MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Reply to: Project MAC 545 Technology Square Cambridge, Mass. 02139

Telephone: (617) 864-6900 x5851

April 2, 1965

The following is Section IV of the Design Notebook.

DISTRIBUTION

- V¥ Vyssotsky
- J. Couleur
- R. Fano
- E. Glaser
- F. Corbató
- R. Graham
- M. Daggett and M. Wagner
- R. Daley and O. Wright
- S. Dunten and M. Child
- G. Schroeder and L. Pouzin
- P. Crisman and D. Oppert
- J. Poduska and J. Saltzer
- C. Garman and R. Stotz
- M. Bailey
- D. Edwards

The SHELL: A Global Tool for Calling and SUBJECT: Chaining Procedures in the System

FROM: Louis Pouzin

April 2, 1965 DATE:

CONTENTS

		Page
I.	Definition of Commands	R-1
II.	Requirements	R-1
III.	Commands as Subroutines	R - 2
IV.	The SHELL	R - 3
V.	Stack Management	R - 7
VI.	Arguments Management	R-18
VII.	Requests Stacking	R - 21
VIII.	Meta-arguments	R - 22
IX.	Permanent Options	R - 24
х.	SHELL Organization	R - 27
XI.	SHELL Flowcharts	R-29
XII.	Further Needs for Common Procedures	R - 33
XIII.	Summing Up	R - 34

* * 4

ABSTRACT

See chapter 13, page R-34.

I. Definitions of Commands

In order not to forget the particularities of commands, it is useful to restate some definitions.

A command is a program which can be initiated from a console, by typing a message directly to the supervisor.

This means that no subsystem need by called first, in order to interpret the user message.

Furthermore, for convenience, the calling message is interpreted according to some general convention: any blank gap is a delimiter of arguments; the first argument is specifically the name of the command; the other arguments are passed over to the command as actual values of formal parameters.

II. Requirements

2.1 The previous definitions imply that a command MUST be designed while keeping in mind the user, sitting at his console, wondering about what might be going on, mistyping or forgetting arguments, even if fully aware of the conventions, and possibly interfering with the command by hasty quits, carriage returns, and other temperamental reactions. These aspects, much more than the particular form of calling procedure, make commands a certain more elaborate class of routines. Indeed, unlike a calling program, a user is barely satisfied with a list of values or error codes, as mere results of the execution. A minimum editing is necessary so as to make the outcome meaningful without looking up a deciphering table. And if some unexpected situation has occurred, the command must specify clearly what the user has to know for further action. Again, unlike a calling program, a user is not assigned once and for all a predetermined set of instructions. One does not know what he plans to do next, and he will eventually overlook some yet highly recommendable checking, if the command does not warn him against any possible misunderstanding as to what has been performed.

2.2 Since the arguments may be typed from a console, they appear as strings of printable BCD characters. Conversion is usually required for those arguments that are to be processed under a different mode, (integer, floating-point) or need a different editing (left or right justification, filling zeros or blanks). Nothing but the command itself is able to perform those conversions, since each command uses a particular format, the variation of which may require a decoding of the whole string of arguments. It amounts to the same thing to say that each command requires a particular decoder for its arguments string.

2.3 For identical reasons, any output sent to a console must be in printable form, and usually inserted into a message intended to clarifythe meaning of the output.

III. Commands as Subroutines

3.1 One might imagine a command as a common procedure with two sets of entries, one set for calls initiated from a console, one set for calls from an internal program. Each set of entries could belong to an associated segment, and end up by calling a common main segment. Similarly, the main segment of the command could branch toward either appropriate terminating segment, depending on the type of the calling procedure. Such an organization (double head, double tail) is not recommended because it would settle the difference of nature between a call from a console, and a call from a program. It would also likely result too often in partial implementations, as private ones usually are, with one head, or one tail missing. Moreover, it would downgrade subroutines by making them inappropriate for being used as commands.

3.2 Indeed, the fact that commands ought to be more elaborate does not imply that other subroutines be never used as commands, through direct calls from the console. The <u>ability to call any subroutine by an entry</u> <u>name</u>, whether <u>from a console</u> or a program brings about a high level of generality, as the difference between commands and subroutines comes out of a matter of degree, rather than a matter of nature.

