REXX on TSO/E

by G. E. Hoernes

REXX is a programming language primarily designed for ease of use. First implemented on the Conversational Monitor System (CMS), REXX has been implemented on TSO Extensions (TSO/E) as a new command language, yet it contains all of the elements of a full-function language. After a brief definition of the main elements of the REXX language, the paper discusses why REXX was implemented on TSO/E, some alternative designs which were considered, and how the final design integrates the new language into the existing TSO/E structure, yet allows REXX programs to be interpreted in any Multiple Virtual Storage (MVS) address space, even outside the TSO/E environment. The paper also introduces the TSO/E "data stack," which is similar to the stack implemented in CMS, and describes how the definition of the CMS stack had to be extended to allow REXX programs executing concurrently on different MVS tasks to either share or not share the data stack. Throughout the paper, compatibility with other Systems Application Architecture environments, particularly CMS, and performance considerations are discussed.

The Restructured Extended Executor Language (REXX), designated as the Systems Application Architecture/Procedure Language, is a programming language designed for ease of use. The author of the language, Mike Cowlishaw, states that the one goal of REXX was to try "to make programming easier than it was before, in the belief that the best way to encourage high quality programs is to make writing them as simple and as enjoyable as possible."[1] This ease of use is achieved by using common English words in the syntax, using syntax which appears "natural" to a beginning programmer, and using relatively few, but well-chosen, commonly used, general-purpose functions.

For example, REXX variables are not declared, they are considered to be varying-length character strings allowed to hold any binary value of any length between zero and an implementation maximum.[2] Because there is no restriction on the values of the characters in the string, some characters in a string may be printable and some may not. If the characters in a string form a valid number, optionally with leading or trailing blanks, that string may participate in arithmetic operations.

The REXX language defines a very rich set of string operations; among them are parsing of a string by words, by character patterns, or by position. Other operations support counting of words, reversing of the string, and indexing by a pattern of characters or by a word. Representing all REXX variables as character strings allows two simplifications. First, the entire set of operations can be used to operate on all variables, thereby avoiding special rules that limit some operations to one type of variable (say, numeric), others to another type (say, character or bit). Second, only one set of operations is required, not one per type of data stored in the variable. Thus, fewer operations need to be defined because no conversion operations between representations have to be defined, and there is no need to define similar operations on different data types. Avoiding data

conversion also avoids the problem of having inaccuracies introduced by that process and allows REXX to operate on numbers with virtually unlimited accuracy.

Another feature of the language is its extensibility. The language supports extensive string operations; many are implemented using *built-in functions*. That set of functions supplied with the language can be extended easily by the user, an installation, or a product using *external functions*. The same syntax is used to call built-in functions—the functions supplied with the language—and external functions—the functions not part of the language added by a user, installation, or product. Although the built-in functions may be augmented by external functions which support specialized requirements, the user may, but need not, be aware of where the base language leaves off and the extensions begin.

REXX has few but very general and powerful primitives.

This extensibility of the language is even carried over to improving performance by using *packages*. It will be shown later how performance of external functions in packages approaches the performance of the built-in functions. Packages are not part of the base language but are supported in the TSO Extensions (TSO/E)[3] Release 2.1 implementation.

Two additional, but unrelated, features of the language are the instructions that support debugging and the ability to target commands to different command environments. Targeting a command to a command environment allows an application executing in TSO/E to issue a TSO/E command and target it to the TSO/E command processor. That command would have no meaning to any other command processor. If the Interactive System Productivity Facility (ISPF) were active, for example, an ISPF command could be targeted to the ISPF command processor within the same REXX program, known as a REXX *exec*.[4] REXX supports an instruction that defines the target for a command. The command processor environments can be added or deleted dynamically, again allowing for extensibility. The other additional language feature in the REXX language supports debugging. Debugging of a program written in REXX is made easier than it is with most languages because REXX contains an extensive set of tracing and debugging features.

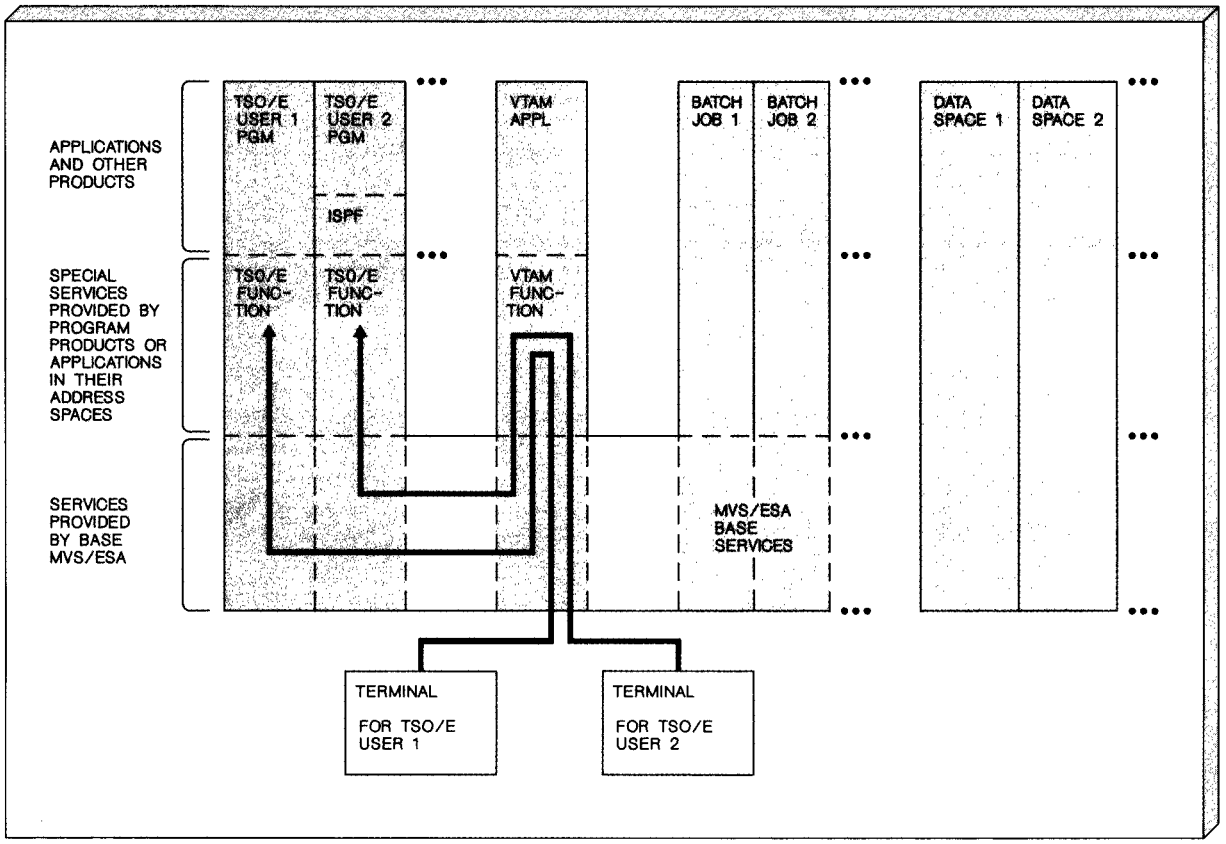
In short, REXX is a language with few but very general and powerful primitives. It is a language that can be extended and customized for special environments by adding "functions" or commands and is a language supporting many debugging options.

The syntax, history, and use of the language are documented extensively.[1,5,6] In addition to defining the syntax, the author of the language implemented an interpreter for REXX and distributed it for experimental use to IBM internal users. The first generally available product implementing REXX was the interpreter on the Virtual Machine/System Product (VM/SP). This interpreter, named the VM/SP System Product Interpreter,[7] was announced February 1, 1983, and is based on the IBM internal interpreter for REXX. Because of the ease of use of the language, it enjoyed rapid acceptance in the VM/SP community as a command language for Conversational Monitor System (CMS), as a general-purpose programming language, and as a language for writing ISPF dialogs. It was also "ported" to TSO/E as a prototype for experimentation within IBM. The REXX that is the product described in this paper is a new design that takes advantage of the CMS implementation and the experience with that prototype.

TSO/E. Before presenting REXX on TSO/E, we briefly discuss the MVS/ESA™ (Multiple Virtual Storage/Enterprise Systems Architecture) operating system.[8] This operating system supports the simultaneous or nearly simultaneous execution of multiple independent applications. The technique used to achieve separation between these independent activities is called an *address space*. A program executing in an address space is capable of addressing private virtual storage in its own address space and virtual storage common to it and to most other address spaces, but the program in one address space cannot address the private virtual storage of other address spaces.

Some program products, such as TSO/E and the Virtual Terminal Access Method (VTAM), operate in their own address spaces and provide specialized

Figure 1 Address spaces in MVS/ESA



services required by their own applications. When an application (or user program) operates with an address space, the services it has available depend on the type of host address space.

Figure 1 represents multiple address spaces, some sharing address ranges with others, some not. In this figure, TSO/E and VTAM provide a platform of functions above the MVS/ESA services. Programs executing in such an address space may call on services in the base or on services provided specifically within that one address space. The figure also shows each batch job in its own address space. Programs in these batch address spaces do not have access to special services beyond those of the base MVS/ESA system. Two data spaces are shown on the right side of the figure; they are included to show that these address spaces have no addressability in common with any other address space.

