KERNFORSCHUNGSANLAGE JÜLICH GmbH

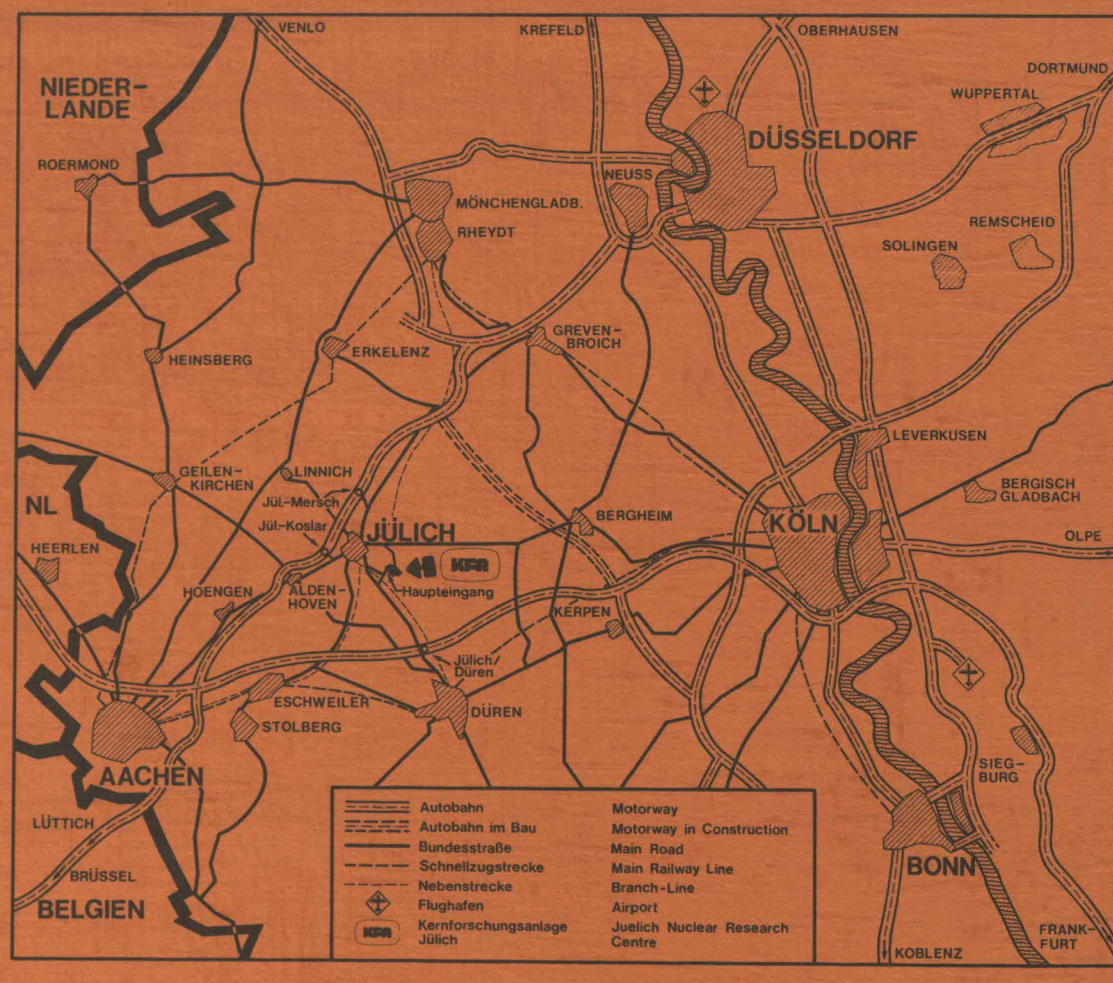
Zentralinstitut für Angewandte Mathematik

TSS VSAPL System Guide

by

W. R. Schwan

Jül - Spez - 61
November 1979
ISSN 0343 7639



Als Manuskript gedruckt

Spezielle Berichte der Kernforschungsanlage Jülich – Nr. 61

Zentralinstitut für Angewandte Mathematik Jül - Spez - 61

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH

Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)

Telefon: 02461/611 · Telex: 833556 kfa d

TSS VSAPL System Guide

by

W. R. Schwan

TSS VSAPL System Guide

PREFACE

This documentation gives technical information on the adaption of VSAPL to the TSS environment. VSAPL has been implemented to run as an interactive subsystem under TSS and serves as a tool for interactive problem solving in the scientific as well as in the administrative area.

The TSS adaption provides a complete implementation of VSAPL, including also the full concept of shared variables. While preserving compatibility, it was our aim to improve the performance of VSAPL by utilizing the advantages of TSS as much as possible, especially concerning the transfer of big amounts of data between external storage and virtual memory.

The standard auxiliary processors for sequential I/O, for issuing commands of the underlying operating system, and for stacking VSAPL input have also been converted. Besides that, means are provided to enable the users to write their own auxiliary processors. VSAPL workspaces created under VSPC, CMS, or TSO can easily be converted to TSS format, and vice versa. Utilities to convert workspaces from APL/360 or APLSV have not been implemented.

Support is provided for most terminals with APL character set. Insertion of character translation tables for new terminals is easy.

Design considerations for this work began in 1976 when the VSAPL system emerged to be an efficient and modular APL system with a clean interface to all IBM operating systems except TSS. These considerations were complemented by discussions with IBM TSS Marketing and Support Group and with Yale University whose TSO adaption seemed to be a model for the work to be done. The major part of the implementation has been completed in 1977, while special supplements concerning shared variables and performance improvements have been added in 1978 and 1979. Further maintenance will be done in close cooperation with the TSS group.

The author would like to thank F. Hossfeld and H.W. Homrighausen for continuous support of this work, D. Bartel, D. Erwin, and K. Wolkersdorfer for useful discussions and suggestions, and Mrs. K. Baumann for assistance in generation and documentation of the system. Thanks are also due to IBM Corporation, especially to E. Mohr who assisted me in utilizing the TSS peculiarities, and to Yale University Computer Center for the readiness in providing us with information on the TSO adaption. We also appreciate the help of W. Hahne, DKFZ Heidelberg, concerning the support of the I/O auxiliary processor and of the 3270 data analysis feature.

1. General Description

1.1. The Modular Design of VSAPL

Although one of the main ideas in the design of the APL language has been operating system independence /1/, the early implementations of APL suffered from dependencies in two aspects: Different ways of language extension and of operating system interfacing have been followed in different implementations.

Since APL was an unstandardized and experimental language in the 1960s, a series of language extensions were implemented by IBM, as well as by customers or other manufactorers. Unfortunately, these dialects happened to be developed under different system environments and led so to incompatibilities which were not necessarily caused by different properties of the underlying operating system.

The fact that early APL implementations had poor communication facilities with the operating system and other users, reflected the original design of APL as a notational language, but the ability of the language to solve data processing problems soon demanded better system communications, especially concerning the access to data stored outside of the APL workspace. Here of course the development of experimental systems for communication with the operating system diverged, and in addition to the danger of diverse dialects, the language began to be interspersed with system interface commands.

To overcome these difficulties, two major steps in the development of APL systems have been done:

The first step was the concept of shared variables developed by Lathwell /2/. APL variables can be declared to be shared between different processors: between an APL user and a concurrently running program performing system services (e.g. data transfer), or between two concurrently active APL users. This concept allowed to transfer system dependent activities into a processor outside of APL, thus having again the language purged from system peculiarities by means of a clean interface to the underlying operating system. Besides that, special system service requests, as time stamps, utilization of system resources etc., may be fulfilled consistently by variables shared with the operating system. APLSV /3/ has been the first system in which a shared variable processor has been implemented. A side effect of the good acceptance of APLSV and of compatible implementations on other machines was that the development of the language itself came to a consolidation, after the integration of important new functions, as "execute", "format", matrix inversion, etc.

The second step has been the design of VSAPL /4/, a modular implementation for all main-line operating systems on the IBM/370 hardware, including compatibility to the 5110 intelligent terminal. Here the language is essentially based on the extent of APLSV; the shared variable concept is realized in a very rigid manner, where also the I-beam system functions have consequently

been substituted by shared system variables. So both language and system have now reached a stage suitable for standardization /5/.

The most important progress of VSAPL, however, is the modularity of the implementation which allows to have common code for versions belonging to different operating systems, while the system dependent parts are kept as small as possible. Even the environment interface (standard auxiliary processors, conversation aids etc.) is functionally equivalent in the VSPC, CMS, and TSO versions, and so a good portability of application programs is guaranteed.

The promising prefix VS (virtual storage) leads to a certain disappointment. It only means that APL is implemented under virtual storage systems; unfortunately it does not mean that the concept of virtual storage is utilized extensively (e.g. by the use of virtual I/O for the copy process).

Nevertheless, concerning the adaption of an APL system under TSS to be used in the KFA computer system and application environment /6/, VSAPL proved to be a good choice for a base, because of the reasons mentioned above. To understand the concepts of the implementation, the components of VSAPL will be discussed in more detail /4/.

1.1.1. Translator and Interpreter

The translator is used to prepare APL source (system commands, function editing, statements) for execution. Its single functions are:

* to initialize the user's workspace
* to analyze terminal input
* to prepare APL statements for interpretation by converting them into internal code (tokens)
* to execute system commands
* to control the function editing

The interpreter is responsible for APL statement execution. It receives control from the translator to execute a tokenized APL statement. Besides the central task of interpreting APL expressions, it has to control function calling and function processing, and it communicates with the shared variable processor when a variable is recognized to be shared during processing.

Translator and interpreter cooperate closely and build the system independent nucleus which performs the basic operations on APL objects and controls the status of the workspace. In the following, we will not distinguish between both components, and we will uniformly call them "interpreter".

1.1.2. Executor and Shared Variable Processor

The executor establishes communication between VSAPL and the operating system. Its code is extremely system dependent and exists in different versions for the VSPC, CMS, and TSO time sharing environments. Nevertheless, all versions perform the same type of functions which can be summarized as follows:

> The initialization routine organizes global tables, scans for information, and loads information on the auxiliary processors used during the session; it allocates space for shared memory, work areas, and the workspaces, builds an incore library table, determines the terminal type, and sets program interrupt exits.
>
> Asynchronous exit routines handle program interrupts, abnormal end situations, table overflows, and terminal attentions (as far as they are not used for input line editing), for the whole APL system. Moreover they support the dump and system diagnostic service requests.
>
> Typical service requests are: send output to terminal, get input from terminal, load or save a workspace, print information concerning workspace or library, as well as requests for supporting the)COPY command, get accounting or timer information, etc. Service requests are normally issued by the interpreter and analyzed by a special routine of the executor which routine in turn calls other executor routines to perform the elementary system services.

The shared variable processor is responsible for the actions resulting from the occurrences of shared variables in the interpreter, as well as for the organization of shared storage. Its first task includes initialization and termination of the shared variable facility and support for shared variable commands (offering, retraction, assignment, reference). This may be regarded logically part of the executor and has no substantial system dependence. It is isolated from the executor code only because in VSPC the APL technique of shared variables is generalized to be used by non-APL processors and thus part of the system. Therefore the shared variable processor is designed as a separate module in systems different from VSPC.

1.2. The TSS Implementation

1.2.1. General Design

The basic design objective of the TSS adaption has been to utilize the properties of TSS while retaining full functional compatibility to other operating systems. TSS turned out to be a very good host system to support an efficient VSAPL implementation. The most advantageous features of the general system structure of TSS are listed below /7/ :

* Although program loading runs very efficiently, the process of loading the APL system may even be fastened by putting the sharable part of the code into the initial virtual memory (IVM). This is a storage area - comparable to the link pack area in OS - which is common to all users, and in which linked modules may be placed and accessed from all active addressing spaces. So the the system loading overhead may be avoided for the individual user.