3.3 For example, it would be very convenient to be able to test from a console a subroutine designed to be normally imbedded in a set of surrounding calling programs, and which in turn calls other subroutines. Present techniques require the writing, and unfortunately also the debugging of a complete set of dummy programs. Such an extraneous task is often overlooked and the complete checkout of the program is postponed until all the components are available. As a result the debugging is all the more difficult and more bugs keep hiding for long periods of time.

3.4 In the following we are going to develop some scheme of implementation which is hoped to meet the general principles outlined above.

IV. The SHELL

4.1 We may envision a common procedure called automatically by the supervisor whenever a user types in some message at his console, at a time when he has no other process in active execution under console control (presently called command level). This procedure acts as an interface between console messages and subroutine. The purpose of such a procedure is to create a medium of exchange into which one could activate any procedure, <u>as if it were called from the inside of</u> <u>another program</u>. Hereafter, for simplification, we shall refer to that procedure as the "SHELL".

4.2 The description given in the following is based on the proposals for GE 636 segment conventions as sketched out by Professor Corbato and the proposals for the GAP assembler by R.M. Graham. But the basic ideas seem to be adaptable as well to some changes if these were necessary.

It is assumed that the supervisor initializes a stack whenever it initiates a process for a user; so should it be for commands, and generally speaking, whenever the user talks to the supervisor. In other words, typing a message to the supervisor amounts only to set up

<u>a call to the SHELL</u>. The supervisor stores the console message in the stack and calls the SHELL with one argument pointing to the message so that the SHELL is also a regular segment, with an entry name making it possible to call it directly from the console, or from a subroutine, either. Passing arguments via the stack allows all recursive calls. Suffice it to say here that such a property is in prevision of a macrocommand scheme. Clearly, if the call is issued from the supervisor, the descriptor of the "present procedure" base is set so that it creates a trap to the supervisor when the SHELL attempts to execute a return to the calling program.

4.3 It is suggested that the SHELL be equipped with a comprehensive set of error returns. This would allow complete control of error conditions at the program level and pave the way for automatic runs using no console. Although a complete proposal would require further study one can yet propose the following options:

ERRØR	1:	Need more arguments
ERRØR	2:	Some anomoly occurred in user's data.
		Possibly not fatal.
ERRØR	3:	Some fatal error due to user's setting.
ERRØR	4:	Interruption due to restrictions in system used.
		(Track quota, time runout, device not available, etc.)
		Restart possible if the environment is modified.
ERRØR	5:	Some error for unidentified reason.
		Maybe hardware trouble.
		Interruption as is.

Some more experience is necessary to estimate how many error returns would be really useful, and what conditions should be grouped altogether.

4.4 The message typed on the console is stored by the supervisor into the stack. It seems desirable that <u>no argument breaking</u> be performed up to this point. The string of characters is stored as is, with a character count in the first nine bits of the string according to the general conventions to be established throughout the system for BCD strings.

It is possible that we put, some day, more refinement in the definition of an argument in a command list. For example, QUOTE sign allowing literals including blanks. Hence, there is some reason to maintain the complete picture of the message up to the point where there is a necessity to break it into a set of arguments. On the other hand, some commands might perform their own scanning for special purposes, and use different delimiters, like arithmetic operators.

4.5 However, the breaking of the console message uses far more frequently blanks as delimiters. Therefore, the SHELL makes up a list of words, each one being a single string headed by its bit count and sets up a calling sequence with two arguments, and all the possible error returns, (even those unused). Breaking is done wherever one or more spaces or tabs occur in the message.

The first argument is the number of strings (single words of text), and the second is the beginning of the list, stored in consecutive machine words. For convience, it may be desirable to store every string beginning at an even location. The storage conventions should be the same as those retained for the STRING pseudo-operation in GAP.

The very first string of the list is used as the name of the procedure segment to be called. A class name is automatically provided, according to the system conventions. Then the SHELL executes a CALL to the specified segment, using only the stack for storage of data. Although it is arbitrary, one may assume for simplicity that the entry name used is the same as the procedure name. If it were not, another symbol recognized through a meta-argument, such as (ØRIG), should specify the symbol reference of the particular entry to be used. (See more details in paragraph 5.7.)