In the TSO/E address space, services tailored to interactive processing are provided. The TSO/E address space is created by a user logging onto TSO/E, which starts an initialization process for TSO/E services. Once initialization is completed, the TSO/E address space is controlled by the Terminal Monitor Program (TMP), remaining active and in control until LOGOFF. The TMP remains in a two-step loop of first reading a command from the input, normally the user terminal, and then executing that command. After the command completes execution, the TMP issues a READY message indicating completion and starts the loop again by waiting for new input from the user via the terminal. This TMP is part of an environment within the address space which allows the execution of an extensive set of commands, a set that can be augmented by users, installations, or other program products. Examples of TSO/E commands are listing allocated files (LISTA) and listing

information about data sets (LISTDS). An example of a program product that is invoked as a TSO/E command is ISPF. This command starts ISPF, which builds its own TSO/E subenvironment and enables an additional set of commands and services for execs and programs executing in that subenvironment.

Commands can be combined into lists of commands and saved as members of partitioned data sets (PDS). Once created, such a member can be executed as a unit. Historically such command lists (CLISTs) were limited to only one language, the TSO/E CLIST language. Starting with the latest release of TSO/E, Release 2.1, such lists of commands can also be written using REXX. REXX is a full-function language; thus the commands can be embedded in programs containing a great deal of logic in addition to the TSO/E commands.

REXX lends itself well as a replacement for CLIST in many applications, because of its characteristics discussed above. A REXX exec can make decisions, can easily read from and write to the terminal, and can issue commands. One reason why REXX is so well-suited for this purpose is that commands and terminal input and output are character strings. As was pointed out earlier, strings can be stored directly into REXX variables, and once stored, the REXX language has a large number of string-oriented functions to parse input and create new output.

REXX in TSO/E or non-TSO/E address spaces. In addition to being well-suited as a command language in TSO/E, REXX is also a good general-purpose language to implement applications. In the TSO/E address space, the interpreter of the REXX exec and commands called from within the exec can take advantage of TSO/E services as indicated by the TSO/E User 1 Program in Figure 1. The figure also shows how TSO/E is supported by MVS, so all MVS services are available in TSO/E as well. If ISPF is active (as it is for the TSO/E User 2 Program in Figure 1) because the REXX exec is called from within ISPF, the execs can also use the ISPF services. If the editor is active within ISPF, the exec can use the editor services in addition to the ISPF services and the services of TSO/E. This structure is hierarchical wherein each environment adds to the available services of the previous one.

Interpretation of REXX execs is not limited to the TSO/E address space. An exec may be used in any MVS address space supporting the base MVS services (Figure 1) because inherently the REXX interpreter does not require TSO/E services. However, if the exec executes outside of the TSO/E address space, the TSO/E services are not available to the exec and may not be used. For example, Batch Job 1 in Figure 1 could be the REXX interpreter interpreting a REXX exec outside of the TSO/E address space.

Interpretation of REXX execs is not limited to the TSO/E address space.

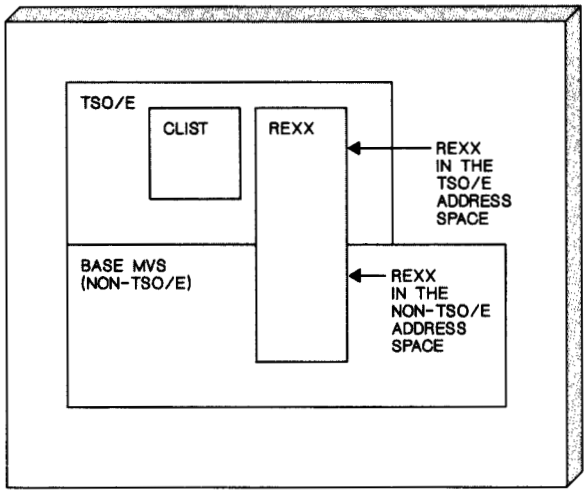
Because of the availability of certain services in some address spaces but not in others, an exec must be written to use only those services available in the address space it is intended to execute in. For example, if an exec is to execute in both MVS batch and the TSO/E address space, it could use only the MVS services or, after determining which address space and whether it is in TSO/E, optionally use TSO/E services.

In the remainder of the paper, references will be made to the operation of REXX in the TSO/E address space and in the non-TSO/E address space. The reason for that distinction is that if REXX executes in the TSO/E address space, it is fully integrated with other parts of TSO/E operating in that address space. Outside of TSO/E this interaction is not possible because services depending on the TSO/E environment are not available. The interaction with other parts of TSO/E in the TSO/E address space greatly enhances the functions supported by REXX. The function in TSO/E is a proper superset of the function in non-TSO/E; details will be shown later.

Design of the REXX component of TSO/E

The most significant requirements for REXX in TSO/E were to implement the Systems Application Architecture/Procedure Language (SAA/PL), the customer needs for an alternate command language in addition to CLIST, and the compatibility between TSO/E and CMS. These points are now discussed in detail.

Figure 2 REXX interprets in both TSO/E and non-TSO/E



SAA/PL. One of the primary functions of SAA is to provide a platform on which applications can be built.[9] One part of that platform is SAA/PL, announced by IBM in March 1987.[10] Applications written in SAA/PL can be "ported" to all other SAA environments, providing the potential for significant savings. SAA/PL is a portable language, but SAA does not define commands. Nevertheless, because commands are system-unique, and many execs contain them, many execs are not portable. However, the language provides information identifying the system at execution time and thus allows the inclusion of code which can issue the commands differently for each system. In many cases this code, which is sensitive to one system, is a small percentage of the total code, making the writing of portable execs only slightly more difficult than the writing of an exec targeted for only one system.

The distinction between SAA/PL and REXX derives from the fact that not all parts of REXX are portable, whereas SAA/PL is fully portable. The differences are very minor and are limited to six built-in functions. The capability of these functions is either not totally general or is available in other built-in functions. SAA/PL is a subset of REXX.

It will be seen later in this section that TSO/E implemented many of the CMS commands as commands of its own to provide additional portability between these two SAA environments, although these commands are not available in any other SAA environment.

Alternative TSO/E command language (CLIST). Implementing REXX as an alternative command language in TSO/E provides a language functionally equivalent to CLIST and takes advantage of the advances made in languages over the last few years.

Prior to TSO/E Version 2, Release 1,[11] the only command language available in TSO/E was CLIST. CLIST was introduced in the early 1970s along with TSO (Time Sharing Option), the predecessor of TSO/E. At its introduction, CLIST allowed a user to group TSO commands into a sequential data set or a member of a partitioned data set (PDS). That group of commands could then be executed one after another by naming the sequential data set or member of the PDS.

From that early definition, the CLIST language has been expanded over the years to become a full interpretive programming language in addition to being the command language for TSO, and later, TSO/E. The features that were added include variables and repetitive variable substitution, decision-making, some looping, error recovery, structured programming primitives, and subroutines.[6] As the language grew, however, it became more difficult to extend the syntax and keep it linguistically consistent. Because of these historic developments, the CLIST language contains certain ambiguities, particularly in the area of repeated substitution of variables and scanning of statements.

General-purpose language. REXX is an effective language for writing general applications in both TSO/E address spaces and non-TSO/E address spaces. The relationship of CLIST and REXX in TSO/E and MVS is shown in Figure 2. As can be seen, REXX can execute within TSO/E or outside of it, or stated more generally, REXX can be interpreted in any address space at any time. Although the language is not system- or address-space dependent, some supporting services such as terminal support, recovery, and tasking support are very dependent on the host address space. In TSO/E, terminal support, for example, is provided, and the interpreter must fit into the existing TSO/E structure. Outside of TSO/E the equivalent service must be provided.

General MVS command language. TSO/E is not the only product in MVS requiring a command language. Other products also require command languages and, in some cases, already support command languages of their own. Because of the extensibility built into the REXX language, it can be adopted by many products as the command language without those

products having to add any special support other than handling their own commands. One example of a product using REXX as a command language is NetView™. It operates in its own address space, a non-TSO/E address space, and uses REXX as its command language.

CMS compatibility. Many customer installations support both TSO/E and CMS on different machines in the same computer complex, and many users and system programmers use or support both TSO/E and CMS alternately in their daily work. Both TSO/E and CMS are SAA environments, so compatibility between these two systems is guaranteed for functions defined by SAA. As stated earlier, SAA/PL being part of SAA and REXX and SAA/PL being virtually identical ensure that the syntax of the language is compatible with TSO/E and CMS. One set of commands used very frequently in CMS are the stack commands. In addition to the stack commands, some programming interfaces supported in TSO/E were modeled after CMS, even to the point of providing duplicate interfaces in TSO/E, one TSO/E-like, the other CMS-like. These new interfaces and the inclusion of stack commands in TSO/E reduce the effort required to convert REXX applications between TSO/E and CMS. ISPF commands are also compatible with TSO/E and CMS, allowing a large percentage of ISPF dialogs to be ported without changes.

Extensive customization was needed to handle differences in MVS address spaces.

