_entry **(input)**
A fullword function code (integer value = 6) containing the add_entry request.

token **(input)**
A fullword with the value of the token associated with the environment that adds this new routine. This token is returned by a (init_main), (init_main_dp), (init_sub), or (init_sub_dp) request.

The *token* must identify a previously preinitialized environment that is dormant at the time of the call.

routine_name **(input)**
A character string of length 8, left-justified and padded right with blanks, containing the name of the routine. To indicate the absence of the name, this field should be blank. If *routine_entry* is zero, this is used as the load name.

routine_entry **(input/output)**
The routine entry address that is added to the PreInit table. If *routine_entry* is zero on input, *routine_name* is used as the load name. On output, *routine_entry* is set to the load address of *routine_name*.

The high-order bit of the entry_point address must be set to indicate the addressing mode for the routine. If the high-order bit is OFF, the routine is called in 24 bit addressing mode and the address must be a valid 24 bit address. If the high-order bit is ON, the routine is called in 31 bit addressing mode and the address must be a valid 31 bit address.

ceexptbl_index **(output)**
The index to the PreInit table where this routine was added. If the return code is nonzero, this value is indeterminate. The index starts at zero.

Note: The environment that was preinitialized can be an XPLINK environment or a non-XPLINK environment. If the routine being added is an XPLINK routine, then the previously initialized environment must also be XPLINK.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The routine was added to the PreInit table.

4 Non-valid function code.

8 If *token* was initialized by (init_main) or (init_sub), CEEPIPI(add_entry) was called from a Language Environment-conforming routine.

If *token* was initialized by (init_main_dp), CEEPIPI(add_entry) was called from a Language Environment-conforming routine that is not running in a (main_dp) environment, or *token* is already in use for another call to CEEPIPI

12 The routine did not contain a valid Language Environment entry prolog. Ensure that the routine was compiled with a current Language Environment enabled compiler. The PreInit table was not updated.

16 The *token* is not valid.

20 The *routine_name* contains only blanks and the *routine_entry* was zero. The PreInit table was not updated.

24 The *routine_name* was not found or there was a load failure; the PreInit table was not updated.

28 The PreInit table is full. No routine was added to the table, nor was any routine loaded by Language Environment.

32 An unhandled error condition was encountered. This error is a result of a program interrupt or other abend that occurred that prevented the preinitialization services from completing.

38 Non-valid entry: A non-XPLINK subenvironment was preinitialized and the program that was being added is an XPLINK program.

42 Non-valid entry: The *routine_entry* had the high-order bit off indicating this routine is a 24 bit addressing mode routine but the environment is an XPLINK 31-bit environment. This is not valid.

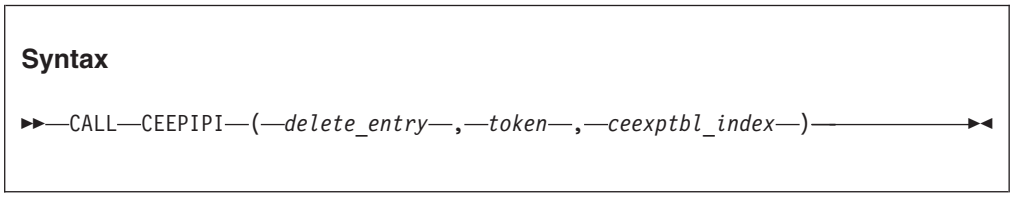
Usage notes:

- The PreInit table is built using the macros described in this topic. Therefore, its size is under the control of your application, not Language Environment.
- None of the routines in the PreInit table can be nested routines. All routines must be external routines.
- Language Environment uses the high-order bit of the entry address to determine what AMODE to use when calling the routine. If the *routine_entry* is zero, and the *routine_name* is supplied, Language Environment uses the AMODE returned by the system loader. If the *routine_entry* is supplied, you must provide the AMODE in the high-order bit of the address.
- An *add_entry* of an XPLINK program into a non-XPLINK preinitialized sub-environment will be not valid. If the environment is non-XPLINK, then the subprogram added with the *add_entry* function must also be non-XPLINK. However, you can do an *add_entry* of a main XPLINK program into a non-XPLINK environment. When a *call_main* is done with this scenario the environment will switch to XPLINK in order to allow the program to run.

(delete_entry) — delete an entry from the PreInit table

This function deletes an entry from the PreInit table. The entry is then available for subsequent (add_entry) functions.