* The concept of virtual memory allows to use large workspaces. Moreover the concept of named segments enables to disconnect parts of virtual storage temporarily and reuse the address space, thus logically extending the virtual memory beyond its limitations of 16 megabytes, which gives an efficient implementation for the copy process (see below).

* The Virtual Access Method (VAM) of TSS extends the page structure of virtual memory even to external data sets. So transferring data items to or from VAM-organized external data set does not substantially differ in function and efficiency from data transfer to or from a paging device. This is heavily utilized in the organisation of the workspace management (cf.1.2.2.2.).

* TSS admits sharing of data areas among different address spaces. This is accomplished by means of shared control segments which usually share program code for simultaneous use by different tasks, thus allowing a full implementation of the Shared-Variable Concept esp. between different APL-users (tasks).

* The program product language interface (PPLI) is a TSS tool to transform OS object modules into TSS structure. It is used to automatically transform the system independent parts of VSAPL into TSS object modules.

* The address space structure gives security for the active workspace.

* The ownership concept for data sets enables the user to construct complex sharing structures and provides security against accessing or changing data without the owner's explicit permission.

For the adaption of the system dependent parts, the CMS code and the TSO code have been inspected for our work. We started the adaption project by checking the CMS code of the executor because it seemed to be closer to our aims than the code for the VSPC environment. When the adaption to TSO accomplished by the Yale University /8/ had become available, it was chosen to be the model for the TSS adaption. At least the different existent versions demonstrated the locations of the system dependent parts of the code. In general, the TSS code has become even simpler because TSS gives a good support for example in virtual memory management and data set allocation.

We tried to structure the code by modular programming and by the extensive use of macro instructions, and we hope this will help to understand and to modify the code, if necessary. So especially the translation tables for the different terminal types, which are supposed to be modified most frequently and also by unexperienced programmers, are constructed by macros which allow easy changes of a given pattern.

The most important design objective for the programming style has been to displace the divide between system dependent and independent part of the code in a manner that tries to minimize the dependencies. So the whole executor has been rearranged to reflect the overall logic independent of the operating system, hiding the system dependent parts behind macros and calls to a subroutine system which contains the code really specific to the TSS system, whereas the nucleus of the executor is system independent on the macro instruction level. The system service routines are grouped into library management, virtual storage management, program interrupt handling, terminal I/O, timer and accounting, error handling, copy command processing, and priviliged system support. Only the last routine runs in priviliged state, all the other ones are unprivileged and should not be touched by TSS release changes.

1.2.2. The Executor

Some of the components of the executor, e.g. account and timer information, have been adapted in a straightforward manner. Others, as interrupt handling, have been rebuilt with a very different system support, but in equivalent logic. The cases in which the TSS system brought substantial advantages for the implementation are described below.

1.2.2.1. Organization of Libraries

In APL libraries are collections of stored workspaces. They are identified by library number, and may be either private or public libraries. Each user has a unique private library which is associated with the user's account number. Other users may have access to a workspace of this library on a 'read only' base by means of knowing this account number, the workspace name and eventually a password protecting this workspace. Public libraries are not assigned to individual users. They may be loaded and copied by everyone but may only be changed by the user who first saved them. In VSAPL 'project libraries' may be created

additionally; they are shared among a group of users.

The TSS implementation creates the partitioned data set (PDS) VSAPL.LIBnnnnn for the library nnnnn, the members of the PDS being the workspaces of the library. The access mechanism is extended to the full TSS concept of sharing a data set, where each user may own more than one library and may permit them to other users on a defined level. As a special case, an individual user may establish the same organization of libraries as the different VSAPL versions define, without being disturbed by the environment.

1.2.2.2. Virtual Memory Management

When loading or saving a workspace, the TSS virtual access methods provide means for very efficient operations. Since accessing a sequential data set only causes the data set to be virtually mapped into the user's address space, no actual data transfer takes place when a)LOAD command is executed, and only the subsequent referencing of data starts the physical transfer of the data. In the same way unchanged pages remain untouched during the execution of the)SAVE command to the same workspace from where it was loaded. So the performance of workspace operations only depends on the data references and changes, not on the size of the workspace.

The APL logic of copying data items from a stored workspace (the source) to the active one is as follows: the active workspace is stored on a temporary data set, then the source is made the active workspace and the items are written into a buffer by the interpreter. Now the original workspace is reloaded, and the interpreter finally copies the items from the buffer into this workspace. This technique is necessary because the copy process has to be performed under the control of the respective workspace contents to get the correct results. But it is very inefficient to use data sets for the temporarily unloaded workspace and for the buffer since in contrast to the load technique mentioned above, the whole data set would have to be moved. Placing the data sets in the virtual storage would have restricted the workspace size. So the concept of named segments has been used to temporarily enlarge the virtual storage. At the beginning the active workspace is disconnected; it is reconnected after the items from the source have been moved to the buffer. A constant segment of virtual memory is assigned for the buffer; if more storage is used, the buffer is enlarged by disconnecting this segment. So large workspaces may be copied without using auxiliary data sets at all, and the work is reduced to only those data transfers in virtual memory which are logically necessary. This also combines a good performance with the possibility of utilizing the whole available address space for the workspace.

This technique has an important advantage for the design of data base applications: It is possible to arrange a data base up to 10 MBytes with an APL workspace. Following a data pass of a query operation now means only to load a set of pages which contain the relevant information. No actual transfer of the whole data set takes place, and therefore this technique can be utilized for a non-sequential access to a data base with a good performance. The designer of the data base may restrict himself on the programming of the logical operations; he may leave the real data transfer to the paging mechanism of the operating system /9/.

1.2.3. Auxiliary Processors

In TSS, auxiliary processors are implemented as independent programs running in the user's address space. They are invoked by the shared variable processor and communicate with the user's APL system only by means of shared variables. The use of shared control sections is not necessary for the implementation of auxiliary processors, since shared data can be accessed in the same address space.

Conversion of auxiliary processors from other operating systems to TSS or writing new processors is easy: the modified set of macro instructions may be used for this essentially as described in /10/. Differences in coding result from the fact that TSS divides program code into common read-only CSECTs and individual data areas called PSECTs.

Standard processors in the TSS VSAPL system are those which are also offered by the IBM distribution package for the CMS version. They are more or less formally converted to TSO and TSS, having the same meaning under different operating systems and thus guaranteeing software portability for this class of auxiliary processors. Their functions include command processing, input stacking, and data transfer.

* The command processor (AP100) shares a character vector with APL whic is, after setting, transmitted to TSS to be executed as a command. So TSS commands can be issued without leaving the APL environment.

* The stack input processor (AP101) is used to create a stack of input data by assigning these data line by line to a character vector. When APL is next ready to accept input, this input is taken from the stack. The stack may be used either as LIFO or as FIFO stack.

* The sequential I/O processor (AP111) has the function of sequentially reading and writing data sets. This processor allows to exchange data between APL format and external format usable by other programming languages. The user may specify the type of conversion by which data are transferred to or from external storage. Three types of conversion are provided: character translation from APL to EBCDIC (with special options for translating special characters and expanding compound characters), internal representation (including descriptor information, without conversion), or

data part of internal representation (without descriptor information).

* The nonsequential I/O processor will support direct access and indexed sequential data transfer.

The nonsequential I/O processor is still under development, whereas the other standard processors may be used either directly via shared variables or by means of a package of APL functions.

Nonstandard processors have been developed for data set definition and for account information; a processor for the efficient direct access transfer of APL shaped external data is in preparation. DKFZ Heidelberg is about to provide an auxiliary processor which allows to dynamically invoke FORTRAN and PL/I programs.

2. VSAPL Components

TSS VSAPL consists of 5 logical sets of modules that can be arranged in 3 levels.

First level: The Interpreter, a black box, containing the whole APL language definition. The TSS Interpreter is the same as in all other VSAPL implementations.

Second level: The Executor, the Shared Variable Manager and Auxiliary Processors, known as the system depending part of VSAPL.

Third level: TSS System Support Routines, the interface between VSAPL and TSS.

Control is passed from one level to another through defined interfaces. The Interpreter always calls the Executor module APLSCFXI, if service from the Executor is required. APLSCFXI examines PTHYYCOD in the Perterm Header (Pointer GLBLTBL in Module APLVMCTL points to the Perterm Header, that is also known as Global Table) to determine the kind of service requested. Two words in the workspace (WSMPARM1 and WSMPARM2) are reserved to pass parameters from the Interpreter to the Executor and vice versa. The Executor returns control to the Interpreter calling module APLITINI. Each Executor module is supposed to store its returncode into PTHSRCOD in the Perterm Header. The defined interface between the Executor and auxiliary processors is the shared variable manager. Control is passed to the shared variable manager through a set of macro instructions described in 'VSAPL for CMS: Writing Auxiliary Processors'. Only Executor module APLSCSVI is supposed to call the shared variable manager, so changes concerning the communication between Executor and shared variable manager should be done in this module.

The third level has been implemented to have the Executor TSS installation independent. Most of the system service routines are small and could be easily changed, if there are differences between one TSS installation and another. The Executor and some auxiliary processors, too, use to call system service routines by macro.

Naming conventions:

Interpreter: see manual 'VSAPL PROGRAM LOGIC'

Executor: all csectnames start with APLSC

Shared variable manager: all csectnames start with ASUSH

Auxiliary processors: AP###, where ### is the processor id for the auxiliary processor

System Service Routines: all csectnames start with APL and a 2 character id determining the type of service.
(ER - error handling
IB - library management
IO - terminal IO
PI - program interruption handling
VM - virtual memory allocation)

To get the module name replace the first A in the csectname by @.

2.1. The Interpreter

2.1.1. Installation of the Interpreter

To provide compatibility with other VSAPL installations the Interpreter is adapted as-is from the VSAPL Basic Material Tape for CMS and VSPC. The object deck converting facility (ODC) of the Program Product Language Interface (PPLI) is used to make the Interpreter object modules accessable in TSS, because the format of TSS object modules is different from that in all other IBM operating systems. ODC generates a seperate module for each control section in the input file. To avoid duplicate module names the ODC has been modified. The modified version of the ODC is distributed together with TSS VSAPL (library VSAPL.SYSLIB, member CESHR). The TSS VSAPL installation script can be used to install the current Interpreter release (see '3.2 Installing a New Interpreter Release' and description of VSAPLGEN procedure)

2.1.2. Correction of Interpreter Bugs

The only way to modify object modules under TSS is to use the VPAT command. For instance, if you get a PTF for the Interpreter saying that location X'4FØ' in csect APLITINI, currently containing X'47FØC5D8', must be changed to X'478ØC5D8', you should act in the following manner:

1. Look at the location in question and verify its contents

```
VDSP VSAPL.SYSLIB,APLITINI,OBJ,X'4FØ',4
```

2. Modify the location

```
VPAT VSAPL.SYSLIB,APLITINI,OBJ,X'4FØ',4,X'478ØC5D8'
```

3. Generate a new link edited version of VSAPL

```
VSAPLGEN LINKEDIT
```

The best way to keep track of all changes applied to Interpreter object modules is to store all VPAT commands into a dataset, so that you can re-call all modifications, if you are forced to switch to a backup version of VSAPL.

2.2. The Executor

The Executor is that part of VSAPL, that executes the VSAPL system commands and controls the communication with the user. The Executor has been rewritten for TSS for two reasons:

* Many things done in the Executor modules for CMS and TSO are already handled within the TSS system itself, so that great amounts of code could be removed.

* Most of the Executor modules are quite big, so that it is better to have them independent of the version of TSS they are running with. The system dependent parts of the Executor have been moved to small modules referred to as System Service Routines, which can be modified much easier.