Customized execution environment. A very general requirement was the ability to customize the execution environment for a REXX exec. Extensive customization was needed to handle differences in MVS address spaces such as conventions for obtaining and freeing storage, reading and writing to the terminal, and other system-related services. The design for REXX support had to include the convention of each of these address spaces. There had to be a focal point for such customization, and this need led to the concept of the *language processor environment.*

Need for a language processor environment. It has been shown that the REXX interpreter can be called in different address spaces and that the implementation of these services in these address spaces differs greatly. To accommodate these differences, either the interpreter must be sensitive to the differences or the interpreter must run within a newly created *environment* which hides the differences.

In the first design, customization parameters were passed to the interpreter when an exec was to be interpreted, and the interpreter handled the differences between services in different address spaces. After some analysis and after the number of customization parameters had increased significantly, it was found that performance could be improved by processing the customization parameters once, retaining the result in a data structure representing the new *environment*, and passing that data structure to the interpreter. The environment created for the interpretation of a REXX exec is called a *language processor environment*, and this *environment* is required before an exec can be interpreted.

The interpreter can only interpret an exec within an environment. That environment established for the interpreter is address-space independent and shields the interpreter from having to be sensitive to differences in the underlying system. It allows the execution environment to be customized and to handle all differences between services provided in different address spaces. For example, in TSO/E the input from the user is expected to come from the terminal, and output to the user again is sent to the terminal. In a non-TSO/E situation, this I/O activity may need to be read from or written to a file or may be handled by the terminal-handling routine of another product. The language processor environment handles the routing to and from the different places and presents the interpreter with a consistent interface in all cases.

Because of the tasking structure of MVS, it was decided early in the design cycle that one language processor environment should be associated with one task. But if one task could be associated with multiple language processor environments, tying a language processor environment to an MVS task allows additional, task-related, information to be associated with an environment. This type of information includes routines which have been loaded

into storage, addresses of control blocks related to open data sets, locks for multitasking, anchors for variable pools, and the data stack (see the subsection, "Stack design"). As discussed later, other language-related information is also associated with the language processor environment. Bringing all these anchors together under the language processor environment made it the concept around which the design solidified.

Design alternatives. Several alternatives were considered in establishing a language processor environment. In one alternative, small customized interface programs for each service and each type of address space were to be created. When a service was required, the interpreter could determine the type of host address space and call the appropriate customizing routine for that service. The disadvantage of this alternative was the implementation cost and the need to maintain many customized routines, one per service and type of address space. It would also be difficult to maintain absolute compatibility among these different routines in the different types of address spaces. On the positive side, it would perform better and require less customization than other alternatives. On the negative side, it could prevent an installation from overriding system defaults of an application or prevent an installation from adding new routines to support additional types of address spaces. It was felt that this alternative was not sufficiently general for the non-TSO/E address spaces but was adopted for the TSO/E address space. In the TSO/E address space, special interfaces to the TSO/E input stack and to authorized commands were needed, which could not be generalized. Also the sharing of storage across all MVS tasks in TSO/E allowed certain performance optimizations not applicable to other address spaces.

Another alternative was to build the environment just prior to interpreting an exec and to take it down when the exec completes. The advantage of this approach is complete flexibility but at the cost of significantly degrading performance to a point which could not be tolerated.

Yet another alternative considered was the creation of a service or services to establish and/or delete a language processor environment. The language processor environment would have a long life and thereby ensure better performance. The initialization program could be table-driven, permitting the user, installation, and product to tailor the execution environment.

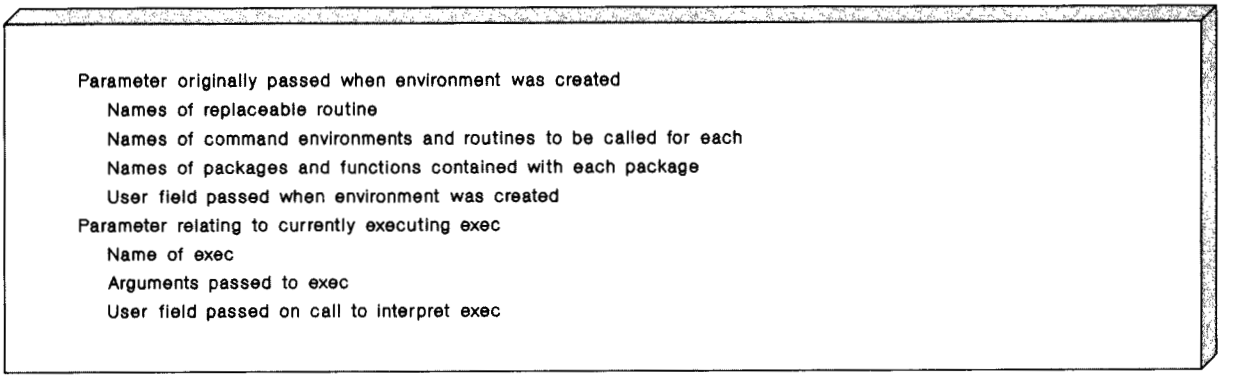
This direction was taken in the design for environments built outside of a TSO/E address space. This alternative does not have the advantage of each exec customizing its own environment, but it was felt that such a degree of flexibility was not needed. Its main advantages are (1) the path length is significantly shorter than the previous alternatives when starting the interpretation of an exec, and (2) it allows for a single, general-purpose implementation for all address spaces.

The language processor environment is the key to connect REXX to the system.

The last alternative was the basis for the design in non-TSO/E situations. The first was chosen for the TSO/E address space, and a few parameters are allowed when calling for the execution of an exec, which is somewhat similar to the second alternative. Thus, the final design contains parts of each of the alternatives.

It has been shown before that whenever an exec is interpreted, an environment must exist. If none has been initialized when an exec is to be interpreted, an environment is built "automatically." The values required to establish a language processor environment could not be fixed by the design because of the differences among customer installations and address spaces. Instead, parameters are obtained from a module with a fixed name. That module is loaded at the time the environment is built and, because of the flexibility in the MVS loading process, is not limited to one set of values per installation. Many modules with the same name may reside on one system in different data sets; the module to be loaded depends on data set allocation in the link list. Each of these modules may define a different set of parameters, creating a different type of environment. If an environment was created automatically to run an exec, the environment is automatically deleted when

Figure 3 Parameters stored in language processor environment block



the exec terminates. The result of this design is that a user can call for the execution of an exec, totally unaware of the existence of a language processor environment.

As part of TSO/E, three such parameter modules, containing different sets of parameters for three different types of language processor environments in different types of address spaces are provided. One module contains the default parameters for the TSO/E session. It is used by the LOGON command. A different module provides parameters when a user enters ISPF, and a third is for language processor environments created outside of the TSO/E address space.

The makeup of a language processor environment. The language processor environment holds many parameters. Among them are the following: the anchor for the data stack, pointers to storage control blocks, and input/output-related control blocks (for example, the Data Control Block, or DCB). The language processor environment is also the focal point for other operating-system-related constructs such as loaded modules. It contains the addresses and names of all routines handling the system services and pointers to execs that might have been preloaded. Because of its importance to TSO as anchor, it was felt that a language processor environment would be of equal importance as anchor for application-related information. This led to the *user token* in the language processor environment. For example, an application program may pass REXX a user token, which another part of the application may retrieve at a later time.

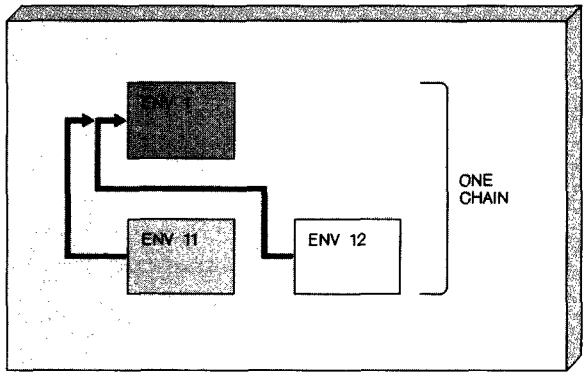
The values used in establishing the language processor environment are fetched from a *parameter module*, the name of which is optionally specified when the environment is created. These values are saved in a control block representing the language processor environment called the ENVB (environment block). The ENVB is made available as a parameter to every routine called from the REXX interpreter.

Because the language processor environment is the key to connecting REXX and the system, it was made available to every program called by the interpreter. Through the environment block, these programs can examine all parameters passed in the parameter module. Also stored in the environment block and therefore accessible to these programs are parameters passed to the interpreter when interpretation of the current exec started. A sampling of the parameters available in the ENVB are shown in Figure 3.

Although the language processor environment is important to the design and must always be present, the author of an exec need not be aware of its existence. The view of the exec programmer is that the exec executes in an environment and he/she is not or need not be sensitive to how that environment had been established or how it could be changed.

Chains of language processor environments. The previous subsection shows that many system-related properties are tied to a language processor environment. At times some of these properties need to be replaced or changed. This means either allowing an existing language processor environment to be modified or a new one to be created. It was decided that modifying existing values of a language processor environment would be error prone, so the design only allows the creation of new language processor environments.