Syntax



delete_entry **(input)**
fullword function code (integer value = 11) containing the delete_entry request

token **(input)**
a fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub), or (init_sub_dp) request.

ceexptbl_index **(input)**
the index into the PreInit table of the entry to delete.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The routine was deleted from the PreInit table

4 The function code is not valid.

8 If *token* was initialized by (init_main) or (init_sub), CEEPIPI(delete_entry) was called from an active environment.

If *token* was initialized by (init_main_dp), CEEPIPI(add_entry) was called from an active environment other than a (main_dp) environment, or *token* is already in use for another call to CEEPIPI.

No entries were deleted from the PreInit table.

16 The *token* is not valid

20 The PreInit table entry indicated by *ceexptbl_index* was empty.

24 The index passed is outside the range of the table.

28 The system request to delete the routine failed; the routine was not deleted from the PreInit table.

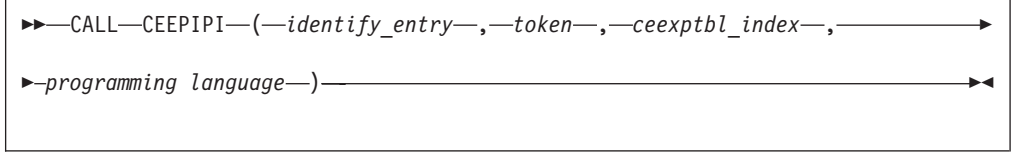
Usage notes:

- The *token* must identify a previously preinitialized environment that is dormant at the time of the call.
- If the routine indicated by *ceexptbl_index* had been loaded by CEEPIPI, it will be deleted.
- (delete_entry) no longer issues return code 12 (the environment indicated by *token* was not created with a (init_main) request; the routine was not deleted from the PreInit table).

(identify_entry) — identify an entry in the PreInit table

This invocation of CEEPIPI identifies the language of the entry point for a routine in the PreInit table.

Syntax



identify_entry **(input)**
A fullword containing the *identify_entry* function code (integer value=13).

token **(input)**
A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.

ceexptbl_index **(input)**
A fullword containing the index in the PreInit table of the entry to identify the programming language.

programming language **(output)**
A fullword with one of the following possible values:

3	C/C++
5	COBOL
10	PL/I
11	Enterprise PL/I for z/OS
15	Language Environment-enabled assembler
16	PL/X

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0	The programming language has been returned.
4	Non-valid function code.
8	CEEPIPI was called from an active environment.
16	The *token* is not valid.
20	The PreInit table entry indicated by *ceexptbl_index* was empty.
24	The index passed is outside the range of the table.

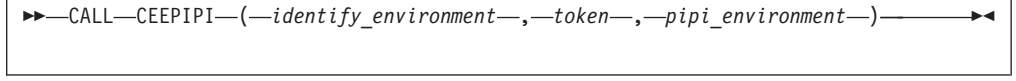
Usage notes:

- The *token* must identify a previously preinitialized environment that is dormant at the time of the call and was established with the (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.
- The *programming language* can be used by the driver to determine the format of the parameter list for the routine in cases where the language of the entry is not known.
- When a PreInit table entry contains multiple languages, *programming_language* is the language of the entry point for the entry.

(identify_environment) — identify the environment in the PreInit table

This invocation of CEEPIPI identifies the environment that was preinitialized.

Syntax



identify_environment **(input)**
A fullword containing the *identify_environment* function code (integer value=15).

token **(input)**
A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.

pipi_environment **(output)**
A fullword (32 Bit) mask value will be returned. For information about the mask value, see Table 69 on page 509.

Table 69. pipi_environment mask values

pipi_environment	Mask value	Action
ceepipi_main	X'80000000'	PreInit main environment is initialized.
ceepipi_enclave_initialized	X'40000000'	PreInit enclave is initialized.
ceepipi_dp_environment	X'20000000'	PreInit sub dp environment is initialized.
ceepipi_dp_seq_of_calls_active	X'10000000'	PreInit seq call function is active.
ceepipi_dp_exits_established	X'08000000'	PreInit sub dp exits is set.
ceepipi_sir_unregistered	X'04000000'	PreInit sir is registered.
ceepipi_sub_environment	X'02000000'	PreInit sub environment is initialized.
ceepipi_XPLINK_environment	X'01000000'	PreInit XPLINK environment is initialized.
ceepipi_init_main_dp_environment	X'00200000'	PreInit main dp environment is initialized.

Note: Mask bits other than those listed in the table may be nonzero. The meaning of these bits is not defined.