4.6 Although the "binder" is not yet designed, it is certain that a user must have some way to specify whether he wants to use a common segment, supplied by the system, or one of his own. Presumably, some pointer in an area of his file directory will carry the answer.

Thus, by using this standard mechanism, the call issued by the SHELL activates either a common segment, say a command, or a private segment of the user, which he may prefer to substitute for a time to the system procedure.

4.7 The called procedure is executed, and interprets the successive arguments according to its own conventions. If exceptionally the complete picture of the message is necessary, the procedure can trace back one call in the stack, to get the message as it has been transmitted to the SHELL, namely a single string with spaces and tabs as typed by the user. Interactions with the console are controlled by the procedure (there may be none). Finally, when the execution is completed, the procedure restores registers and executes a return to the calling program, which here happens to be the SHELL. One of the error returns may be used if there is such a need.

4.8 In case of a normal return, the SHELL saves arguments returned by the called procedure, as it will be discussed later. Then it returns to the calling program, usually the supervisor.

In case of an error return, a standard error procedure is called, from the common package, or from the user's package, according to the previous setting of an "error" segment, (see permanent options in chapter nine).

If the procedure called is missing, it is assumed that a trap to the supervisor will automatically initiate a generating process in an attempt to create the missing segment. Otherwise, an error return in the SHELL could do it.

4.9 An important facility is that the SHELL being itself a common procedure <u>may be replaced by a private one</u> supplied by the user. On that way, not only a particular procedure can be replaced on user's choice, but all conventions about typing commands may be tailor-made to user's wishes just by providing his own SHELL. One can, for example, build his own set of meta-arguments (see chapter eight for discussion on metaarguments) or use "," as a delimiter between arguments, or convert all numeric arguments into floating point numbers, etc... 4.10 We already mentioned in various places that users should be able to provide their own segments to be substituted for the system tools, whenever they wish. One can think of many ways in order to implement such a facility. The following way is suggested, although not better than others.

Whenever there is a need to establish a link to a segment which is not in core, the supervisor searches the user's file directory for that missing segment. If there is one, it uses it. This segment may be just the standard system tool if the user has set up a linked file entry in his directory. Otherwise, the supervisor will attempt to find the segment among the set of common procedures. If it does not suceed, the process is interrupted, and diagnostics procedures are called.

V. Stack Management

5.1 As we have said, the stack is initiated by the supervisor as part of the initialization of a user's process. Normally, a fresh stack is started for every request to the supervisor, i.e. the previous contents of the stack is lost. Indeed there must be some automatic cleaning out policy, otherwise user's would let grow their live information without much consciousness of system overhead.

5.2 However, this drastic policy is not always satisfactory, since it would be convenient to keep for awhile the previous contents of the stack. A particular entry to the supervisor may solve the problem, by setting a read only flip-flop which can be modified only through the supervisor. Whenever the supervisor initiates a new process, it starts a fresh stack if the switch is off, or it stores its own calling information from the current pointer if the switch is on, and turns it off.

This technique is preferred to a permanent setting, since anything permanent may be overlooked.

Procedures which do not want to leave information after their execution may call the supervisor to turn the switch off. Needless to say, if they are embedded in a larger procedure, the master program may modify this setting.

By convention, the QUIT signal sets the switch on, in order to allow any saving request without killing first the status to be saved.

5.3 For easy reference in the following, we call the switch BRØØM. The corresponding entry to the supervisor allows four types of calls (distinguished by a single argument code for example:)

- No change
- Turn ØN
- Turn ØFF
- Invert

Any call returns as a function value the status of the switch before modification, permitting to save it.

5.4 When controlling the execution of a procedure, the SHELL does not normally alter the setting of the BRØØM, i.e. the saving of the stack depends upon the procedure itself. But one can use some conventional metaarguments, when calling the SHELL, so that it preserves in any case the contents of the stack by turning the BRØØM on before the return. This provides the possibility to issue any sequence of requests from the console, while keeping a dormant status of a process partially executed, without having to enter explicitly the SAVE and RESTØR procedures.