Figure 4 Chains of language processor environments



New environments can be created at any time. At the time the creation service is called, an extensive set of parameters is passed, which defines all values for the new environment. Environments are related hierarchically in what is known as a *chain of language processor environments.* Unless specifically changed at initialization time, the dependent environment inherits the properties and resources of the parent. If more than one dependent environment is created from a given environment, the resulting structure is, strictly speaking, a hierarchy. However, the structure is referred to as a *chain,* because in most cases the structure of related language processor environments is linear as is a chain.

Another reason for different language processor environments is the MVS tasking structure. Storage, open data sets, and other MVS resources are tracked on an MVS task level. If execs operate on different MVS tasks, they must operate in different language processor environments. The dependent environment may have the same characteristics as the parent, but it is associated with a different task and so allows resource management and recovery. The only exception is the TSO/E environment. In TSO/E an exec may attach a command, which calls a second exec. The second exec operates on a lower-level task than did the original exec, but because of the internal design of TSO/E and the sharing of virtual storage subpool 78 in TSO/E, both execs can operate in one language processor environment.

Note that such special cases made the implementation in the TSO/E address space different from the implementations in other address spaces. However, none of these differences are visible to the user of REXX or the user of any of the interfaces. They are contained internally.

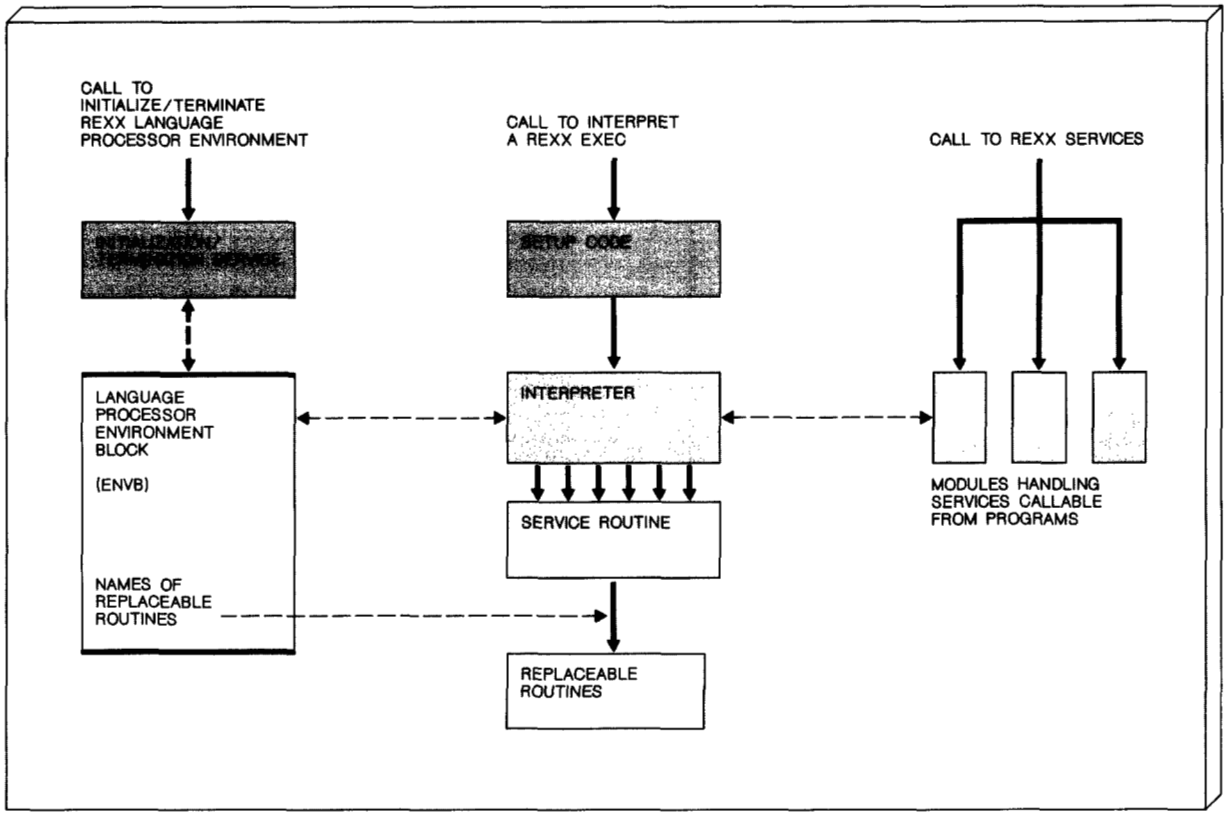
Different chains are completely independent from one another. Figure 4 shows one chain. Env 1 was the original environment, then two additional environments Env 11 and Env 12 were created under it. This structure is present for ISPF, for example, where the top environment represents the TSO/E READY mode, and the dependent environments correspond to the two applications executing in split screen mode. In the READY mode, only one environment is present, such as Env 1, which would represent a degenerate case of a chain.

Typically only one chain exists in one address space at any one point in time. However, an arbitrary number of independent chains may be created in an address space. (It is even possible, using a special technique called *reentrant language processor environments,* to support an arbitrary number of chains of environments on a single task. This topic will not be expanded here.) In the figure, chains are identified by pointers from each language processor environment to its parent. A language processor environment with no parent is the *head of a chain.* When a head-of-a-chain environment is created and values are not explicitly specified on the call at initialization time or in the parameter module that may be specified, the values default to system default values. When a language processor environment that is not the head of a chain is created, parameters for the new environment are taken from the parent environment, unless a new value is specified either on the call to the initialization routine or in the parameter module passed on that call to the initialization routine.

REXX in any address space. The design implements three basic types of calls supporting REXX functions: initialization and termination of language processor environment, interpretation of an exec, and services. They are shown in Figure 5.

The *initialization* routine establishes a new language processor environment by creating a number of control blocks, among them the *environment block* (ENVB). Execs can now be interpreted in this newly created environment. As stated earlier, the environment determines which routines are to handle system services and commands. These service routines replace default system services and are therefore called *replaceable routines,* described later. Their names are saved in the environment block. The *termination* routine reverses the action of the initialization routine by deleting the current language processor environment.

Figure 5 Main components of REXX



Two different services support *interpretation* of an exec. They differ primarily in the format of the parameters on the call. One is a general service supporting many parameters and arguments to be specified, and the other permits only the name of the exec and one argument. The latter service is tailored to be called from the EXEC statement of the MVS job control language (JCL). The *EXEC statement* is used to identify the first program to be executed in a batch address space; in this case it is the interpreter. The parameter sent to the interpreter is the PARM on the EXEC statement. It defines the REXX exec to be interpreted and the arguments for that exec.

When an interpretation service is called, some minor setup steps are performed. For example, the current language processor environment is located, and if none exists, a default environment is created. If the exec has not already been loaded, it is loaded. (This condition exists when the interpreter is called from the JCL EXEC statement.) After the setup the *interpreter* is called. It is the heart of the product. The interpreter is the code that interprets each instruction of the language, maintains variables, and issues the language-related error messages. The code for the interpreter is the same code used by CMS, thereby saving the cost of implementation and ensuring total compatibility between the TSO/E and CMS implementations of REXX.

Whenever this common code requires services from the operating system, *service routines* are called. Service routines are different for TSO/E and CMS. The design defines a number of REXX service routines. Examples of such services are fetching or setting variables, requesting stack services, performing I/O functions from programs, or loading execs.

Replaceable routines or exit routines. It was shown earlier in this paper that a main emphasis of the design was the ability to tailor the language processor environment to the host address space. This is accomplished by allowing every system service to be

funneled through a routine appropriate for the host address space.

One approach to tailoring is to use exit routines. When an exit routine is called, control goes to the exit routine, along with certain parameters. Depending on the design of the interface to the exit routine, the capability of that routine may intentionally or unintentionally be limited. When the exit routine returns control, the system may either continue or terminate the requested service.

The replaceable routine is a more general approach than exit routines.

Early in the formulation of the design, exit routines were considered but were abandoned because the underlying assumption of an exit routine is that it modifies an exiting function and does not replace it. The degree of customization needed for this design was such that any of the system services were not to be modified but totally replaced. Thus it was necessary for a language processor environment to *replace* the routine performing the system service. This reasoning lead to the concept of the *replaceable routine.*

The replaceable routine is a more general approach than exit routines because it ensures that all parameters needed to provide the service are passed to the routine. This replaceable routine may check, change, or ignore input parameters, provide either full service or partial service, refuse to give the service at all, call another routine to perform the requested service for all or some cases, and either accept the result of that called routine or ignore it. In short, it has complete control.

The types of services handled by replaceable routines are loading of execs, generalized I/O functions, storage management, and data stack. The name of each of the replaceable routines is stored in each language processor environment and, once specified, may not be changed. When any part of the REXX component of TSO/E requires one of these services, the appropriate routine is called; when control returns to the caller, the service has been performed. For each of these services at least one routine is supplied; at times one is supplied for the TSO/E address space, and another is supplied outside of the TSO/E address space.

Products or subsystems frequently implement an independent layer of service routines for their own internal use. A unique feature of TSO/E is that this independent layer is externalized, because the service routines can be called by any program wherever a language processor environment has been established.

Replaceable routines are only used for those services mentioned above. For other functions to be tailored, the design used exit routines. For example, exit routines were used for preinitialization and post-initialization and during termination of a language processor environment.