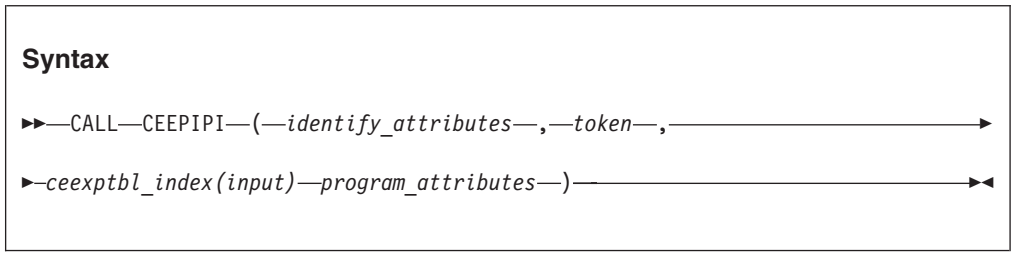
Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The Preinitialization environment mask has been returned.
4 Non-valid function code.
8 CEEPIPI was called from an active environment.
16 The *token* is not valid.

(identify_attributes) — identify the program attributes in the PreInit table

This invocation of CEEPIPI identifies the program attributes of a program in the PreInit table.

Syntax



identify_attributes **(input)**
A fullword containing the *identify_attributes* function code (integer value=16).

token **(input)**
A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.

ceexptbl_index **(input)**
A fullword containing the index in the PreInit table of the entry to identify the programming attributes.

program_attributes **(output)**
A fullword (32–bit) mask value will be returned indicating the following:

Table 70. program_attributes mask values

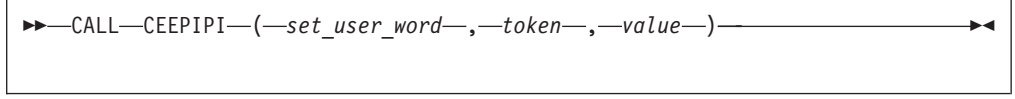
program_attribute	Mask value	Action
loaded_by_pipi	X'80000000'	The Preinitialization entry was loaded by Language Environment
XPLINK_program	X'40000000'	The Preinitialization entry loaded is an XPLINK program
Address_not_resolved	X'20000000'	The Preinitialization entry could not be loaded

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The Preinitialization environment mask has been returned.
4 Non-valid function code.
8 CEEPIPI was called from an active environment.
16 The *token* is not valid.
20 The PreInit table entry indicated by ceexptbl_index was empty.
24 The index passed is outside the range of the table.

(set_user_word) -- set value to be used to initialize CAA user word

Syntax



set_user_word **(input)**
A fullword containing the set_user_word function code (integer value = 17).

token **(input)**
A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.

value **(input)**
A fullword value that will be used to initialize the user word in the initial thread CAA when the application is invoked using the (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) functions for the passed-in environment token.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are::

0 The User Word has been set.
4 Non-valid function code.
16 The *token* is not valid.

Usage notes:

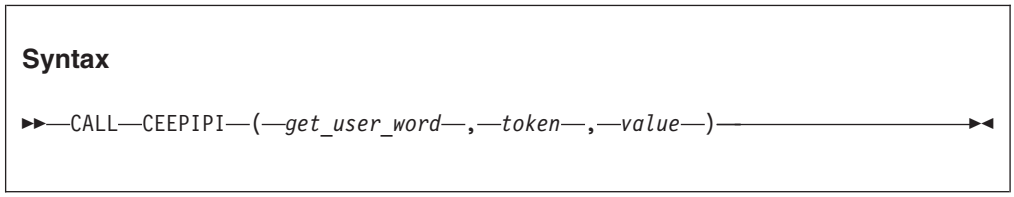
- This value is saved in an area associated with the passed-in environment token. It is copied into the CAA for the initial thread when the next (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) function is done to start an application. The application can then examine or update this user word in the CAA (CEECAA_USER_WORD). When the application ends, the final value in CEECAA_USER_WORD is not copied back into the area associated with the environment token. When the next application

is started using a function such as (call_main), (call_sub), or(call_sub_addr), the user word value last established by (set_user_word) is used again.

- The user word associated with the environment token is initialized to 0 when (init_main), (init_main_dp), (init_sub), or (init_sub_dp) is done. The CAA for the initial process thread is initialized with 0 if no (set_user_word) function call has been done before the application is started.
- The user word in all CAAs other than the initial thread CAA is set to 0. The user word in all CAAs in nested enclaves is set to 0.
- When fork() is done, the user word in the CAA for the new process inherits the value that is in the CAA at the time fork() is done.
- The use of the CAA user word is not supported in the assembler user exit routine (CEEBXITA and related modules), or in the CEEPIPI service routines specified in the service routine vector (@LOAD, @DELETE, @GETSTORE, @FREESTORE, @EXCEPRTN, @MSGRTN).
- Any user code that runs on a CEEPIPI environment before the first (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) request will see zero in the CAA_USER_WORD. Examples of this code include static constructors run for programs that get loaded when a CEEPIPI environment is initialized. Any changes to the CAA_USER_WORD made by this code are overlaid when the next (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) is done for that environment.