2.2.1. Installation of the Executor

The source code for all Executor modules is stored as a regionized VISAM member (SOURCES) in library VSAPL.ASMLIB. Assembly is done in the VSAPL installation script VSAPL.SYSGEN using procedure VSAPLASM.

2.2.2. Executor Components

The Executor is divided into 3 main parts:

* the terminal management, including all modules that communicate with the VSAPL user

* the library management, handling)LOAD,)SAVE,)LIB,)DROP,)COPY and)PCOPY system commands

* miscellaneous service requests as)CLEAR,)QUOTA, QUAD-AI, etc.

2.2.2.1. Terminal Management

Modules: APLSCTYP, APLSCDPY, APLSCTBL

VSAPL distinguishes between two types of terminals:

* 3270 displays with APL keyboard
* 2741 - like typewriters

Input and output from and to displays is done by module APLSCDPY, all other terminal types are handled by module APLSCTYP. The system service routines are supposed to know about special cases as teletypes, etc., although all those terminals are treated as 2741s by the Executor, however.

2.2.2.1.1. Character Translation

APL stores characters in a special code, called Z-Code. Upon output Z-Code characters must be translated to EBCDIC, upon input vice versa. This is done in APLSCDPY (for displays) or APLSCTYP (for all other terminals). Special characters like backspace, startfield (for 3270) etc. are also handled in these modules. APLSCDPY and APLSCTYP do not support translations depending on special terminal types or TSS input/output facilities. All system or terminal dependent translations are done by the system service routines APLIOGET, APIOPUT, APIOSEL and APLIODEF (see 2.5.1.2.).

2.2.2.1.2. The Input Stack

Although TSS supports stacking of input data since multiple sysins and sysouts are available, VSAPL has its own input stack. VSAPL does not use the multiple sysin facility, because using datasets to support a stack is less efficient than keeping it in virtual memory. The VSAPL input stack occupies an area of virtual memory that is also used as buffer during)COPY and)PCOPY. Therefore it is held as a disconnected segment group while the last mentioned system commands are active. Input data in the stack is stored in Z-code format to avoid unnecessary character translations. No translation is done upon reading from the input stack. APLSCTYP and APLSCDPY recognize input from the stack by testing bit 0 in byte 9 of the parameterlist for APLIOGET (see 2.5.1.2). The auxiliary processor APL101 must be used to store data into the input stack.

2.2.2.2. Library Management

Module: APLSCLIB

2.2.2.2.1 Format of Libraries and Workspaces

VSAPL libraries are implemented as TSS VPAM datasets with the standard dataset name VSAPL.LIB#####, where ##### represents the VSAPL library number, which must be specified in all library requests (the default library number is 0). Workspaces are saved as members of a library. Workspace names must not exceed 8 characters, because member names are restricted to a maximum length of 8. If the user specifies a workspace name longer than 8 characters, the request will be rejected by the Executor. The format of a workspace member is similar to the format of a TSS load module. So the TSS program control system (PCS) can be used to display or modify locations inside the workspace even outside VSAPL. Also assembler programs may use VSAPL workspaces specifying the workspace name inside a V-type address constant.

A TSS load module consists of 3 parts:

PMD	(program module dictionary)
TXT	(data)
ISD	(internal symbol dictionary)

The ISD is omitted for VSAPL workspaces. The TXT part contains the workspace data starting at location WSMFREEA up to the first unused page of the workspace. WSMFREEA (the first 4 bytes of the saved workspace data) contains the length of the saved workspace in bytes. The PMD contains the name of the workspace, the version id (data and time when saved) and some additional data identifying the load module as a VSAPL workspace.

2.2.2.2.2. The)LOAD Command

Entry point SCLOAD in module APLSCLIB

SCLOAD loads a workspace from a library into the user's virtual memory. First, SCLOAD locates the library in the user's catalog and opens the member the user wants to load. Then the PMD is read to retrieve information about the workspace (length, version id, etc.). Checks are made to ensure that the member is really a VSAPL workspace. If SCLOAD was successful until this point, it will compute the user's new current workspace size according to following rules:

* if the user specifies a workspace size in the)LOAD command, take this as the new current workspace size
* else take the old current workspace size, if it is greater than or equal to the size of the workspace going to be loaded
* else take the size of the workspace going to be loaded

SCLOAD then checks the new current workspace size:

* it must be not greater then the maximum allowed workspace size and not less than the size of the workspace going to be loaded.

Otherwise the)LOAD command will be rejected.

Last not least SCLOAD loads the workspace using the @PLLOAD macro instruction.

2.2.2.2.3. The)SAVE command

Entry point SCSAVE in module APLSCLIB

SCSAVE saves a workspace, if one of the following conditions is true:

* The workspace name specified in the)SAVE command is not equal to the current workspace name and it does not exist in the output library.
* The workspace name specified in the)SAVE command is equal to the current workspace name,
* The workspace name is CONTINUE.

If none of the conditions above is true, the)SAVE command will be rejected. If the user exceeds his permanent storage ration while saving a workspace, a warning will be printed at his terminal, but the the workspace will be saved in spite of that. However, if the user's permanent storage ration is already exceeded before)SAVE starts, the request will be rejected. If a user gets the warning message mentioned above he should erase or migrate some of his datasets prior to)OFF or LOGOFF, because he might no longer be able to LOGON afterwards. (Note: LOGON with PRISTINE=X will always work in this case, but most users do not know about that.)

SCSAVE uses the @PLSAVE macro instruction to store the workspace into the library. All system dependent things as building the PMD and moving the data to the library are done inside system service routine APLIBSAV.

2.2.2.2.4. The)COPY and)PCOPY Commands

Entry points SCOPA,SCCOPO,SCCOPI,SCCOPZ in module APLSCLIB

The COPY process consists of 5 steps:

1. COPY initialization (SCCOPA)
2. COPY output (SCCOPO)
3. Switch from COPY output to COPY input (SCCOPZ)
4. COPY input (SCCOPI)
5. COPY termination (SCCOPZ)

No difference is made between)COPY and)PCOPY. Things in)PCOPY that are different from)COPY are handled inside the Interpreter.

Step 1: COPY initialization

Since the workspace specified in the)COPY or)PCOPY command must be loaded during the COPY process, the user's current workspace must be saved. SCCOPA simply disconnects the current workspace, so that the same virtual memory locations can be used to load the COPY workspace (for a detailed description of the TSS disconnected segment support see 'IBM Time Sharing System: Assembler User Macro Instructions, GC28-2004-6', DISCSEG,CONSEG,DELSEG,RSVSEG and RELSEG macro instructions). Then SCCOPA loads the workspace specified in the)COPY or)PCOPY command and returns to the Interpreter.

Step 2: COPY output

The Interpreter splits the COPY workspace and passes the items to SCCOPO. SCCOPO is supposed to save those items for later use. Items are saved in an area of virtual memory referred to as COPY buffer. When the COPY buffer is full, it will become a disconnected segment group and new virtual memory will be allocated to save the further amount of copy data.

Step 3: Switch from COPY output to COPY input

If all relevant data of the COPY workspace has been stored into the COPY buffer, the Interpreter calls SCCOPZ to switch from COPY output to COPY input mode. The following must be done by SCCOPZ before the Interpreter can read the previously stored copy data:

* Restore the user's current workspace by re-connecting it to the same area of virtual memory it has occupied before)COPY starts.

* Disconnect the last copy buffer and re-connect the first one.

Step 4: COPY input

SCCOPI is called to read the COPY data saved during COPY output. SCCOPY reads from the current copy buffer until it is empty, then reconnects the next buffer, if one exists. The data is returned to the Interpreter in WSMBUFF (a 1K buffer at the beginning of the workspace).

Step 5: COPY termination

SCCOPZ is called again to terminate the COPY process. Nothing but setting up some bits is done by SCCOPZ upon COPY termination.

Errors during)COPY or)PCOPY:

Any error condition during COPY process causes the user's current workspace to be restored. If that is not possible, the user's task will be abnormally terminated.

2.2.2.2.5. The)LIB Command

Entry point SCLIB in module APLSCLIB

SCLIB returns all workspace names associated with a given library number. Workspace names will be returned in WSMBUFF in alphabetical order. If WSMBUFF is full, the Interpreter will be supposed to call SCLIB as long as no more workspace names exists. SCLIB retrieves the workspace names by reading the directory (POD) of the VPAM dataset that represents the library. This implies a lock to be set on the RESTBL in case of a shared library.

2.2.2.2.6. The)DROP Command

Entry point SCDROP in module APLSCLIB

SCDROP is called to remove a workspace from a library. The member containing the workspace is deleted by @PLIBDEL macro instruction.

2.2.2.2.7. Export/Import of Workspaces

TSS VSAPL is able to load any workspace that has VSAPL format. Also TSS VSAPL can be used to produce workspaces that can be exported to other VSAPL installations. All export/import workspaces are associated with a pseudo library number 100000. The specification of library number 100000 in the)LOAD,)SAVE,)COPY or)PCOPY command has following effect:

* for)LOAD,)COPY,)PCOPY:

  The workspace will be loaded from a sequential file on tape or disk. The dataset name of that file must be equal to the workspace name and must be in the user's catalog or previously defined by a DDEF command.

* for)SAVE:

  The workspace will be saved on a sequential file on tape or disk. The datasetname of that file will be equal to the workspace name. A DDEF command specifying that name must be issued prior to)SAVE, if saving on a tape file is requested, otherwise the system assumes a disk file.

All other commands issued with library number 100000 will cause an error message be printed on the user's terminal.

All sequential files used with the export/import facility should have RECFM=F or FB and LRECL=80. The export/import facility is not allowed to users joined with authority=u.

Example: You want to load a CMS VSAPL workspace TYPEDRIL from tape and save it in your library 0.

```
DDEF ANYDD,PS,TYPEDRIL,DISP=OLD,UNIT=(TA,9D3),-
     LABEL=(10,NL),VOLUME=(,CMSTAP),-
     DCB=(DEN=3,RECFM=FB,LRECL=80,BLKSIZE=800)
VSAPL
)LOAD 100000 TYPEDRIL
)SAVE TYPEDRIL
)OFF
```

2.2.2.3. Miscellaneous Service Requests

Modules: APLSCMSC, APLSCMSG and APLSCERR

2.2.2.3.1. Message Handling

Module: APLSCMSG

APLSCMSG supports the)MSG command.)MSG ON and)MSG OFF cause the message receive bit in the user's TSI (task status index) to be set 1 or Ø. If the user has issued the)MSG command to send a message to another user, the following is done:

* send the message to the user specified in the)MSG command, if he has an active task
* wait until the receiver of the message sends a response or the sender hits attention.

2.2.2.3.2. Quad-DL Support

Entry point SCDELAY in module APLSCMSC

The Quad-DL system variable enables the VSAPL user to put his task into wait state. If the user specifies Quad-DL, routine SCDELAY in module APLSCMSC will be called. SCDELAY uses system service routine APLMWAIT which issues a STIMER wait macro instruction to enter the wait state.

2.2.2.3.3. The)CLEAR Command

Entry point SCCLEAR in module APLSCMSC

The)CLEAR command is used to erase the user's current workspace and to change the workspace size. Since allocation of virtual memory is always done for the maximum workspace size,)CLEAR does not affect the amount of core reserved for the user's APL session. Only a few control fields in the perterm header and the global table are changed by)CLEAR.