Stack design. Several different stacks are now discussed. To distinguish them, the existing stack in TSO/E will be referred to as the *TSO/E input stack*, the existing stack in CMS will be referred to as the *CMS stack*, and the new stack created for this design will be called either the *data stack* or simply the *stack* if there is no confusion about which stack is being referred to.

The language defines a number of instructions that apply to a stack or queue. The PULL instruction removes the top element from the stack and returns it to the caller. The PUSH instruction adds a new element above the old top; the new element supplied by the caller becomes the new top. The QUEUE instruction adds an element below the old bottom element, and the new element becomes the new bottom. A built-in function, QUEUED, returns the number of elements on the stack.

This construct, which we call a stack, has a *top*, a *bottom*, and an arbitrary number of elements between those extremes. It has a length, is read from the top to the bottom, and, when an element is read, is removed from the stack.

Neither MVS nor TSO/E supported such a stack function prior to the implementation of REXX, although TSO/E uses a construct called the TSO/E input stack. This stack is very different from the data stack re-

quired by REXX, which meant a new stacking function had to be implemented.

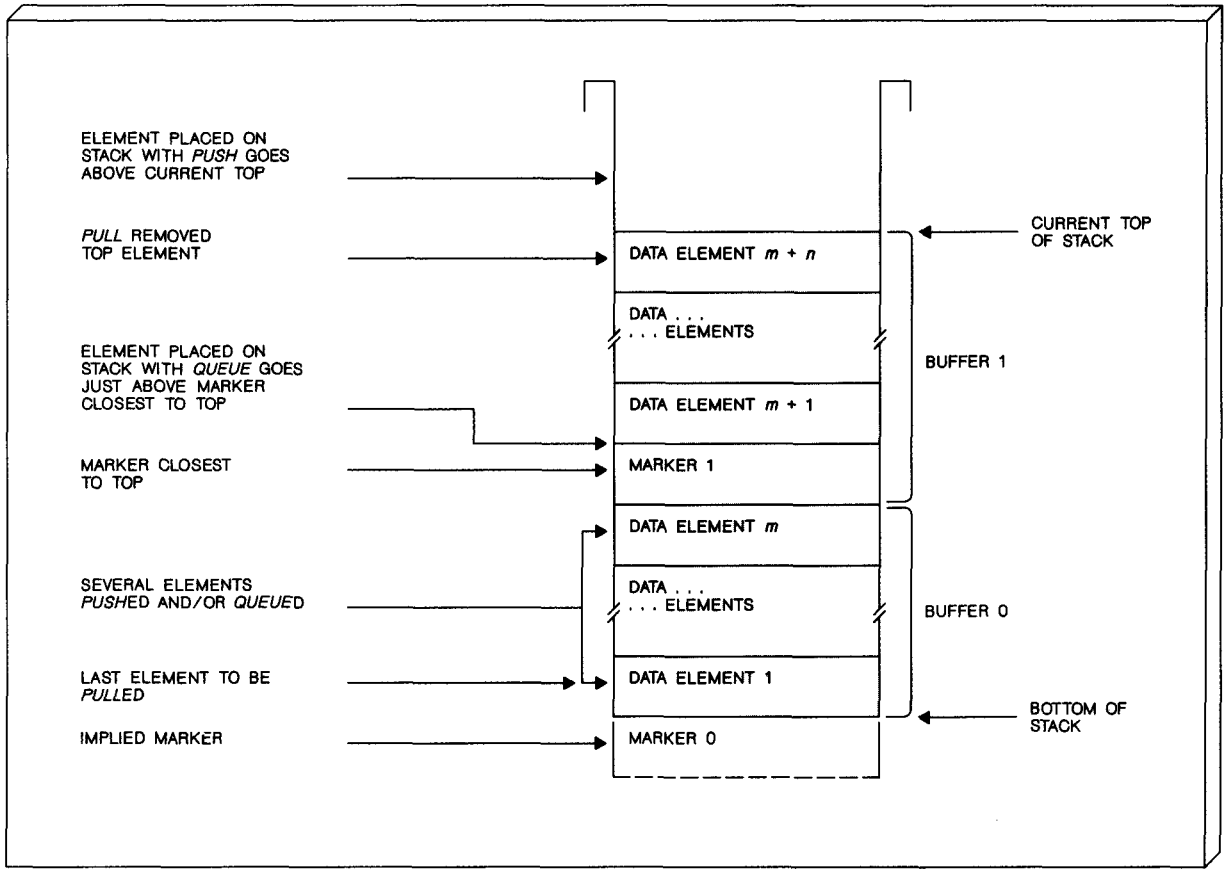
As stated earlier, REXX was designed to execute in both the TSO/E and non-TSO/E address spaces, so this new stacking function needed to be implemented in all address spaces. Outside of a TSO/E address space, any stack model could have been used, but rather than choose a new model, the CMS model was chosen because it ensures added compatibility between REXX in any MVS address space and CMS. This degree of compatibility is beyond the SAA requirement but was deemed to be important. Whatever stack model was to be used in the non-TSO/E address space had to be consistent with the new stack in the TSO/E address space, which in turn had to be compatible with the existing TSO/E input stack.

The starting point for the new data stack in a non-TSO/E address space was the CMS stack. Before describing this new data stack, the CMS model is described.

The CMS data stack. The CMS stack is a part of the CMS system, not the REXX language, and holds data placed on it by REXX execs or programs. In addition to data, an exec or program may place markers on the stack, allowing an exec or program to remove all elements above, and including, a given marker. These markers define parts of the stack called *buffers*. SAA does not define a stack or buffers, which means the TSO/E usage of the CMS stack provides cross-system compatibility beyond the SAA base.

In the example of a CMS stack shown in Figure 6, *m* elements were placed on the stack, followed by a *marker* and by additional elements. Each element is a character string with arbitrary length, up to the implementation limit, which in TSO/E is almost 16 megabytes. A null string—a string with zero length—

Figure 6 Example of CMS stack



is a valid element. If the stack is in the state shown in the figure and an exec PUSHes an element, the element would be placed above the top data element, element $m + n$. If an exec QUEUEs an element, the element would be placed just above the marker closest to the top, which means that in the case of the figure, the new element would be added between element $m + 1$ and marker 1. One can think of a marker 0 being on the bottom of the stack.

Elements between two markers and between the top marker and the top of the stack are frequently referred to as *buffers*. This term gives rise to the commands MAKEBUF and DROPBUF which add and remove markers. A PULL instruction returns the top data element on the stack, removing any markers which may be present between the top of the stack and the top data element.

Thus, in terms of buffers, the PUSH instruction adds an element to the top of the top buffer, the QUEUE instruction adds an element to the bottom of the top buffer, and the PULL instruction removes and discards any markers above the top element, then removes the top element and returns the element to its caller.

If the stack were in the state shown in Figure 6 and the REXX built-in function QUEUED is called, the number of elements, in this case $m + n$, is returned. QUEUED does not count markers on the stack.

In CMS the stack also participates in reading input from the terminal. A REXX exec may request input from the terminal directly, but the terminal can also be thought of as an extension of the stack. If a REXX exec executes a PULL instruction, the element is taken off the top of the stack. If, however, the stack is empty, the process continues, and data are read from the terminal. PULL is a two-step process and will always return an element. It should be noted again, a null element is a valid element.

Basic design of the TSO/E data stack. As stated earlier, the starting point for the design of the TSO/E data stack was the CMS stack. From this stack, a CMS-like data stack was developed that satisfied all requirements placed on it by the REXX language. However, several extensions had to be made to that model of the stack.