(get_user_word) -- get value to be used to initialize CAA user word

Syntax



get_user_word **(input)**
A fullword containing the get_user_word function code (integer value = 18).

token **(input)**
A fullword with the value of the token of the environment. This is the token returned by a (init_main), (init_main_dp), (init_sub) or (init_sub_dp) request.

value **(output)**
A fullword that will be returned containing the current value that will be used to initialize the CAA user word when the next application is invoked using the (call_main), (call_sub), (call_sub_addr), (call_sub_addr_nochk), or (call_sub_addr_nochk2) functions.

Return codes: Register 15 contains a return code indicating the success or failure of the request. Possible return codes (in decimal) are:

0 The current value of the User Word has been returned.
4 Non-valid function code.
16 The *token* is not valid.

Usage notes:

- The value returned will be the one previously set by the last (set_user_word) request for this token. If no (set_user_word) has yet been done for this token, 0 will be returned.

Service routines

Under Language Environment, you can specify several service routines to execute a main routine or subroutine in the preinitialized environment. To use the routines, specify a list of addresses of the routines in a service routine vector as shown in Figure 108.

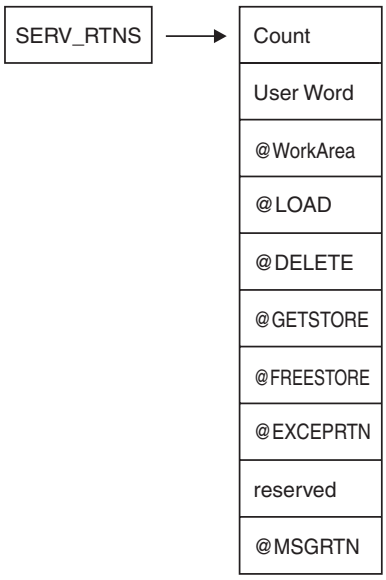


Figure 108. Format of service routine vector

The service routine vector is composed of a list of fullword addresses of routines that are used instead of Language Environment service routines. The list of addresses is preceded by the number of the addresses in the list, as specified in the *count* field of the vector. The *service_rtns* parameter that you specify in calls to (init_main) and (init_sub) contains the address of the vector itself. If this pointer is specified as zero (0), Language Environment routines are used instead of the service routines shown in Figure 108.

The @GETSTORE and @FREESTORE service routines must be specified together; if one is zero, the other is automatically ignored. The same is true for the @LOAD and @DELETE service routines. If you specify the @GETSTORE and @FREESTORE service routines, you do not have to specify the @LOAD and @DELETE service routines and vice-versa.

When replacing only the storage management routines without the program management routines, the user must be aware that they may not be accounting for all the storage obtained on behalf of the application. Contents management obtains storage for the load module being loaded. This storage will not be managed by the user storage management routines.

The service routines may be AMODE(31) / RMODE(ANY) if the application has no AMODE(24) programs. Otherwise the service routines must be AMODE(ANY) / RMODE(24).

Count
A fullword binary number representing the number of fullwords that follow.

The *count* does not include itself. In Figure 108 on page 512, the count is 9. For each vector slot, a zero represents the absence of the routine, a nonzero represents the presence of a routine.

User Word
A fullword that is passed to the service routines. The *user word* is provided as a means for your routine to communicate to the service routines.

@WorkArea
An address of a work area of at least 256 bytes that is doubleword aligned. The first word of the area contains the length of the area provided. This parameter is required if service routines are present in the service routine vector. This length field must be initialized each time you bring up a new PreInit environment.

@LOAD
This routine loads named routines for application management. The parameter that is passed contains the following:

Name_addr
The fullword address of the name of the module to load (input parameter).

Name_length
A fixed binary(31) length of the module name (input parameter).

User_word
A fullword user field (input parameter).

Load_point
Either zero (0), or the address where the @LOAD routine is to store the load point address of the loaded routine (input and output parameter).

Entry_point
The fullword entry point address of the loaded routine (output parameter).

Module_size
The fixed binary(31) size of the module that was loaded (output parameter).

Return code
The fullword return code from load (output).

Reason code
The fullword reason code from load (output). The return and reason codes are listed in Table 71.