2.2.2.3.4. The)QUOTA Command

Entry point SCQUOTA in module APLSCMSC

The)QUOTA command displays the following data upon the user's terminal:

* used external storage (that means storage on disks)
* unused external storage (" " " " ")
* default workspace size (workspace size used when the user starts his APL session)
* maximum workspace size
* maximum number of shared variables
* size of memory available to store shared variables

Different from CMS VSAPL the default workspace size is not equal to the maximum workspace size under TSS, which can be set by the user using the MAXSIZE parameter of the VSAPL command (system default: 8MB).

2.2.2.3.5. Quad-AI Support

The system variable Quad-AI contains the user's account number, accumulated CPU-time, terminal connect time and keying time. The following is important to know about:

* the user's account number, which is also used as processor id in shared variable requests, is not a constant (1001 in other VSAPL installations), but the TASKID multiplied by 4096.

* CPU-time is accumulated for the user's task, not for his APL session only. The user can therefore use Quad-AI to check the CPU-time already accounted since LOGON against the maximum allowed for his task, assumed, that the TSS version he's working with has implemented a task CPU-time limit.

2.3. The Shared Variable Manager

The Shared Variable Manager handles all requests using shared variables. The CMS version of the Shared Variable Manager has been slightly modified due to the fact that TSS VSAPL supports the full concept of shared variables including sharing variables among APL users. The best way to support sharing virtual memory between TSS tasks is to use a public csect as it is done by TSS VSAPL. This implies that all addresses the Shared Variable Manager used to store into shared virtual memory had to be changed to offsets, because it is not guaranteed that public csects are loaded at the same virtual memory address in each task they are referenced by.

2.3.1. Installation of the Shared Variable Manager

Under TSS VSAPL the Shared Variable Manager is considered as a part of the Executor. Its source modules are stored in member SOURCES of library VSAPL.ASMLIB together with the Executor source modules. The Shared Variable Processor will be automatically installed, when the VSAPL system generation script VSAPL.SYSGEN is executed (see 3. 'System Generation').

2.3.2. Shared Storage under TSS

Since there is the possibility to share virtual memory among serveral TSS tasks TSS VSAPL supports sharing of variables between APL users. Shared storage is automatically allocated by the dynamic loader when loading public csects. If the user wants to share variables with other users, a public csect named APLSVMEM will be loaded. The first task referencing APLSVMEM builds control blocks and buffers for shared variables (see also 2.5.1.1. 'Virtual Memory Management'). Since the Shared Variable Manager already simulates a multitask environment only changes mentioned above has been done for multitask shared variable support. Some changes inside Executor modules APLSCSVI and APLSCSHV are important to know about concerning shared variable support

* APLSCSVI formerly expected the control block (TSKBLOCK) associated with the user's main task to be on top of the control block chain; it is now identified by the user's task id. Also unique processor ids are generated for the auxiliary processors (see APLSMPID routine in chapter 2.5.1.1.). The user will not know about generated processor ids but specify the original id (100 for APL100, etc.) as usually.

* APLSCSHV did not handle deadlock situation correctly. It has been modified to perform an AWAIT to keep the user waiting. Deadlock situations might occur during access to shared variables controlled by the access control vector (Quad-SVC). The task waiting for a shared variable event will not leave the wait state until the task its waiting for satisfies the shared variable request or the user signals double attention. The TASKIDs of all VSAPL tasks beeing in wait state are kept in a queue starting at the beginning of the public csect APLSVMEM. Tasks, that are not waiting, are supposed to send a message if they have satisfied the shared variable request one of the tasks in the wait queue is waiting for. A VSEND SVC with a message code of 127 is used for that purpose. Access to shared virtual memory must be synchronized. A lock word is reserved at APLSVMEM.(X'00') to do that. The CS instruction (compare and swap) is used to access the lock word. If a user holds the shared virtual memory lock, the lock word must contain his taskid. Shared variable support under TSS VSAPL is a full implementation of the APL shared variable concept as described in publication 'APL Language' (GC26-3847).

2.4. Auxiliary Processors

Auxiliary processors are independent programs used as a gate from APL to the host system or other languages. APL and auxiliary processors talk to each other via shared variables. A set of macro instructions is available to simplify access to shared variables (see 'VSAPL for CMS: Writing Auxiliary Processors').

2.4.1. Installation of Auxiliary Processors

The auxiliary processors distributed together with TSS VSAPL are automatically installed by the installation script. New auxiliary processor should be written according to following conventions:

* Each auxiliary processor should have a PSECT and a standard TSS savearea. That makes using of TSS macro instructions much easier.

* Move all data areas from auxiliary processor's CSECT to its PSECT, because for auxiliary processors used by several user's it's better to have public readonly CSECTs.

* If you're going to adapt an auxiliary processor from other VSAPL installation, be sure that no USING defines base registers at locations inside the PSECT or CSECT. If those USINGs are not removed, it may happen that an auxiliary processor cannot be used twice during a user's session without intervening UNLOAD, because of locations being modified that should not be changed.

* Upon first entry to an auxiliary processor register 13 points to its PSECT or to an one page workarea, if it has no PSECT. Do not change the contents of register 13 inside the auxiliary processor! Never forget, that the workarea passed to an auxiliary processor without a PSECT is used for other purposes when the auxiliary processor is not active. The contents of data areas located in the workarea will be unpredictable when the auxiliary processor is scheduled for the next request.

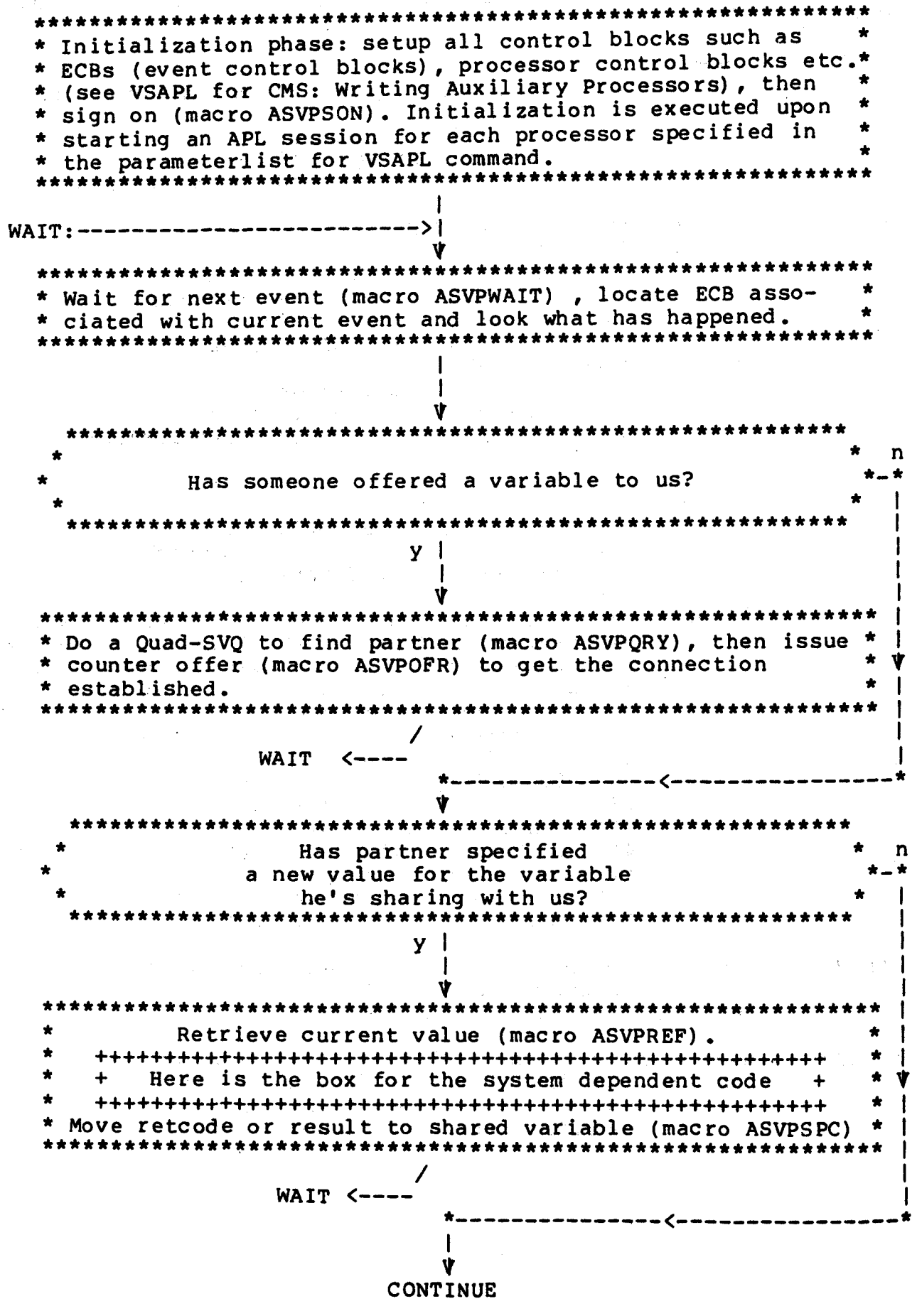
APL100, APL101, APL111, APL200 or APL333 can be used as a frame for new auxiliary processors for TSS VSAPL (see also flowchart below).

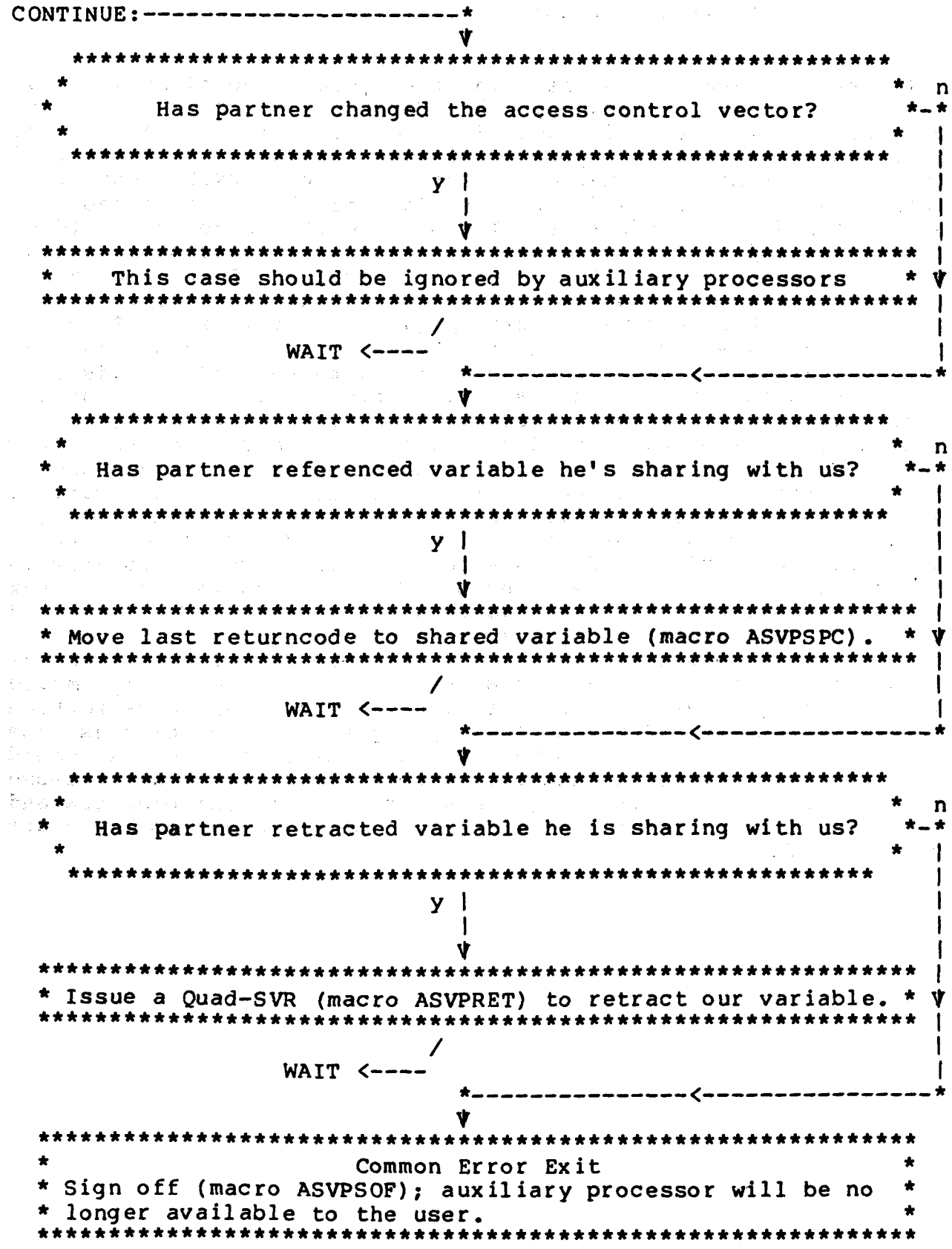
2.4.2. Auxiliary Processors available under TSS