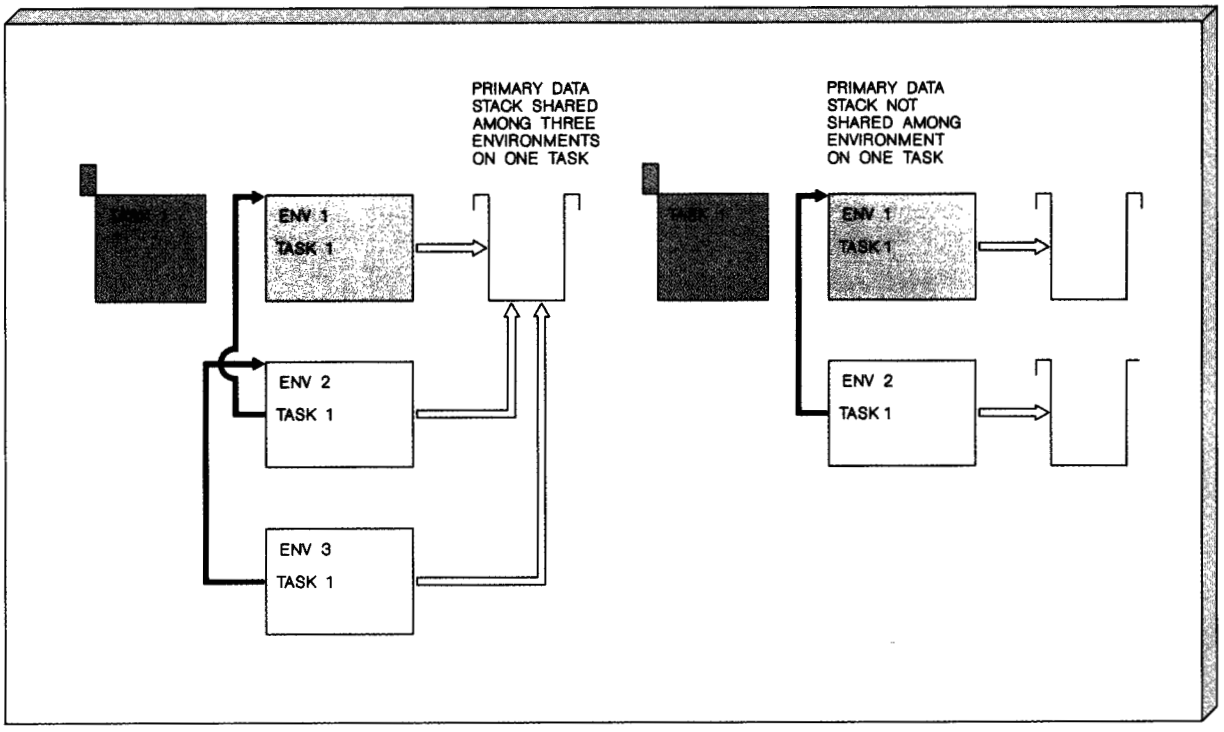
It was shown earlier that the design of REXX in the non-TSO/E address space allows for multiple language processor environments to be created in a hierarchical relationship. One of the services associated with such an environment is the data stack. If two language processor environments are created on the same MVS task, there cannot be a synchronization problem for the simultaneous update of the data stack, because only one program can execute at any one time on one task. Therefore, execs in the lower language processor environment must complete before the ones in the upper language processor environment continue. Because no general rule can be made about the sharing of stacks for execs executing on these two levels, the design allowed the sharing of the stack to be specified at the time the lower-level language processor environment is created. If the data stacks are to be shared, the lower language processor environment does not contain a stack, but any stack-oriented command issued by an exec executing on the lower environment is directed to the data stack associated with the higher-level language processor environment. The left side of Figure 7 shows how three environments, Env 1, its dependent Env 2, and Env 3, share one data stack. All environments share the stack and execute under one task, Task 1.

No general rule can be made on sharing of data stacks among execs on different tasks.

If the stack is not to be shared, a new stack is initialized. Then the lower-level language processor environment is built. This isolates the stack associated with the higher-level language processor environment from an exec or program executing in the lower one. (See right side of Figure 7.) Note that the word *primary* data stack implies *secondary* data stacks in Figure 7. These secondary stacks will be discussed later.

The sharing of the data stack between a language processor environment and its parent is established at the time the dependent language processor environment is created. A language processor environ-

Figure 7 Sharing primary data stack under one task



ment can only share its data stack with that of the immediate parent. Only if that parent shares a stack with its parent will all three environments share a stack.

Stack and multitasking. The CMS-like data stack was sufficient for the non-TSO/E address space if multitasking was not present. However, the model had to be extended because REXX execs could execute concurrently on multiple tasks, yet needed to share the data stack. It was shown earlier that the environment blocks and the associated language processor environments contain task-related data such as anchors to store, open data sets, and the like. So, if execs are to be interpreted on different tasks, they need to be interpreted in different environments, at least one per task.

Such parallel processing is a fundamental concept in MVS and had to be fully supported. If execs execute in different environments on the *same* task, only one exec can be active at one time, and all execution is done synchronously. If, however, different language processor environments sharing one data stack are associated with *different* tasks, the stack becomes a resource shared among execs executing on different tasks, and execs executing in these environments operate asynchronously.

Sharing the stack among asynchronously executing execs has several implications. The first one is that two execs executing on different tasks may execute at the same time and access the shared data stack at the same time. The sharing of the data stack allows two execs to communicate via the stack by PUSHing and PULLing elements on the stack. In contrast, sharing the data stack implied a lack of its privacy. An exec on one language processor environment PUSHing a series of entries on the stack has no guarantee that another exec may not PUSH entries at the same time, which results in the entries from the two execs being intermixed and the content of the stack being unpredictable. When the original entries are PULLed, unwanted and possibly unrecognized entries would be returned from the stack.

An example of where such mixing would result in errors is the case in which an exec stacked entries via an I/O command called EXECIO, which reads a number of records and places them on the stack. If

one exec had issued the EXECIO command and had processed some, but not all, entries on the stack, and another exec were to either place entries on the stack or pull some off, errors would occur.

Another side effect of multitasking is the need to lock the data stacks. Locking of stacks had to be considered in the design. To simplify the design and to optimize performance, locking was done on the basis of chains of environments. Although it is not the smallest possible scope for a lock, this level was chosen because two chains are guaranteed to be independent. It was also felt that there would be a minimum of interference on the lock, that it would reduce implementation cost, and that performance would be improved because extensive tests for the scope of locking would be avoided.

The CMS model of the stack is a proper subset of the TSO/E model.

The privacy issue was more difficult to resolve. One solution was to create new environments when the state of sharing a stack was to change. This approach was considered to be not sufficiently dynamic, because one exec may at times wish to share its stack and at other times wish not to share the data stack with another exec. The decision of sharing had to be made at execution time, which led to the introduction of *secondary stacks.*

A secondary stack is created by the NEWSTACK command, which deactivates but does not change the currently active stack after NEWSTACK is issued. This secondary stack is the only active stack for any exec in that language processor environment, and a secondary stack is never shared among language processor environments but is shared among all execs in the environment in which it was created. Multiple secondary stacks can be created, but at any one time only the last stack created is active and accessible to an exec. The primary stack and all secondary stacks can be thought of as a stack of stacks, the last stack created being the only stack accessible to the execs.

A secondary stack has the same properties as the primary stack. Entries can be PUSHed on the stack and PULLed from the stack, and when the stack is empty, the input is read from the input file (or terminal). Markers have the same meaning on a secondary stack as they do on a primary stack.

Another command, DELSTACK, deletes the last secondary stack and reactivates the previous stack. Should a DELSTACK command be issued while the primary stack is active, the command is ignored, because the primary stack may be shared with execs executing in other language processor environments, and deleting the shared stack could cause the failure of execs associated with another language processor environment.

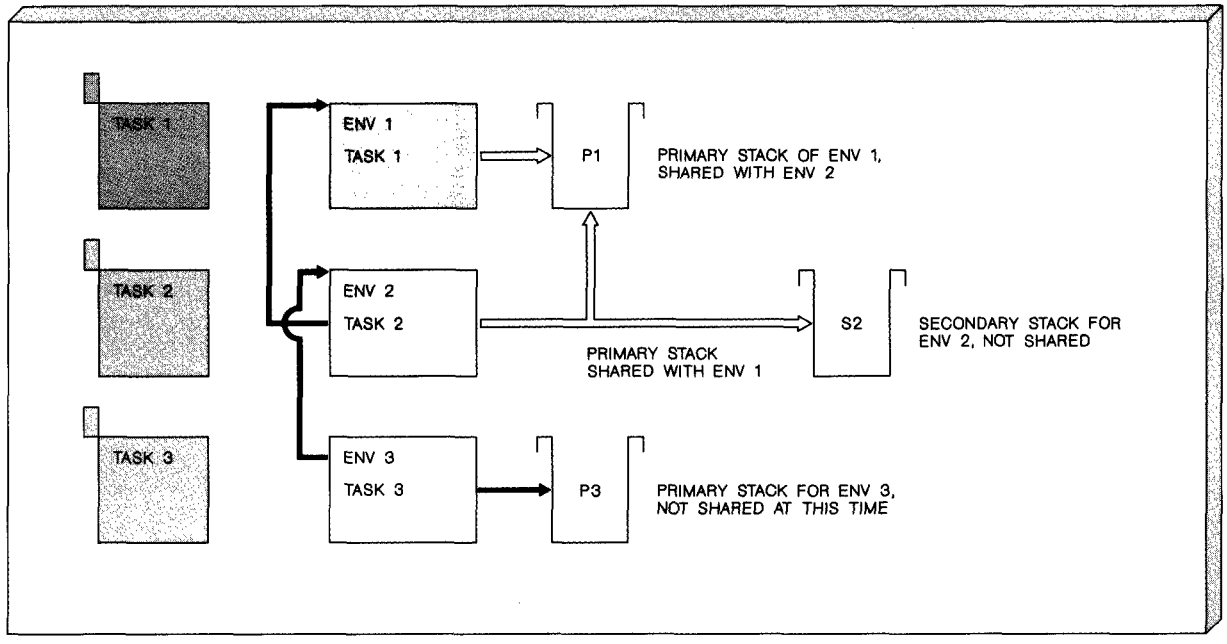
To summarize, the CMS stack model has been generalized in two directions for TSO/E. First, data stacks can be shared among different language processor environments, a concept with no parallel in CMS, and second, new stacks can be created to ensure privacy in multiprocessing, again a concept not supported by CMS. But to an exec being interpreted in an environment, only one stack is visible, and that stack appears identical to the CMS stack. This means that the CMS model of the stack is a proper subset of this design of the TSO/E model, and when seen from the viewpoint of the exec, the data stack is identical to the CMS stack. The two stacks have an identical appearance to the exec because at any one time, one exec can access only one data stack.