Table 71. Return and reason codes

Return code	Reason code	Description
0	0	Successful
0	12	Successful — loaded using SVC8
4	4	Unsuccessful — module loaded above the line when in AMODE(24)
8	4	Unsuccessful — load failed
16	4	Unsuccessful — uncorrectable error occurred

@DELETE
This routine deletes routines for application management. The parameter that is passed contains the following:

Name_addr
The fullword address of the module name to be deleted (input parameter).

Name_length
A fixed binary(31) length of module name (input parameter).

User_word
A fullword user field (input parameter).

Rsvd_word
A fullword reserved for future use (input parameter); must be zero.

Return code
The return code from delete service (output).

Reason code
The reason code from delete service (output). The return and reason codes are listed in Table 72.

Table 72. Return and reason codes

Return code	Reason code	Description
0	0	Successful
8	4	Unsuccessful — delete failed
16	4	Unsuccessful — uncorrectable error occurred

@GETSTORE
This routine allocates storage on behalf of the storage manager. This routine can rely on the caller to provide a save area, which can be the @Workarea. The parameter list that is passed contains the following:

Amount
A fixed binary(31) amount of storage requested (input parameter).

Subpool_no
A fixed binary(31) subpool number 0-127 (input parameter). Language Environment allocates storage from the process-level storage pools.

User word
A fullword user field (input parameter).

Flags A fullword flag area (input parameter), as shown in the following table. The remaining flag bits are reserved for future use and must be zero.

Bit	Setting	Description
Zero	ON	The storage required must be allocated below the 16MB line.
	OFF	The storage required can be allocated anywhere.
One	ON	The storage required was requested to be backed by 1MB pages. This setting might be ignored.
	OFF	The storage required was requested to be backed by the default 4KB pages.

Stg_address
The fullword address of the storage obtained or zero (output parameter).

Obtained
A fixed binary(31) number of bytes obtained (output parameter).

Return code
The return code from @GETSTORE service (output parameter).

Reason code
The reason code from the @GETSTORE service (output parameter).

The return and reason codes are listed in Table 73.

Table 73. Return and reason codes

Return code	Reason code	Description
0	0	Successful
16	0	Unsuccessful — uncorrectable error occurred

@FREESTORE

This routine frees storage on behalf of the storage manager. The parameter list passed contains the following:

Amount
The fixed binary(31) amount of storage to free (input parameter).

Subpool_no
The fixed binary(31) subpool number 0-127 (input parameter). Language Environment allocates storage from the process-level storage pools.

User word
A fullword user field (input parameter).

Stg_address
The fullword address of the storage to free (input parameter).

Return code
The return code from the @FREESTORE service (output).

Reason code
The reason code from the @FREESTORE service (output).

The return and reason codes are listed in Table 74.

Table 74. Return and reason codes

Return code	Reason code	Description
0	0	Successful
16	0	Unsuccessful — uncorrectable error occurred

@EXCEPRTN

This routine traps program interruptions and abends for condition management. The parameter list passed contains the following:

Handler_addr
During an initialization call, this parameter contains the address of the Language Environment condition handler. During a termination call, this parameter contains a pointer to a fullword field containing zeroes.

Environment_token
A fullword Recovery Environment token (input). This token is different from the Preinitialization environment token used with CEEPIPI calls.

User_word
A fullword user field (input parameter)

Abend_flags
A fullword flag area containing abend flags (input)

Check_flags
A fullword flag area containing program check flags (input)

Return code
The return code from the @EXCEPRTN service (output).

Reason code
The reason code from the @EXCEPRTN service (output).

The exception router is responsible for trapping and routing exceptions. These are the services typically obtained via the ESTAE and ESPIE macros.

During initialization, Language Environment puts the address of the Language Environment condition handler in the first field of the above parameter list, and sets the environment token field to a value that must be passed on to the Language Environment condition handler. It also sets abend and check flags as appropriate, and then calls your exception router to establish an exception handler.

The meaning of the bits in the abend flags are given by the following declare:

```
dcl
 1 abendflags,
   2 system,
     3 abends bit(1),  /* control for system abends desired  */
     3 rsrv1  bit(15), /* reserved                           */
   2 user,
     3 abends bit(1),  /* control for user abends desired    */
     3 rsrv2  bit(15); /* reserved                           */
```

The meaning of the bits in the check flags is given by the following declare:

```
      1 checkflags,
        2 type,
          3 reserved3             bit(1),
          3 operation             bit(1),
          3 privileged_operation  bit(1),
          3 execute               bit(1),
          3 protection            bit(1),
          3 addressing            bit(1),
          3 specification         bit(1),
          3 data                  bit(1),
          3 fixed_overflow        bit(1),
          3 fixed_divide          bit(1),
          3 decimal_overflow      bit(1),
          3 decimal_divide        bit(1),
          3 exponent_overflow     bit(1),
          3 exponent_underflow    bit(1),
          3 significance          bit(1),
          3 float_divide          bit(1),
        2 reserved4             bit(16);
```

The return and reason codes that the exception router must use are listed in Table 75.