The following auxiliary processors are distributed together with TSS VSAPL:

```
APL100      execute TSS commands
            input:  a string containing one or more TSS commands
            output: a return code
                    0    successful completion
                    1    invalid first specification
                    444  severe error (invalid input variable)

APL101      stack input data for VSAPL
            input:  (1rst specification)
                    a parameterlist as described in
                    VSAPL for CMS: Terminal User's Guide
                    (all following specifications)
                    a string containing valid VSAPL input
            output: a return code
                    0    successful completion
                    1    invalid first specification
                    444  severe error (invalid input variable)

APL111      VAM I/O
            input:  (1rst specification)
                    a parameterlist as described in the
                    VSAPL for CMS: Terminal User's Guide
                    (all following specifications)
                    data
            output: if the user has reference the shared variable,
                    data retrieved from the VAM file or a 0 vector
                    signaling an error if the user has specified
                    a new value for the shared variable:
                    0    successful completion
                    8    incorrect record length on input
                    12   end-of-file reached
                    15   fixed length record wrong size
                    17   record too big for file
                    440  can't open file for output
                    441  can't open file for input
                    443  not enough free space
                    444  wrong size, shape or data type
                    445  not enough shared memory

APL200      execute a DDEF for VSAPL and send back the returncode
            input:  paramterlist for DDEF macro
            output: return code from DDEF macro

APL333      return first element of QUAD-AI for a given userid
            input:  a string containig the userid
            output: account number associated with the userid
                    specified as input (TASKID * 4096)
```

General Auxiliary Processor Flowchart





Note: Upon entering one of the macros mentioned in the flowchart control may be passed to the APL user or to other auxiliary processors before the processor that issues the request will be reactivated. Scheduling and dispatching of auxilary processors and the user's APL session, however, is done by module APLSCSVI.

2.5. System Service Routines

The system service routines do the system dependent work for VSAPL. They are called by the Executor or some auxiliary processors, if there is something to do for TSS. We have to distinguish between two types of service routines: the privileged ones, that cannot be invoked directly but via ENTER supervisor call, and non-privileged routines. The privileged routines are all combined in module APLPRIVI, which must reside in SYSLIB or IVM. Enter code 165 is reserved for VSAPL. Upon execution of an ENTER supervisor call (SVC 121) with register 15 containing 165 control is passed to module CZAPLMR in initial virtual memory (IVM). CZAPLMR assumes, that register Ø points to 2 words containing the R-CON value and the V-CON value (in this order) of the privileged service routine going to be entered. Register 1 can be used to pass parameters to the privileged service routine. Nonprivileged system service routines are invoked by a BALR 14,15 instruction with register 15 containing the address of the entry point. System service routines usually start with a @PLENTRY macro instruction and end with @PLEXIT macro instruction. @PLENTRY generates a call to the standard prologue routine APLENTRY in module APLERMON. The standard prologue routine assumes, that register 13 points to a savearea, which usually is not a standard TSS savearea, because Executor modules use a stack of 18-word saveareas in the VSAPL global table. The standard prologue routine also allocates space in a workarea for the service routine's savearea and modifiable data (description of global table and workarea in chapter 2.5.1.1.). The @PLEXIT macro instruction generates a call to the standard epilogue routine APLEXIT in module APLERMON. The epilogue routine re-loads the calling routine's registers. Since system service routines should not call each other or themselves recursively (that would cause big trouble in the workarea), checks are made in the prologue and epilogue routines that make sure, that prologue and epilogue are excecuted in turns.

2.5.1. Nonprivileged routines

2.5.1.1. Virtual Memory Management

Modules: APLVMA, APLVMCTL

Before we come to details something about linkage conventions. Linkage to routines inside APLVMA is different from that described in chapter 2.5., because some routines are called at locations where we must not change any register except register 15. This implies a special linkage that cannot be done by the standard prologue and epilogue routines. APLVMA contains the following entry points:

1. APLVMALL — initial allocation of virtual memory of VSAPL
2. APLVMDEL — release virtual memory upon VSAPL termination
3. APLCOPYA -
4. APLCOPYI |
5. APLCOPYO > virtual memory management for)COPY and)PCOPY
6. APLCOPZ1 |
7. APLRESTR -
8. APLSWAIT — wait for shared variable event
9. APLRWAIT — remove current task from wait queue
10. APLSTPSH — push vsapl input stack
11. APLSTPOP — pop vsapl input stack
12. APLAPGET — get extra vm for aps (disconnect workspace)
12. APLAPPUT — release extra vm for aps (connect ws)
13. APLSMGET — get shared storage
14. APLSMPUT — release shared storage
15. APLSMPID — generate unique identifications for auxiliary processors

Linkage to entries 1 to 12 is as follows:

```
      .
      .                          load parameter registers 0 and 1
    L 15,=V(entry point)         address of required entry
    BALR 14,15
      .
      .
```

Entries 13 to 15 require a special linkage technique:

```
      .
      .
    L 15,=V(entry point)         get address of required entry
    EX 0,0(15)                   execute 1st prologue instruction
    EX 0,4(15)                   execute 2nd prologue instruction
      .
      .
```

It is not necessary to know about linkage conventions, because macro instructions are supplied for each entry point in APLVMA (@PLVMALL, @PLVMDEL, @PLCOPYA, @PLCOPYI, @PLCOPYØ, @PLCOPZ1, @PLSTACK,@PLSMGET, @PLSMPUT, @PLSMPID). All entry points are placed in APLVMA's PSECT, so that the calling routine does not have to supply the R-CON value in the 19th word of its savearea. Entry points that have no special macro to be called by can be invoked via @PLCALL EP=entry point macro instruction.

The following actions take place when VSAPL starts. APLVMALL is called by APLSCINI to allocate all virtual memory areas VSAPL needs during a session:

* the shared storage
* the workarea including the global table
* the copydata buffer
* the workspace area

First of all APLVMALL allocates the shared storage. The shared storage is a public csect named APLSVMEM. It is loaded by a DLINK SVC. If the user does not want to share variables with other users (SHARE=N in the VSAPL command), a GETMAIN is issued instead of the DLINK. If it is not possible to load APLSVMEM for some reason, a warning will be printed at the user's terminal and APLVMALL will proceed as if SHARE=N has been specified in the VSAPL command (see above and chapter 2.6.).

Before allocation for the user's workspace is done some virtual memory is reserved for the system (normally 2 segments). After that APLVMALL adds the size of the workarea and of the copy buffer to the size specified in the MAXSIZE parameter of the VSAPL command and tries to allocate the whole area. If this fails, a retry is made with MAXSIZE decreased by one segment. Retries are done as long as MAXSIZE is still greater than one segment. The workarea consists of the global table (also referred to as PERTERM header) and some free pages that are used as "PSECT" by system service routines and auxiliary processors that have no own PSECT. Executor and Interpreter modules use the APLDEFN macro instruction to expand DSECTs describing the global table. Space for system service routines is allocated within the workarea by the standard prologue routine. System service routines refer to the workarea using a DSECT named WORKAREA, which is internal to the module. System service routines use a special technique when executing TSS macro instructions:

1. a MF=L expansion of the macro instruction is placed in the module's CSECT

2. before execution of the macro instruction the MF=L expansion is moved to the workarea

3. the macro instruction is executed using the MF=E format

Only macro instructions that do not modify any data area should be used in its standard format within a system service routine. The copy data buffer (normally 2 segments long) is used to save the tokenized copy workspace during)COPY and)PCOPY. The copy buffer will become a disconnected segment group when it is full and a new copy buffer will be allocated. The copy buffer is

maintained by APLCOPYA, APLCOPYI, APLCOPYO and APLCOPZ1 routines in module APLVMA. The copy buffer is also used as VSAPL input stack. This implies, that the stack must be disconnected while a)COPY or)PCOPY command is in process.

Concerning the size of a workspace there are three parameters that are important to know about:

* initial workspace size (keyword WSSIZE in the VSAPL command), which is the size of the workspace when the user starts his APL session

* the size of the currently loaded workspace, referred to as current workspace size

* the maximum workspace size (keyword MAXSIZE in the VSAPL command), which is the maximum allowed workspace size for the current session.

APLVMALL reserves virtual memory always for the maximum size. This is done to prevent the user from some strange problems caused by virtual memory segmentation.

Example:

Provided storage is not reserved for the maximum workspace size but will be allocated at the time the user needs it. Then the following sequence of VSAPL statements will cause the user to be notified, that his virtual memory quota is exceeded:

```
)CLEAR 5000000
  .
  .
  .
OBEY 'TASKNO'          (execute "TASKNO" command using APL100
  .                     within the APL function OBEY)
  .
  .
)LOAD DATABASE 8000000
```

Reserving storage for the maximum size should due to the structure of TSS not decrease performance significantly, because although a lot of virtual memory is assigned for the user's task only pages actually referenced by the user are taken into account by TSS.

The system default for the maximum workspace size is 8 megabyte. The user may specify a larger value in the MAXSIZE parameter of the VSAPL command, but must be very careful doing this. Some free space must be available to the TSS system, particular, if the user wants to execute TSS commands during his APL session or uses the VAM I/O processor (OPEN needs virtual memory to allocate buffers, for instance). The user's task will be abnormally terminated, if there is not enough space available to the system.

All virtual memory and segment groups are released by APLVMDEL upon VSAPL termination.

Entry points APLCOPYA, APLCOPYO, APLCOPYI and APLCOPZ1 in module APLVMALL are called from APLSCLIB during)COPY or)PCOPY process. APLCOPYA saves the user's current workspace as a disconnected segment group and initializes the copy data buffer. APLCOPYO saves the copy items in the copy data buffer and disconnects the buffer, if it is full. APLCOPYI reads the copy items from the copy data buffer and re-connects the next buffer, if the current one is empty. APLCOPZ1 re-connects the user's active workspace and the first copy data buffer, so that APLCOPYI can read from it.