Figure 8 shows an example of a chain of three language processor environments. Env 1, associated with Task 1, owns a stack P1. That stack is the only stack for Env 1 and therefore is a primary stack and the active stack for Env 1. When Env 2 was created, no new primary stack was created, but the environment was to share the primary stack with Env 1. However, some exec or program created a private, or secondary, stack S2, which is the currently active stack for Env 2. After S2 is deleted, P1 will become the active stack for Env 2. Env 3 has its own primary stack P3. Env 3 could create a secondary stack, but regardless of any actions in any environment, Env 3 cannot share a stack with either of the other two environments.

The TSO/E input stack

Before discussing how the data stack was integrated into TSO/E, we need to describe the function of the TSO/E input stack prior to the incorporation of REXX.

Figure 8 Sharing data stacks among different tasks



The input stack is created when a user first logs on; it is initially empty. It is continuously examined by a program, called the *terminal monitor program*, or TMP. The TMP, the main controlling program in TSO/E, is implemented as a two-step loop:

1. Read input
   a. If the input stack is not empty, execute all CLIST statements up to the first command, return that command, and continue with step 2.
   b. If the input stack is empty, read a command or CLIST name from the terminal and continue with step 2.
2. Process the command
   a. If the command is LOGOFF, terminate the TSO/E session and exit.
   b. If the input is a program (a command), execute it and continue with step 1.
   c. If the input is the name of a CLIST, load it, place a pointer to the loaded CLIST on the input stack, and continue with step 1.

We now present an example. Initially after logon, the input stack is empty, and the TMP looks for terminal input. When the user enters a command, the TMP first tries to locate an executable module. If a command is found, it is executed. When the command completes, the terminal again calls for another command from the terminal. If the command was not an executable module, it must be a CLIST, so the TMP calls for the loading of the CLIST and places a pointer on the input stack. Once the CLIST is stacked on the input stack, it is processed one statement at a time.

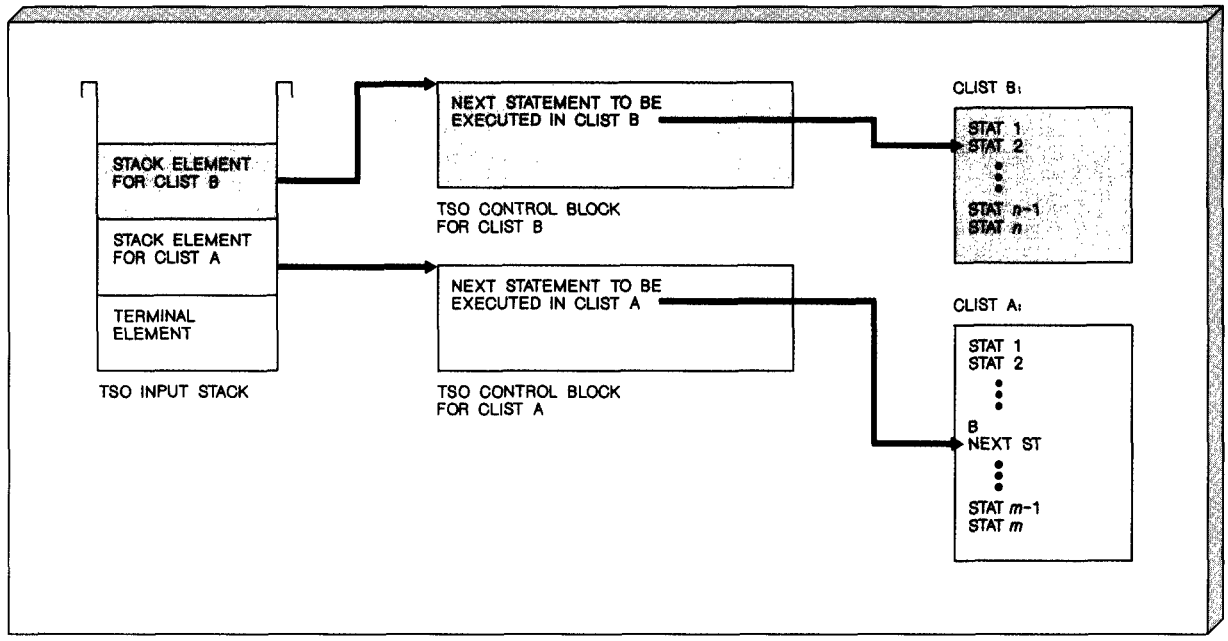
On the assumption that one of the statements in that CLIST is the name of another CLIST in our example, the second CLIST is again loaded and a pointer stacked above the first pointer on the input stack. After the stacking, the TMP again executes step 1 by processing the first statement of the second CLIST, followed by the second statement of the second CLIST, and so on. This condition is shown in Figure 9.

The first element on every input stack is a different type of element, called a *terminal element*, or TE, which, when read by the TMP, indicates that the bottom of the input stack has been reached and the input is to be obtained from the terminal.

Data stack in the TSO/E address space

Reconciling two totally different stack models—the stack model based in CMS and extended for the non-

Figure 9 Example of CLIST elements on TSO/E input stack



TSO/E address space and the TSO/E input stack model—presented the biggest challenge in the integration of REXX into TSO/E.

The first change was to make the TMP sensitive to the data stack. It was necessary because if an exec PUSHed a command on the data stack and terminated, that command had to be executed by the TMP before calling for input from the terminal. The change was accomplished by modifying step 1 of the TMP to add a step to pull data from the data stack before reading terminal input.

The design had to allow REXX execs to call CLISTs and CLISTs to call REXX execs. This arrangement meant that in step 2 a pointer to a REXX exec had to be stacked on the input stack, much like pointers pointing to a CLIST. The REXX interpreter does not interpret one REXX instruction at a time, but an entire exec, so it was necessary to call the interpreter at this point in step 2. The structure of the phases could not be changed because CLIST processing could not be changed, so step 2 performs the normal loading process for both CLISTs and execs, although the internal formats are different. At the end of loading, if the top element on the TSO/E input stack corresponds to an exec, the interpreter is called from within one pointer through the TMP. CLIST is a two-phase process; REXX is a one-phase process.

When the REXX interpreter encounters a command within an exec, it calls the appropriate routine based on the currently active address command environment. The name of that routine is found in a field located in the language processor environment. If the address environment is TSO/E, the TSO/E service facility is called. In past releases, the TSO/E service facility was only intended for authorized commands, but for this release of TSO/E it has been expanded to process CLISTs, execs, or commands and return only after the CLIST, exec, or command completed. The TSO/E service facility thus has been expanded to be an unauthorized TMP which can be called by any program in TSO/E.

Calling this new TMP from within the interpreter creates a totally different call structure for REXX than for CLIST, because in CLIST the same invocation of the TMP interprets all statements of all CLISTs, whereas the REXX interpreter is invoked for each exec. Because of the call chain created by the REXX interpreters, a new TMP had to be written. This new TMP is called whenever a TSO/E command is detected in a REXX exec. It has the exact same function as the original TMP, although in the implementation it is new code.

The modifications above resulted in the following changes in the two-step loop of the TMP:

1. Read input
   a. If the input stack is not empty, execute all CLIST statements up to the first command, return that command, and continue with step 2.
   b. If the input stack is empty and the data stack not empty, pull the top entry off the data stack and continue with step 2.
   c. If both stacks are empty, read a command or name of a CLIST or REXX exec from the terminal and continue with step 2.
2. Process the command
   a. If the command is LOGOFF, terminate the TSO/E session and exit.
   b. If the input is a program (a command), execute it and continue with step 1.
   c. If the input is the name of a CLIST or exec, load it, place a pointer to the loaded CLIST or exec on the input stack, and, if a CLIST, continue with step 1.
   d. Call the interpreter, and when interpretation is completed, continue with step 1.

One more change had to be made to TSO/E to allow program commands to be sensitive to the data stack. In the CMS stack model and in the definition of the REXX language, the data stack is examined before the terminal is read. This change was made by intercepting all calls for input to the terminal, and if elements were on the data stack, pulling the top element off the data stack and returning it to the caller. In TSO/E, the routine that is called by commands to read data from the terminal is GETLINE. This routine was modified to examine the data stack before reading the terminal. If the data stack was not empty, the top element would be returned to the caller. If the data stack was empty, the terminal would be read and the terminal data returned to the caller. With these changes, an exec can place input for a command on the stack and call a command, and the command reads the data as though they came from the terminal. The command can be an old command which now takes advantage of the new REXX data stack without having to be changed.

This design also assured one other mandate, namely that the new design with old CLISTs and old commands would be totally compatible to previous releases of TSO/E.

Performance considerations

Performance of REXX on TSO/E was a major consideration during the design process. It was shown earlier that the concept of a language processor environment was created in part to improve performance on the critical path of interpreting an exec. Several functions were added to allow a user to *tune* performance, but in each case, defaults were chosen for the most common cases. The defaults allow the typical user to ignore tuning.

One of the basic tradeoffs in any design is to choose between the size of the code and performance. In this product, better performance was traded at the cost of larger code size (number of bytes required by the code).