Table 75. Return and reason codes

Return code	Reason code	Description
0	0	Successful

Table 75. Return and reason codes (continued)

Return code	Reason code	Description
4	4	Unsuccessful — the exit could not be established or removed
16	4	Unsuccessful — unrecoverable error occurred

When an exception occurs, the exception handler must determine if the Language Environment condition handler is interested in the exception (by examining abend and check flags). If the condition handler is not interested in the exception, the exception handler must treat the program as in error, but can assume the environment for the thread to be functional and reusable. If the condition handler is interested in the exception, the exception handler must invoke the condition handler, passing the parameters listed in Table 76.

Table 76. Parameters for language environment condition handler

Parameter	Attributes	Type
Environment Token	Fixed Bin(31)	Input
Address of SDWA	Pointer	Input
Return Code	Fixed Bin(31)	Output
Reason Code	Fixed Bin(31)	Output

The return and reason codes upon return from the Language Environment condition handler are listed in Table 77.

Table 77. Return and reason codes

Return code	Reason code	Description
0	0	Continue with the exception. Percolate the exception taking whatever action would have been taken had it not been handled at all. In this case, your exception handler can assume the environment for the thread to be functional and reusable.
0	4	Continue with the exception. Percolate the exception taking whatever action would have been taken had it not been handled at all. In this case, the environment for the thread is probably unreliable and not reusable. A forced termination is suggested.
4	0	Resume execution using the updated SDWA. The invoked Language Environment condition handler will have already used the SETRP RTM macro to set the SDWA for correct resumption.

During termination, the exception router is invoked with the condition handler address (first parameter) set to zero to de-establish the exit (if it was established during initialization).

When a nested enclave is created, Language Environment calls the exception router to establish another exception handler exit, and then makes a call to de-establish it when the nested enclave terminates. If an exception occurs while the second exit is active, special processing is performed. Depending on what

this second exception is, either the first exception will not be retried, or processing will continue on the first exception by requesting retry for the second exception.

If the Language Environment condition handler determines that execution should resume for an exception, it will set the SDWA with SETRP and return with return/reason codes 4/0. Execution will resume in library code or in user code, depending on what the exception was.

The exception handler must be capable of restoring all the registers from the SDWA when control is given to the retry routine. The ESPIE and ESTAE services are capable of accomplishing this.

In using the exception router service:

- The exception handler should not invoke the Language Environment condition handler if active I/O has been halted and is not restorable.
- This service requires an XA or ESA environment.

If an exception occurs while the exception handler is in control before another exception handler exit has been stacked, the exception handler should assume that the exception could not be handled and that the environment for the program (thread) is damaged. In this case, the exception handler should force termination of the preinitialized environment.

When @EXCEPRTN is specified, the following items are not supported:

- XPLINK applications
- POSIX(ON) applications
- DYNDUMP settings other than DYNDUMP(,NODYNAMIC)
- IMS applications
- Applications that use Binary Floating Point (BFP) or Decimal Floating Point (DFP) numbers
- Applications that use the Compare-and-Trap family of instructions

Note:

1. If the passed-in SDWA from the exception handler to the Language Environment condition handler does not contain valid high registers, the "HR_VALID" flag bit in the Machine State "FLAGS" field will be off, indicating that the saved high registers are not valid.
2. If a nested enclave ends because of an unhandled condition and a 4094-40 ABEND is declared, the high registers may not be valid in the Machine State that contains information about the 4094-40 ABEND.
3. If registers in the passed-in SDWA at the time of interrupt (in the SDWAGRSV field) are not appropriate or recognizable, and Language Environment instead saves the registers from the SDWASRSV field in the Machine State, the high registers may not be valid in the Machine State.

@MSGRTN

This routine allows error messages to be processed by the caller of the application.

If the message pointer is zero, your message routine is expected to return the size of the line to which messages are written (in the line_length field). This allows messages to be formatted correctly — that is, broken at places such as blanks.

Message
A pointer to the first byte of text that is printed, or zero (input parameter).

Msg_len
The fixed binary(31) length of the message (input parameter).

User word
A fullword user field (input parameter).

Line_length
The fixed binary(31) size of the output line length. This is used when Message is zero (output parameter).