Entry points APLSMGET, APLSMPUT and APLSMPID are important in connection with shared variable processing. APLSMGET is called by ASVPSRVC (interface to shared variable processor in module APLSCINI) and SCSVPINI in module APLSCSVI (shared variable initialization phase) to lock the shared storage, so that no other user can access it until the current user's service request will be finished. If shared storage is not yet initialized when APLSMGET is called, APLSMGET will build some control blocks, that are required by the shared variable processor. Also, APLSMGET saves all registers of the calling routine in a control block (TSKBLOCK) associated with that routine. Each auxiliary processor and the APL user himself own one TSKBLOCK (task block) allocated within shared storage. APLSMPUT resets the lock imposed on shared storage by APLSMGET. APLSMPUT is called by ASVPSSERV in module APLSCSVI upon return from shared variable processor. APLSMPID generates unique processor ids for auxiliary processors. The processor id is a number that identifies an auxiliary processor (APL100 has processor id 100, for instance). Problems will arise, if two users, sharing variables with each other, use the same auxiliary processor. APLSMGET computes an internal processor id, which consists of the user's taskid (bits 4 to 19) and the original processor id (bits 20 to 31). Bits 0 to 3 are intentionally left zero. With this feature implemented each user may have his own set of auxiliary processors, even if he shares variables with another user. Also, this feature implies no modifications within code for auxiliary processors or for the shared variable processor. APLSMGET is called by ASVPSRVC in module APLSCINI.

APLSWAIT and APLRWAIT handle the case that two users sharing variables with each other and one is waiting for the other. This may occur if the access control vector associated with the shared variable in question specifies, that one user has to specify a value for the variable before the other is allowed to reference it, etc. When APLSWAIT is called, it stores the taskid of the task it is running in into a wait queue kept in shared virtual memory (starting at word 3). Then the task is put into wait state by an AWAIT svc. Tasks that are not in wait state are supposed to send a message to the waiting ones in the wait queue, if they share variables among each other. Messages are send before exit from APLSMPUT using a VSEND svc with a message code of 127. APLRWAIT removes a task from the wait queue, if the shared variable request it was waiting for has been satisfied, the user signals double attention or a serious error (such as program check) occurs.

Module APLVMCTL contains the virtual memory management control

block. The control block is generated by the @PLVMCTL macro instruction. If this macro instruction is assembled in other modules than APLVMCTL, it will generate a DSECT and establish a base register, so that the data fields of the VM control block are accessable. The first word of the VM control block contains the base address of the workspace. The next two bytes are used as flags. @PLDFSIZ is an EQU defining the system default for the WSSIZE parameter (initial workspace size = one megabyte).

2.5.1.2. Terminal I/O and Input Stack

Modules: APLIOCLO, APLIOCTL, APLIODEF, APLIOERR, APLIOGET, APLIOSCA, APLIOSEL, APLIOTAB

APLIOCLO is called upon VSAPL termination to close all DCBs open for library requests.

APLIOCTL is the terminal I/O control block. It contains the current character translation table, DCB and control fields for the input stack, the terminal type, a register dump area and the VSAPL error log. Additionally core for three small subroutines is provided in the terminal I/O control block:

* APLABEND - print the error log and terminate user's task; called by VSAPL using the @PLABEND macro instruction (the error error log is described at APLERROR below).

* APLERROR - print the error log
  This routine is not used by VSAPL but can be called directly from TSS using the CALL command (debugging aid). The error log contains the following information (listed from the left to the rigth as printed on the terminal): macro, that caused the error; name of the csect, the error occured in; offset from beginning of csect, the error occured at (hex); returncode from the macro, that caused the error (hex), or X'BAD', if it is an error generated by an Executor module or a system service routine.

* POPSTACK - control stacked input
  POPSTACK is called by APLIOGET to determine, whether input from the stack is allowed or not. There are three input modes in VSAPL: input for quad operator, input for quote-quad operator and input for immediate execution. The currently active input mode is indicated in flagbyte WSMODE inside the workspace. POPSTACK checks bit STIMEX in WSMODE, thus allowing stacked input for immediate execution only. This can be changed easily by putting a small routine into APLIOCTL that modifies the contents of control field STACKCTL in the terminal I/O control block. An auxiliary processor might be written, that invokes this small routine allowing the user to control stacked input during his APL session.

System service routines user do access APLIOCTL using the @PLIOCTL macro instruction, which generates a DSECT for the terminal I/O control block.

APLIODEF sets some default values concerning translation of system messages. To have TSS system messages readable during an APL session defaults for ALPHABET, ALPHAIN and ALPHAOUT are set to 3 forcing the system to perform reverse folding (upper case characters are translated to lower case characters and vice versa) for every input and output request. Installations intending to support terminal types, that are not yet supported by VSAPL, should take care of reverse folding. APLIODEF is also called at VSAPL termination to restore the original defaults.

APLIOERR is the common SYNAD and EODAD exit for VSAPL. All DCBs used by VSAPL should be defined with SYNAD=APLIOERR and EODAD=APLIOERR.

APLIOGET reads from the user's terminal or input stack. Since all Z-code translation is done by APLSCTYP or APLSCDPY only EBCDIC to EBCDIC translation is performed within APLIOGET. Only one translation table, APLIOCTT in the terminal I/O control block, is used for character translation. APLIOGET assumes that APLIOSEL has moved the appropriate translation table to APLIOCTT before APLIOGET is called for the first time. APLIOGET expects a parameter list similar to that used by the CMS macro instruction RDTERM (Parameter lists for input/output routines are reserved in the global table). Bytes 10 to 12 of the parameter list contain the address of the input buffer and bytes 15 and 16 its length. Byte 9 (unused in CMS) is used as a flag. APLIOGET will set bit 0 of this byte to 1, if input originates from the stack. In this case APLSCTYP and APLSCDPY suppress EBCDIC to Z-code translation, because the input stack is supposed to contain Z-code data only.

APLIOPUT is the common terminal output routine. APLIOPUT assumes that all Z-code to EBCDIC translation is already done. APLIOPUT only handles terminal specific EBCDIC to EBCDIC translations using the second half of APLIOCTT in the terminal I/O control block. Also checks are made for special characters such as linefeeds and idles, that may not be available on some terminal types. APLIOPUT expects a parameter list similar to that of a WRTERM macro instruction under CMS. The parameter list is identically to that already described for APLIOGET in this chapter.

APLIOSCA is the adaption of the CMS DMSSCN routine, which is used by some auxiliary processors to scan CMS formatted parameter lists. The format of paramater lists for auxiliary processors adapted from other system has not been changed, so that auxiliary processors can be used under TSS VSAPL as described in the Terminal User's Guides for CMS or TSO. However, some options as the conversion option for the stack processor (APL101) are ignored under TSS VSAPL.

APLIOSEL is a very important module called upon VSAPL initialization. It determines the user's terminal type and moves the appropriate translation table for input and output to APLIOCTT in the terminal I/O control block (APLIOCTL). There are two sources for the terminal type: the TERMTYPE parameter in the

VSAPL command and the interrupt storage area (ISA).

If the TERMTYPE parameter has been specified, it will be used to identify the user's terminal type. Otherwise APLIOSEL checks a 2 byte location inside the ISA, which has been set by TSS at LOGON time. Each terminal supported by TSS VSAPL must be identfied by a 2 byte terminal type id, that must be placed into a table starting at label TABLE within module APLIOSEL. Also, each terminal type must be associated with the address of a translation table inside module APLIOTAB. Translation tables are always 512 bytes long. The first 256 bytes are used for input translation, the rest for output translation. If APLIOSEL manages to identify a valid terminal type, it will move the appropriate translation table from APLIOTAB to APLIOCTT in the terminal I/O control block.

APLIOTAB contains all terminal specific input and output character translation tables. Macro instruction @PLTABLE is used to generate translation tables. Input parameter for @PLTABLE is a diagram representing the translation table, so that it is easy to recognize, how characters are translated. APLIOTAB must not be changed using the ALTER command, because due to the extraordinary structure of @PLTABLE's parameterlist it has no line numbers in columns 73 to 80. Changing character translation tables is easy (see example below). Also several tables that have been intentionally left free are reserved within APLIOTAB for future use thus allowing support of new terminal types.

Example: Changing Character Translation Tables

Assumed, your terminal is a display, the following sequence of commands will cause the table shown below appearing upon your screen:

```
vsapledt vsapl.asmlib(sources),@pliotab
x fi aplt
y _top
x;y;p36
```

```
0002800 APLT2741 @PLTABLE 'REVERSE=Y',-
0002900   '** TRANSLATION TABLE FOR 2741, 3767, ETC.      **  ',-
0003000   '   ------------------------------------------------ ',-
0003100   '0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |',-
0003200   '   ------------------------------------------------ ',-
0003300   '1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |',-
0003400   '   ------------------------------------------------ ',-
0003500   '2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |',-
0003600   '   ------------------------------------------------ ',-
0003700   '3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |',-
0003800   '   ------------------------------------------------ ',-
0003900   '4 |  |  |  |  |  |  |  |  |  |  |62|  |42|46|48|; |',-
0004000   '   ------------------------------------------------ ',-
0004100   '5 |49|  |  |  |  |  |  |  |  |  |( |6A|45|47|< |: |',-
0004200   '   ------------------------------------------------ ',-
0004300   '6 |+ |  |  |  |  |  |  |  |  |  |  |  |= |- |  |77|',-
0004400   '   ------------------------------------------------ ',-
0004500   '7 |  |  |  |  |  |  |  |  |  |  |43|70|59|44|41|) |',-
0004600   '   ------------------------------------------------ ',-
0004700   '8 |  |A |B |C |D |E |F |G |H |I |  |  |  |  |  |  |',-
0004800   '   ------------------------------------------------ ',-
0004900   '9 |  |J |K |L |M |N |O |P |Q |R |  |  |  |  |  |  |',-
0005000   '   ------------------------------------------------ ',-
0005100   'A |  |  |S |T |U |V |W |X |Y |Z |  |  |  |  |  |  |',-
0005200   '   ------------------------------------------------ ',-
0005300   'B |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |',-
0005400   '   ------------------------------------------------ ',-
0005500   'C |  |63|75|73|65|52|_ |66|67|57|  |  |  |  |  |  |',-
0005600   '   ------------------------------------------------ ',-
0005700   'D |  |68|''|69|| |76|58|* |? |53|  |  |  |  |  |  |',-
0005800   '   ------------------------------------------------ ',-
0005900   'E |  |  |64|54|56|74|51|72|55|71|  |  |  |  |  |  |',-
0006000   '   ------------------------------------------------ ',-
0006100   'F |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |',-
0006200   '   ------------------------------------------------ ',-
0006300   '   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  '
```

This table represents the character translation table used upon input from a 2741 terminal. A reversed table for output is generated automatically, because the first parameter of the @PLTABLE macro instruction is specified as 'REVERSE=Y'. If this parameter is specified as 'REVERSE=N', an output translation table must be allocated exactly behind the input translation

table. The @PLTABLE macro instruction specified with 'REVERSE=N' should be used to do this.
Changing an existing table is easy: the cursor must be moved to the field that must be changed and a new value may be typed in. Values can be specified as characters (A, Z, $ etc.) or in hexadecimal representation (40 for blank, 16 for backspace, etc.). A ' and a &, however, must be specified as '' and && as usual in assembler programs. Spaces indicate, that no translation is required.
Translation tables that are not yet used can be located using the REDIT command : find APLTDUM .

2.5.1.3. Library Maintenance

Modules: APLIBADD, APLIBCTL, APLIBDEF, APLIBDEL, APLIBLIB, APLIBLOA, APLIBLOC, APLIBNAM, APLIBPOD, APLIBSAV and entry points SYS@LGET and SYS@LPUT in @PLPRIVI

The module APLIBCTL, the library management control block mainly contains DCBs for library I/O and export/import facility. A special facility of TSS is used for the library I/O:

)LOAD: A GET macro instruction is issued with a data address specified on a page boundary (WSMFREEA= Start of workspace data on disk is forced to be allocated at a page boundary by APLSCINI). Upon execution of the GET macro TSS will neither transfer data from external storage to virtual memory nor use a buffer. Only the external storage addresses of the workspace pages are mapped into the user's page tables starting at the page specified in the parameterlist for GET. Pages are actually moved from external storage to virtual memory at the time the user references them.