In addition to the externally available functions, code was written to optimize internal performance. For example, storage management code was included, which is called by all internal routines and which is capable of handling the allocation and freeing storage in arbitrary sizes from an arbitrary number of MVS subpools either above or below 16 megabytes. The storage was associated with a language processor environment. This storage management code was needed because storage management is one of the replaceable routines, and if that replaceable module were called for each individual storage request, performance would suffer. This internal storage management code acquires storage in multiple pages and doles it out to internal requests, frequently eight and twelve bytes at a time. The path length for the internal get and free main storage routines is very short. These internal storage management routines are based on the "Radix Partitioned Tree Algorithm."[12]

External functions supporting performance. Performance of an application can be optimized by managing the loading and freeing frequently used functions and subroutines. An application has three options. For one, it can load an exec and pass the address of the exec to the interpreter. Another option is that the application call the replaceable routine for loading of execs directly and request the loading and at a later time call for its interpretation, passing the address of the preloaded exec. Yet another option is for the application not to preload or to call any other service, but to simply call for the execution of the exec.

In the first case, the interpreter performs no input or output operations. The caller is responsible for creating an image of the exec in storage and passing it to the interpreter. The caller is also responsible for freeing the storage the exec resides in. From the point of view of the interpreter, this option is the

best performing because the load process is totally omitted.

In the second case, the interpreter is instructed to LOAD an exec and retain it for later use. Whenever that exec is to be interpreted, the interpreter will scan the list of execs that have been loaded and, if found, will interpret the exec without reloading it.

Function packages are an additional method of improving performance of applications.

The program that called for the loading of the exec may call a service to FREE (unload) the exec. Loaded execs are associated with the language processor environment, and when the environment is deleted, the execs associated with the environment are automatically freed.

In the third case, the interpreter will again scan the list of loaded execs and, if found, use the exec without reloading it. If the exec cannot be found, it is loaded and internally retained as in the previous case. The execs are freed when the language processor environment is deleted or if the data set from which they are fetched is closed.

Performance of interpretation in a TSO/E address space. The design of the REXX interpreter was made distinctly different from the CLIST interpreter partly to improve performance of the REXX interpreter, and partly because the REXX interpreter cannot suspend operations when it encounters a command as does the CLIST interpreter (see earlier section, "Data stack in the TSO/E address space"). As shown in that section, when the TMP executes a CLIST, it executes it in two passes though the TMP code. The first step is to execute the command calling for the interpretation of the CLIST (for example, the TSO/E EXEC command). This command loads the CLIST into storage, places a CLIST entry on the TSO/E *input stack*, and returns to the TMP. This pass is called *phase 1*. After phase 1 completes, the second pass, *phase 2*, executes the CLIST. Executing a REXX exec is a one-phase process, thereby improving the overhead associated with cycling through the TMP a second time.

Execs can be stored in data sets allocated to either SYSEXEC or SYSPROC. SYSEXEC is searched first, followed by SYSPROC. The searching of the additional files degrades performance but is necessary to distinguish the difference between some execs and CLISTs. To prevent degradation of performance, searching SYSEXEC is optional, allowing the user to bypass a search of that file. Bypassing the search is also the most efficient method of using the virtual lookaside facility, a method of retaining CLISTs or execs in storage. It assures the shortest path by avoiding the input operation needed for loading. Yet if mostly execs are used, they can be stored in SYSEXEC, which remains open, avoiding the OPEN/CLOSE forced by CLIST.

Function packages. Function packages are an additional method of improving performance of applications written in REXX. The packages are not part of the REXX language or SAA/PL; however, they are implemented in CMS. CMS allows one or more functions or subroutines commonly called by execs to be *packaged* and made quickly accessible, which improves performance. The performance gain is achieved by preloading the entire package once and retaining it in storage, thereby preventing the need for multiple loads, one per function, and repeated loads, one per invocation. Additional performance is gained because these packages are first in the search sequence for functions or subroutines. Usually service routines are placed into packages, but the package capability is sufficiently general to allow any program called as a function or subroutine to be placed in a package. (Execs cannot be placed into packages.) The design of packages and the interfaces to the package are generalizations of the CMS implementation, again providing a great deal of compatibility between these two systems.

A package is associated with a language processor environment and is inherited from the parent language processor environment. The design allows for three types, or levels, of packages. Each level can hold multiple packages. Each package may contain multiple functions or subroutines. It is expected that products will provide special REXX functions or subroutines to create their own packages. On the basis of this assumption, it is necessary to allow a given user to run with those packages supporting the products he or she plans to use.

Packages can be placed on one of three levels: the *user* level, the *local* level, or the *system* level. As the name implies, the first of those levels is intended for private packages written by the user. Functions in these packages are searched before either of the other levels is searched. If a function or subroutine is found on this level, the search is terminated and use of that function or subroutine is given precedence over functions or subroutines in packages on other levels. The local level is for packages supporting local applications, and again, functions and subroutines on this level have precedence over those on the system level. The system level is for packages supporting products. There is no mechanism to enforce the placement of packages on any given level; it is only a convention.

Concluding remarks

An interpreter for the REXX language (the SAA/Procedure Language) has been added to TSO/E. This interpreter is not only capable of interpreting execs (programs written in REXX) in the TSO/E address space, but in any MVS address space.

In a non-TSO address space, execs can serve as command languages for any product or application or can be used as a general-purpose programming language particularly well-suited for high productivity in creating prototypes.

If execs are interpreted in the TSO/E address space, they have all of the capabilities of CLIST (the TSO/E command language supported in previous TSO/E releases) but a much richer set of functions. They can call CLISTs, CLISTs can call execs, and the same commands can be invoked from execs as from CLISTs. For applications written in CLIST, CLISTs can be translated into execs one at a time, because externally the two cannot be distinguished.

A data stack facility has been added which is a superset of the CMS stack and which is available in *any* address space. This stack can, but need not be, shared among execs interpreting on different MVS tasks in the same address space and can be used as an additional method of sharing data among many programs and/or execs.

Execs are interpreted within their own execution environment, called the language processor environment, that allows a high degree of customization when it is first established. With this environment the creator of the environment can specify defaults and routines which intercept and/or modify virtually all system-oriented calls that optimize performance.

In both TSO/E and non-TSO/E address spaces the addition of the interpreter provides another platform upon which applications can be built for execution on MVS, for porting to or from CMS, or for porting to other SAA environments.

Cited references and notes

1. M. F. Cowlishaw, *The REXX Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1985), p. ix.
2. The TSO/E limit is just under 16 megabytes.
3. TSO/E Release 2.1 of MVS (TSO/Extensions of Multiple Virtual Storage), will be discussed in the next subsection. Release 2.1 of TSO/E is supported both as an MVS/XA™ (Multiple Virtual Storage/Extended Architecture) feature and as an MVS/ESA™ (Multiple Virtual Storage/Enterprise Systems Architecture) feature. The REXX function described in this paper applies equally to both features. See *TSO Extensions Version 2, REXX Reference*, SC28-1883, and *TSO Extensions Version 2, REXX User's Guide*, SC28-1882, IBM Corporation; available through IBM branch offices.
4. A program written using the SAA/PL or REXX language is normally referred to as a *REXX exec* or simply an *exec*. An exec is interpreted by a program called the *REXX interpreter* or simply the *interpreter*. In the CMS implementation, the interpreter is called the System Product Interpreter.[7]
5. M. F. Cowlishaw, "The design of the REXX language," *IBM Systems Journal* **23**, No. 4, 326–335 (1984).
6. *TSO Extensions Command Language Reference* SC28-1307, and *TSO Extensions Command Language Implementation and Reference*, SC28-1304, IBM Corporation; available through IBM branch offices.
7. *VM/SP System Product Interpreter Reference*, SC24-5239, and *VM/SP System Product Interpreter User's Guide*, SC24-5238, IBM Corporation; available through IBM branch offices.
8. C. E. Clark, "The facilities and evolution of MVS/ESA," *IBM Systems Journal* **28**, No. 1, 124–150 (1989).
9. *Systems Application Architecture, Common Programming Interface—Procedure Language Reference*, SC26-4358, IBM Corporation; available through IBM branch offices.
10. *Announcement of SAA/PL*, Announcement Letter 287-088, IBM Corporation; available through IBM branch offices.
11. *IBM TSO Extensions Version 2*, Announcement, April 19, 1988, IBM Corporation; available through IBM branch offices.
12. L. Woodrum, *Indirect Indexed Searching and Sorting*, IBM Corporation, U.S. Patent Number 3,611,316 (October 5, 1971); L. Woodrum, *Directory Generation System Having Efficiency Increase with Sorted Input*, IBM Corporation, U.S. Patent number 4,086,628 (April 25, 1978).

Gerhard E. Hoernes *IBM Data Systems Division, P.O. Box 390, Poughkeepsie, NY 12602.* Dr. Hoernes is a senior programmer and was responsible for the design of REXX in TSO/E. He has published papers on logic design, microprogramming, and database, is the author of a book on logic design, and is a contributor to the first edition of the *Handbook of Electronics*. He holds several patents and has received IBM awards for patent activity and his outstanding contributions. Dr. Hoernes has also taught extensively both in and outside of IBM, and has been associated with the computer science department of Vassar College.

Reprint Order No. G321-5359.