Return and reason codes
Two fullwords containing the return and reason codes listed in Table 78 (output parameters).

Table 78. Return and reason codes

Return code	Reason code	Description
0	0	Successful
16	4	Unsuccessful — uncorrectable error occurred

An example program invocation of CEEPIPI

This section includes a sample of a PreInit assembler driver program. This assembler program called ASMPIPI invokes CEEPIPI to:
- Initialize a subroutine environment under Language Environment
- Load and call a reentrant HLL subroutine
- Terminate the Language Environment environment

Following the assembler program are examples of the program HLLPIPI written in C, COBOL, and PL/I. HLLPIPI is called by an assembler program, ASMPIPI. ASMPIPI uses the Language Environment preinitialized program subroutine call interface. You can use the assembler program to call the HLL versions of HLLPIPI.

```
*COMPILATION UNIT:  LEASMPIP
*********************************************************************
*                                                                   *
*   Function :  CEEPIPI - Initialize the Preinitialization          *
*                         environment, call a Preinitialization     *
*                         HLL program, and terminate the environment. *
*                                                                   *
* 1.Call CEEPIPI to initialize a subroutine environment under LE.   *
* 2.Call CEEPIPI to load and call a reentrant HLL subroutine.       *
* 3.Call CEEPIPI to terminate the LE Preinitialization environment. *
*                                                                   *
* Note:  ASMPIPI is not reentrant.                                  *
*                                                                   *
*********************************************************************
*
* ==================================================================
* Standard program entry conventions.
* ==================================================================
ASMPIPI  CSECT
         STM   R14,R12,12(R13)    Save caller's registers
         LR    R12,R15            Get base address
         USING ASMPIPI,R12        Identify base register
         ST    R13,SAVE+4         Back-chain the save area
         LA    R15,SAVE           Get addr of this routine's save area
         ST    R15,8(R13)         Forward-chain in caller's save area
```

```
         LR    R13,R15            R13 -> save area of this routine
*
* Load LE CEEPIPI service routine into main storage.
*
         LOAD  EP=CEEPIPI         Load CEEPIPI routine dynamically
         ST    R0,PPRTNPTR        Save the addr of CEEPIPI routine
*
* Initialize an LE Preinitialization subroutine environment.
*
INIT_ENV EQU   *
         LA    R5,PPTBL           Get address of Preinitialization Table
         ST    R5,@CEXPTBL        Ceexptbl-addr -> Preinitialization Table
         L     R15,PPRTNPTR       Get address of CEEPIPI routine
*                                 Invoke CEEPIPI routine
         CALL  (15),(INITSUB,@CEXPTBL,@SRVRTNS,RUNTMOPT,TOKEN)
*                                 Check return code:
         LTR   R2,R15               Is R15 = zero?
         BZ    CSUB                 Yes (success).. go to next section
*                                   No (failure).. issue message
         WTO   'ASMPIPI : call to (INIT_SUB) failed',ROUTCDE=11
         C     R2,=F'8'           Check for partial initialization
         BE    TSUB                 Yes.. go do Preinitialization termination
*                                   No.. issue message & quit
         WTO   'ASMPIPI : INIT_SUB failure RC is not 8.',ROUTCDE=11
         ABEND (R2),DUMP          Abend with bad RC and dump memory
*
* Call the subroutine, which is loaded by LE
*
CSUB     EQU   *
         L     R15,PPRTNPTR       Get address of CEEPIPI routine
         CALL  (15),(CALLSUB,PTBINDEX,TOKEN,PARMPTR,                  X
               SUBRETC,SUBRSNC,SUBFBC)   Invoke CEEPIPI routine
*                                 Check return code:
         LTR   R2,R15               Is R15 = zero?
         BZ    TSUB                 Yes (success).. go to next section
*                                   No (failure).. issue message & quit
         WTO   'ASMPIPI : call to (CALL_SUB) failed',ROUTCDE=11
         ABEND (R2),DUMP          Abend with bad RC and dump memory
*
* Terminate the environment
*
TSUB     EQU   *
         L     R15,PPRTNPTR       Get address of CEEPIPI routine
         CALL  (15),(TERM,TOKEN,ENV_RC)  Invoke CEEPIPI routine
*                                 Check return code:
         LTR   R2,R15               Is R15 = zero ?
         BZ    DONE                 Yes (success).. go to next section
*                                   No (failure).. issue message & quit
         WTO   'ASMPIPI : call to (TERM) failed',ROUTCDE=11
         ABEND (R2),DUMP          Abend with bad RC and dump memory
*
* Standard exit code.
*
DONE     EQU   *
         LA    R15,0                Passed return code for system
         L     R13,SAVE+4           Get address of caller's save area
         L     R14,12(R13)          Reload caller's register 14
         LM    R0,R12,20(R13)       Reload caller's registers 0-12
         BR    R14                  Branch back to caller
*
* ====================================================================
* CONSTANTS and SAVE AREA.
* ====================================================================
SAVE     DC    18F'0'
PPRTNPTR DS    A                  Save the address of CEEPIPI routine
*
* Parameters passed to a (INIT_SUB) call.
```

```
*
INITSUB  DC    F'3'               Function code to initialize for subr
@CEXPTBL DC    A(PPTBL)           Address of Preinitialization Table
@SRVRTNS DC    A(0)               Addr of service-rtns vector, 0 = none
RUNTMOPT DC    CL255' '           Fixed length string of runtime optns
TOKEN    DS    F                  Unique value returned (output)
*
* Parameters passed to a (CALL_SUB) call.
*
CALLSUB  DC    F'4'               Function code to call subroutine
PTBINDEX DC    F'0'               The row number of Preinitialization Table entry
PARMPTR  DC    A(0)               Pointer to @PARMLIST or zero if none
SUBRETC  DS    F                  Subroutine return code (output)
SUBRSNC  DS    F                  Subroutine reason code (output)
SUBFBC   DS    3F                 Subroutine feedback token (output)
*
* Parameters passed to a (TERM) call.
*
TERM     DC    F'5'               Function code to terminate
ENV_RC   DS    F                  Environment return code (output)
*
* ====================================================================
* Preinitialization Table.
* ====================================================================
PPTBL    CEEXPIT  ,               Preinitialization Table with index
         CEEXPITY HLLPIPI,0       0 = dynamically loaded routine
*
         CEEXPITS ,               End of PreInit table
*
*
         LTORG
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   ASMPIPI
```