)SAVE: If the user saves his currently loaded workspace without changing its library number or workspace name, only pages that have been modified since the workspace was loaded will be re-written to external storage by the PUT macro instruction, that is used to save the workspace. This only works provided that the DCB for the current workspace is still open. For that reason several DCBs are reserved for library I/O in APLIBCTL, so that the DCB associated with the current workspace can be kept open even during a)COPY or)PCOPY command. You must be very careful, if you want to increase the number of DCBs in APLIBCTL, because OPEN or FIND macro instructions (depending on LRECL being specified in the DCB or not) allocate buffer space, although buffers are not used by GET or PUT under VSAPL. If too many DCBs are open at the same time, the user will be forced to decrease his maximum workspace size or will get an ABEND due to software generated program interrupt 60 (not enough virtual memory).

Note: The GET and PUT macros are executed in privileged state (entry points SYS@LGET and SYS@LPUT in @PLPRIVI), because some fields in the RESTBL associated with the current library must be changed to allow GETs and PUTs for amounts of data greater than 1 megabyte. So only one GET is necessary to load a workspace and only one PUT to save it. The RESTBL is restored to its original status upon return from GET or PUT. The rest of library management is done using standard facilities concerning handling of partitioned data sets as FIND, STOW, OPEN and CLOSE macro instructions.

Summary of library management functions:

APLIBADD - create a new member in a library

APLIBDEF - create a new library

APLIBDEL - delete a member

APLIBLIB - locate a library in the user's catalog

APLIBLOA - load a workspace

APLIBLOC - locate a member in a library

APLIBNAM - read the program module dictionary (PMD) associated with a workspace and perform some checks to be sure, that it is really a workspace (remember, that VSAPL workspaces are saved as TSS load modules and, therefore, have a PMD)

APLIBPOD - read the directory of a library (for)LIB command)

APLIBSAV - build the PMD for a workspace and save the workspace data (this module is an adaption of the TSS CSTORE module, but has been modified to allow saving more than 1 megabyte)

At last a few words about the concept of public, project and private libraries implemented under other VSAPL versions. TSS VSAPL does not distinguish between public, project and private libraries. However, a user may define public or project libraries using the TSS SHARE and PERMIT commands. TSS VSAPL checks the user's access to a library before execution of a)SAVE or)DROP request.

2.5.1.4. Interrupt Handling

Modules: APLPISIR,APLPIDIR,APLPICTL

VSAPL requires interrupt handling for program interrupt and attention. APLPISIR defines interruption handling routines upon APL initialization, and after the execution of a TSS command by auxiliary processor APL1Ø0, APLPIDIR deletes all interruption handling routines when the user terminates his APL session. Upon occurance of a program interrupt control is passed to entry point SCSPIE in Executor module APLSCERR. SCSPIE saves the contents of all registers at the point of interrupt in the workspace and notifies the Interpreter, that a program interrupt has occured. If the Interpreter cannot recover from a program interrupt, an attempt will be made to save a CONTINUE workspace into user's library zero and the APL session will be terminated. Upon occurances of an attention interrupt control is passed to entry point SCATTN in module APLSCTYP. Since the Interpreter does not recognize asysnchronous attention interrupts SCATTN only sets some bits in the perterm header, thus allowing the Interpreter to handle an attention interrupt whenever it wants to do so. During APL initialization a special interruption handling routine is used to check whether VSAPL runs on a machine with APL microcode assist feature or not. APLSCINI issues a microcode instruction. If a program interrupt 1 (operation) occurs during execution of the microcode instruction, it will be assumed, that microcode assist is not available on the machine VSAPL is running on.

Interrupt control blocks (ICBs) are allocated within control block APLPICTL. The @PLPICTL macro instruction can be used to get access to APLPICTL.

2.5.2. Privileged Routines

Module: @PLPRIVI

Module @PLPRIVI contains the privileged part of TSS VSAPL. There are two types of entry points in @PLPRIVI:

* entry points, that define the starting address of a routine performing privileged service for VSAPL

* entry points, that define 2 address constants containing the R-CON and V-CON value of a privileged routine, which is part of the TSS system. This facility can be used to call TSS commands directly without passing control to the command analyser, for instance.

To ensure that a tricky user cannot execute privileged routines using the VSAPL enter code 165, checks are made within the enter code support module CZAPLMR: either the 2 address constants register Ø is supposed to point to upon entry to CZAPLMR must be located inside @PLPRIVI or, if the address constants are defined within the calling routine, the 2nd word (the V-CON) must specify the address of one of @PLPRIVI's privileged routines. A system programmer (authority O or P), however, may define his own module @PLPRIVI and use the VSAPL enter code 165 to test a privileged routine he is just working on.

2.5.2.1. Message Handling

Entry points SYS@MSGØ and SYS@MSG1 are used to support the)MSG OFF and)MSG ON commands. The privileged SETUP macro instruction is used to set or reset the message receive bit in the user's TSI.

2.5.2.2. Controlling User's Permanent Storage Ration

Entry point SYS@PRAT is used to retrieve the maximum number of external storage pages that can be used by the current task and the number of pages that are already in use by this task. This information is needed for the)QUOTA command.

2.5.2.3. Calling Command Routines

Some commands (such as LOGOFF) are called directly from VSAPL. The R-CON and V-CON values of those command routines are defined within module @PLPRIVI and can be accessed using a SYS entry point.

Example: If you want to call LOGOFF, you might specify:

```
L    Ø,=V(SYS@AFN1)  address of LOGOFF's R-CON and V-CON
LA   15,165          the VSAPL enter code
ENTER                enter command routine
```

2.5.2.4. Retrieving Information from the Active User List

Entry point SYS@UID locates the TASKID specified as parameter in the active user list (CHBAUL) and returns the USERID associated with the TASKID, if the TASKID could be found in CHBAUL. SYS@AUL locates a USERID in the active user list and returns the TASKID, if the USERID is in CHBAUL.

2.5.2.5. Miscellaneous Privileged Services

SYS@LGET - privileged parts of)LOAD
SYS@LPUT - and)SAVE

SYS@MCAS - entry to TSS MCASTAB command

SYS@GDDN - increment to generated ddname count (for ddnames starting with $$$) in the task common (SYSTCM) .

SYS@AUTH - give a privileged user (authority O or P) authority U during load of public csect @PLSVMEM, because the dynamic loader does not load public csects for privileged users, so that they cannot share variables with other users (the authority code is restored to its original value, however, as soon as loading of @PLSVMEM has been done).

2.6. The VSAPL command

The VSAPL command must be installed using the TSS command

```
BUILTIN VSAPL,VSAPL
```

Format of the VSAPL command:

```
VSAPL TERMTYPE=  one or two characters terminal id,
      LIBNO=     integer without sign,
      WSNAME=    up to 8 characters workspace id,
      APS=       ( a list of entry points for auxiliary
                 processors, each up to 8 characters long),
      WSSIZE=    integer without sign,
      MAXSIZE=   integer without sign,
      SHARE=     Y | N,
      ATTN=      ON | OFF,
```

TERMTYPE:

SC	3270 without APL keyboard
PR	Printer (for nonconversational tasks)
DE	DEC writer terminal
HP	Hewlett Packard Display
51	5100 intelligent terminal

The following terminals are recognized automatically (the termtype parameter must not be specified):

IBM 2741,3767,etc.
IBM 3270 with APL Keyboard
IBM CMC

LIBNO:

any number from 000000 to 99999
specifies the library VSAPL will use to load the initial workspace specified in the WSNAME parameter.

WSNAME:

name of the initial workspace; the user will get a CLEAR WS, if he does not specify the WSNAME parameter.

APS:

a list of entry points enclosed by paranthesis; each entry point must be seperated from each other by a comma and must specify the location the initial sequence of an auxiliary processor starts at.

WSSIZE:

initial workspace size; default one megabyte. The value specified in the WSSIZE parameter may optionally be suffixed by M (for megabyte), P (for pages) or K (for kilobyte).

MAXSIZE:

maximum workspace size for current APL session; default: 8 MB. Value may optionally be suffixed by M, P or K (see WSSIZE).

SHARE:

Y enable multitask shared variables. Remember, that all users specifying SHARE=Y use the same virtual memory to store all their shared variables (even those shared with auxiliary processors). The shared storage might soon become full, if too many users specify SHARE=Y, so that it is better to leave the default for SHARE N as it is.

N disable multitask sharing (system default)

ATTN:

ON allow attention interrupts (system default)

OFF disallow attention interrupts
ATTN=OFF for instance is used for VSAPL applications, that do not allow the user to use APL itself. The WSNAME parameter can be used to load the application workspace. If this workspace has a Quad-LX specifying a function, that is able to control all the user's activities (the stack processor APL101 can be used to issue APL commands), the user will not notice, that he is working with APL.

3. System Generation and Maintenance

3.1. The TSS VSAPL Distribution Tape

The VSAPL distribution tape contains the following datasets moved to tape by VT command:

VSAPLLIB	(VP)	linkedited version of VSAPL and all distributed auxiliary processors.
VSAPL.SYSGEN	(VI)	system generation script
VSAPL.ASMLIB	(VP)	source and object for Executor, auxiliary processors and system service routines, as well as macro library
VSAPL.SYSLIB	(VP)	source and object for SYSLIB modules and installation utilities as well as procdefs (member SYSPRO) and updates for CHBET and CHBVM
VSAPL.PRINTALL	(VI)	script to print all VSAPL sources
VSAPL.UPDATE.DAT#####		several update datasets for VSAPL, if any update exists (##### represents a date with the format YYDDD)

Updates:

Updates for VSAPL are distributed as regionized VISAM datasets. They are automatically installed by the VSAPL generation script or the VSAPLASM procdef (see below). The dataset name for a VSAPL update should have the following format:

VSAPL.UPDATE.DATyyddd

where yyddd represents the year and the day the update has been written. This ensures, that updates are always executed in correct order, because the TSS EXPAND command with parameter SORT specified as Y is used to retrieve the update datasets. The region name must specify the module the update belongs to. Installation that want to install their own updates for VSAPL should build them according to rules described above.

Description of the most important members in VSAPL.ASMLIB:

member SOURCES : regionized dataset containing all sources for execcutor modules, auxiliary processors and non privileged system service routines

member MACROS : regionized dataset containing all macros used to assemble VSAPL modules

Description of the most important members in VSAPL.SYSLIB:

member SOURCES : regionized dataset containing all sources of privileged system service routines, the VSAPL enter code module CZAPLMR, CZATM (support for SET/GDV macros) and the shared storage module @PLSVMEM

member UTILITY : regionized dataset containing all sources for VSAPL system installation utilities

member UPDATES : regionized dataset containing updates ready for ALTER for TSS modules CHBET and CHBVM. The updates are used to establish the VSAPL enter code 165.

member SYSPRO : regionized dataset containing usefull procdefs that will be generated by the VSAPL installation script.

VSAPL.SYSLIB(SYSPRO) contains several procdefs that are useful for VSAPL maintenance.

Procdef VSAPLGEN:

```
             |‾ALL         ‾|
   VSAPLGEN  | EXECUTOR     |,LISTINGS=|‾Y‾|
             | INTERPRETER  |          |_N_|
             |_LINKEDIT    _|
```

This procdef can be used to install the VSAPL system or parts of it.

ALL — Generate the entire VSAPL system including the Interpreter, the Executor and auxiliary processors. The VSAPL basic material tape must be available (see also description of INTERPRETER parameter below).

EXECUTOR — Install a new version of the Executor. Linkediting is done automatically.

INTERPRETER — Install a new release of the Interpreter. The VSAPL basic material tape must be available. Also it is necessary to check the DDEF command for the basic material tape in the script VSAPL.SYSGEN, because the file sequence number for the Interpreter object code may change from time to time.