HLLPIPI examples

Following is an example of a C subroutine called by ASMPIPI:

```
/*Module/File Name:  EDCPIPI  */
 /*********************************************************/
 /*                                                       */
 /* HLLPIPI is called by an assembler program, ASMPIPI.   */
 /* ASMPIPI uses the LE preinitialized program            */
 /* subroutine call interface. HLLPIPI can be written     */
 /* in COBOL, C, or PL/I.                                 */
 /*                                                       */
 /*********************************************************/
#include <stdio.h>
#include <string.h>
#include <time.h>
#pragma linkage(HLLPIPI, fetchable)
HLLPIPI ()
{
```

```
 printf ( "C subroutine beginning\n" );
 printf ( "Called using LE PreInit call\n" );
 printf ( "Subroutine interface.\n" );
 printf ( "C subroutine returns to caller\n" );
}
```

Following is an example of a COBOL program called by ASMPIPI:

```
CBL LIB,QUOTE
      *Module/File Name: IGZTPIPI
      ************************************************************
      *                                                          *
      * HLLPIPI is called by an assembler program, ASMPIPI.      *
      * ASMPIPI uses the LE preinitialized program               *
      * subroutine call interface. HLLPIPI can be written        *
      * in COBOL, C, or PL/I.                                    *
      *                                                          *
      ************************************************************
       IDENTIFICATION DIVISION.
       PROGRAM-ID. HLLPIPI.

       DATA DIVISION.
       WORKING-STORAGE SECTION.
       PROCEDURE DIVISION.
           DISPLAY "COBOL subprogram beginning".
           DISPLAY "Called using LE Preinitialization ".
           DISPLAY "Call subroutine interface.".
           DISPLAY "COBOL subprogram returns to caller.".

           GOBACK.
```

Following is an example of a routine called by ASMPIPI:

```
 /*Module/File Name:  IBMPIPI                                */
 /************************************************************/
 /*                                                          */
 /* HLLPIPI is called by an assembler program, ASMPIPI.      */
 /* ASMPIPI uses the LE preinitialized program               */
 /* subroutine call interface. HLLPIPI can be written        */
 /* in COBOL, C, or PL/I.                                    */
 /*                                                          */
 /************************************************************/
  HLLPIPI: PROC OPTIONS(FETCHABLE);
           DCL RESULT FIXED BIN(31,0) INIT(0);
           PUT SKIP LIST
               ('HLLPIPI  : PLI subroutine beginning.');
           PUT SKIP LIST
               ('HLLPIPI  : Called LE PIPI Call ');
           PUT SKIP LIST
               ('HLLPIPI  : Subroutine interface.   ');
           PUT SKIP LIST
               ('HLLPIPI  : PLI program returns to caller.');
           RETURN;
  END HLLPIPI;
```