LINKEDIT — Execute the linkedit step of VSAPL generation. This step is always executed if one of the other parameters is specified in the VSAPLGEN procdef, but sometimes it is better to execute the linkedit step separately. For instance, if only one Executor module has been changed, it will help to save time when this specific module is assembled with the VSAPLASM procdef (see below), followed by a linkedit step.

LISTINGS=Y — Print the list datasets produced during assembly and linkediting.

LISTINGS=N — Suppress printing of list datasets and erase them. Note: Printing of the output produced by the ALTER command cannot be suppressed.

VSAPLGEN sets some defaults that are used within the VSAPL.SYSGEN script and then executes that script as a batch task. VSAPLGEN should not be used while another batch task started by VSAPLGEN is active. Note: The PROFILE command is used to have the defaults set by VSAPLGEN available in the batch task. All defaults set prior to VSAPLGEN will therefore become permanent, too.

Procdef VSAPLASM:

```
VSAPLASM module,UPDATES=|Y|,LISTINGS=|Y|
                        |N|           |N|
```

This procdef assembles one VSAPL module.

module — specifies the module going to be assembled.

UPDATES=Y — specifies, that all updates available for the required module are executed before assembly.

UPDATES=N — suppresses execution of updates. Version Ø of VSAPL (Oct. 78) will be assembled.

LISTINGS=Y — print assembly list datasets.

LISTINGS=N — suppress printing of list datasets (erase them).

Procdef VSAPLEDT:

```
VSAPLEDT regionized ds,region
```

Procdef to edit VSAPL sources and updates. Sources are stored as regionized members of VSAPL.ASMLIB (non-privileged) or VSAPL.SYSLIB (privileged). Updates are regionzed datasets named according to rules explained above. VSAPLEDT calls the research editor (REDIT) to perform edit functions. All REDIT commands can be used except PA and CONEDIT which migth cause confusion.
Note: VSAPL sources cannot be changed using VSAPLEDT, although the procdef may be used to list them. APLIOTAB (character translation tables) is the only module that can be changed with VSAPLEDT. Updates, however, are always allowed to be changed.

regionized ds — specifies the source library the required module is stored in. All regionized members of VSAPL.ASMLIB and VSAPL.SYSLIB and all update datasets can be used. The member name must be specified (for example: VSAPL.ASMLIB(SOURCES)), if editing inside a member is intended.

region — specifies the module name of the module going to be edited.

Procdef VSAPLTST:

VSAPLTST

This procdef loads the non-linkedited version of VSAPL (for debugging), so that all csect names and entry points are available to be used by PCS. The VSAPL command must be used to start the APL session.

Example:

```
'load the non-linkedited version of VSAPL';
VSAPLTST;
'trace all program interrupts and display the VPSW';
AT SCSPIE;DISPLAY L'50'.(0R,8);STOP
'stop upon occurance of an ABEND';
AT APLIOCTL.APLABEND;STOP
'start the APL session';
VSAPL APS=(APL100,APL101);
```

To install VSAPL the following is to be done:

1. TV all datasets from the VSAPL distribution tape

2. Have the IBM VSAPL basic material tape available

3. Macro libraries ASMMAC and ASMNDX must be available

4. Issue the following command sequence:

   DEFAULT UTILITY=Y,INTRPRTR=Y,ASSEMBLY=Y,LISTINGS=Y,UPDATES=Y;
   PROFILE; EXECUTE VSAPL.SYSGEN;

   Be sure, that no CPU time limit applies to your nonconversational task. If you want to have assembly and linkedit listings to be suppressed, set the default for LISTINGS to N. Wait until the script is executed. To suppress updates set default of UPDATES to N (Version Ø of VSAPL will generated).

5. Copy members @PLPRIVI and @PLSVMEN from VSAPL.SYSLIB to SYSLIB(Ø) and SYSLIB(-1).

6. Build a delta dataset and copy members CZAPLMR and CZATM to it (if support for SET and ADV macros is already installed in your current version of TSS, forget about all what have been said concerning CZATM in this documentation).

7. Install updates for CHBET and CHBVM from member UPDATES in VSAPL.SYSLIB

8. Punch a card for your delta dataset built in step 6 and perform a longstart for TSS.

9. If you've reached this step , you will have:

   * modules CZAPLMR and CZATM in IVM
   * VSAPL enter code 165 installed
   * a macro library VSAPL.MACROS and a macro index VSAPL.INDEX (generated by VSAPL.SYSGEN)
   * some PL/1 programs (installation utilities) and some useful procdefs in your USERLIB
   * and last not least a linkedited version of VSAPL in the library VSAPLLIB

1Ø. Make a copy of VSAPLLIB available to the users and install BUILTIN VSAPL,VSAPL in SYSLIB.

3.2. Installing a New Interpreter Release

Provided, that you have already installed VSAPL for the first time, you can use procdef VSAPLGEN to install a new Interpreter release:

```
VSAPLGEN INTERPRETER,LISTINGS=|_Y_|
                              |_N_|
```

This will generate a nonconversational task executing parts of the installation script VSAPL.SYSGEN (don't forget to have IBM VSAPL basic material tape available).

3.3. Modifying Executor Modules, Auxiliary Processors and System Service Routines

To modify one of those modules mentioned above, build an update dataset ready for the ALTER command using VSAPLEDT:

```
VSAPLEDT VSAPL.UPDATE.DAT#####,module
```

where ##### represents the current date in the format YYDDD.
Then issue the following command sequence:

```
VSAPLASM module,|_Y_|,|_Y_|;
                |_N_| |_N_|

VSAPLGEN LINKEDIT,LISTINGS=|_Y_|;
                           |_N_|
```

If you want to change a module without using an update dataset, you might issue:

```
VSAPLEDT VSAPL.ASMLIB(SOURCES),module
    .
    .
    .
      here you are within the research context editor
    .       (REDIT) and are allowed to modify the module
    .
    .
    .
   FIL;Q

VSAPLASM module,|_Y_|,|_Y_|;
                |_N_| |_N_|

VSAPLGEN LINKEDIT,LISTINGS=|_Y_|;
                           |_N_|
```

Note: Only module APLIOTAB (character translation tables) may be changed using the command sequence listed above! All changes applied to other modules will be automatically ignored by VSAPLEDT.

3.4. VSAPL Generation Utilities

Three utility programs written in PL/1 perform linkediting during VSAPL generation:

INCLUDE,
COMBINE and
RENAME

INCLUDE generates a source dataset for the TSS LNK command and is used to pack several csects and psects into one module. Execution of the LNK command is automatically performed within INCLUDE using TSS OBEY facility (subroutine SYSOBP).

Format: INCLUDE 'output-module input-library output-library';

Csect and psect names are retrieved from SYSIN according to following conventions:

```
CSECTS:
  .
  .  (csect names must be typed in here)
  .
PSECTS:
  .
  .  (psect names must be typed in here)
  .
(a zero length line to signal end of data)
```

Note: If INCLUDE is excuted in a non-conversational task, each input line must be padded with blanks up to maximum record length (with the exception of the zero length line indication the end of the input stream).

COMBINE is used to pack all csects of a given module into one csect and all psects into one psect. To get the resulting module as small as possible COMBINE attempts to optimize packing of csects and psects. This implies that the TSS LNK command must be executed for several times before COMBINE is complete (the LNK command is automatically executed within COMBINE using subroutine SYSOBP).

Format: COMBINE 'module library';

RENAME is used to rename entry point or csect names or to remove them from the external symbol dictionary of the module.

Format: RENAME 'module library';

Names going to be renamed or removed are retrieved from SYSIN according to following format:

```
name1 name2     (to rename an entry point or a csect)
name3           (to remove an entry point or a csect)
```

If two or more entry point or csect names start with the same sequence of characters, these charcters may be specified as name1 or name3 to rename or remove all entry points and csects starting with that sequence. For instance, you want to rename all csects starting with APL, you migth specifiy:

```
RENAME 'TSSVASPL VSAPLLIB';
APL EXEC
(zero length line to terminate input)
```

Csect APLSCINI will be renamed to EXECCINI, and so on. Note: If RENAME is executed in nonconversation task, input lines must be padded with blanks up to maximum record length of the SYSIN dataset due to PL/1 conventions.

Appendix A: Module Dependency Overview

called by:	module:	calls:
Executor modules:		
@PLSCFXI	@PLSCDPY	@PLIOGET, @PLIOPUT
@PLSCFXI program interrupt	@PLSCERR	@PLVMA, @PLSCLIB, @PLSCFXI, @PLMDATE
Interpreter	@PLSCFXI	@PLSCDPY, @PLSCERR, @PLSCINI, @PLSCLIB, @PLSCMSC, @PLSCMSG, @PLSCSHV, @PLSCTYP
@PLSCFXI, @PLSCSHV all aux. proc.	@PLSCINI	@PLVMA, @PLSCSVI, @PLPISIR, @PLPIDIR, @PLIOSEL, @PLIODEF, @PLIOCLO
@PLSCFXI @PLSCERR @PLSCLIB	@PLSCLIB	@PLVMA, @PLIBSAV, @PLIBADD, @PLIBDEF, @PLIBDEL, @PLIBLIB, @PLIBLOA, @PLIBLOC, @PLIBNAM, @PLIBPOD
@PLSCFXI	@PLSCMSC	@PLVMA, @PLMDATE, @PLPRIVI
@PLSCFXI	@PLSCMSG	@PLPRIVI
@PLSCFXI	@PLSCSHV	@PLSCINI, @PLVMA
@PLSCINI	@PLSCSVI	@PLVMA, shared var. man. (@SUSH...)
@PLSCFXI attention interrupt	@PLSCTYP	@PLIOGET, @PLIOPUT
auxiliary processors:		
@PLSCSVI	@PL100	@PLSCINI, @PLPIDIR, @PLPISIR
@PLSCSVI	@PL101	@PLSCINI, @PLVMA
@PLSCSVI	@PL111	@PLSCINI
@PLSCSVI	@PL200	@PLSCINI
@PLSCSVI	@PL333	@PLSCINI, @PLPRIVI

Modules that are not called directly from the Executor:

eodad exit	@PLIOERR
all routines using @PLENTRY, @PLEXIT or @PLERROR macros	@PLERMON

Privileged support:

@PLVMA, @PLIBLOA, @PLIBSAV, @PL100, @PLSCMSC, @PLSCMSG	@PLPRIVI

References
==========

/ 1/ A.D. Falkoff, K.E. Iverson, The Design of APL, IBM Journal of Research and Development 17, 1973, pp. 324-334

/ 2/ R.H. Lathwell, System Formulation and APL Shared Variables, IBM Journal of Research and Development 17, 1973, pp. 353-359

/ 3/ APLSV User's Manual, Form No. SH20-1460, IBM Corporation

/ 4/ VS APL Program Logic, Form No. LY20-8032, IBM Corporation

/ 5/ A.D. Falkoff, D.L. Orth, Development of an APL Standard, APL Quote Quad 9, 1979, pp. 409-453

/ 6/ H.W. Homrighausen, VSAPL under TSS: Implementation, Use, and First Experience, Minutes of the TSS Project Meeting, SHARE 50, Denver,Co, 1978

/ 7/ TSS Concepts and Facilities, Form No. GC28-2003, IBM Corporation

/ 8/ VSAPL for TSO, Yale University System Guide, LY20--2255, IBM Corporation

/ 9/ D. Bartel, D. Erwin, Implementation of Large APL Workspaces in a Virtual Memory Environment, Proc. SEAS Anniversary Meeting, Hamburg 1979

/10/ VSAPL for CMS: Writing Auxiliary Processors, Form No. SH20-9068, IBM Corporation