5.5 In order to make this point clear, let us take some examples.

5.5.1 - A CØMBIN procedure reads several files and makes up a new file out of components. First, the supervisor sets up a call to the SHELL with the string of arguments to the CØMBIN procedure. Assuming that the BRØØM is OFF, the supervisor initializes a fresh stack on calling the SHELL. Then the CØMBIN procedure is called in turn, and assuming that no error condition occurs, returns to the SHELL. Since the only information to keep after execution is the created file, CØMBIN does not request the setting of the BRØØM, which is still OFF when the SHELL returns to the supervisor. Consequently, no dormant status has been kept on beginning, and the terminating status of the CØMBIN procedure will be lost on starting the next procedure. This means that the descriptor segment of the process itself will be lost, and the supervisor will forget all about the process.

5.5.2. - A hypothetical DEBUG system is called from the console in order to behave as a monitor system for debugging any collection of procedures. During a typical session, several programs must be edited, printed, assembled, some segments are to be created, or renamed, etc..., while the user proceeds debugging. All such tasks may be accomplished by using separate tools available in the system. However, they are completely independent from the DEBUG package; nevertheless the user wants to call on them, without loosing control thus far reached throughout DEBUG. The solution is simple. A QUIT signal, or a PAUSE request built in DEBUG will release control to the SHELL, then to the supervisor, but with the BROOM set ON. Consequently the next request typed to the supervisor will be stored on the top of the current stack, as a new call to the SHELL, and the procedure requested, whatever it is, will be executed without disturbing in any way the status of the DEBUG system. Then control is returned to the SHELL, which executes a return to the calling program. But, this point is important, the second supervisor call to the SHELL carried the machine condition of the interrupted DEBUG system; therefore the return from the SHELL lands directly into DEBUG, as if the procedure just executed had been initiated at the DEBUG level. Depending on the interrupting conditions, the program may be restarted immediately, or it hangs up on a waiting message instruction. One may ask why only one procedure can be executed before returning to the previous process. As we have said before, a reason is that some automatic restriction must prevent the user from letting unconsciously unterminated process pile upon one another at system overhead expenses. On the other hand there are several possibilities of executing on purpose any number of inbetween procedures, such as: enter an appropriate sub-system (e.g. macro-commands), explicitly SAVE and later on RESTØR, or as discussed below.

5.5.3. - In the previous example, we might have preferred a slight variation in the implementation of our DEBUG system. For

example, once a PAUSE request has been typed, all returns from the SHELL (i.e. after execution of in-between procedures) trigger a return from DEBUG to the supervisor. But, normally the BROOM has been set OFF on starting the first procedure following the pause, and consequently the next request should destroy the process. The normal solution is that DEBUG sets the BROOM ON before returning to the supervisor, then an indefinite number of procedures may be executed during the pause. In order to get DEBUG restarted, one may for example type the name of a procedure that is the "restart" entry to DEBUG itself. There are obviously some other methods for reentrance.

5.5.4. - We have seen that setting the BROOM ON is enough to allow keeping the process available. There are two other places where this can be done. First, the procedure called may be designed so that it always sets the BROOM ON before returning to the SHELL. An example would be a PATCH SEGMENT procedure, supposed to act upon a dormant process. Second, the SHELL may be directed to perform the setting, regardless of what conditions are left by the procedure. This is done through the meta-argument (SAVE), which forces the BROOM ON if it is encountered on scanning a request.

5.5.5. - It is hoped that the above examples have sufficiently pointed out the flexibility of the BROOM scheme, which quickly kills abandoned status, while providing conscious users with all gimmicks for interrupting and intertwining various levels of processes.

5.6 Any procedure called with a set of arguments may return results by modifying the data specified by the pointers in the argument list. In addition, the procedure itself may yield a "value", as when used in a function-like call. There will be some conventions in the system as to which registers on which locations in the stack are used to carry the function value. Therefore, when the SHELL takes back control on returning from the executed procedure, it saves the "value" registers in the stack in order to pass them back to its calling program, and to keep them available for any need. This saving may be suppressed if it is ascribed for convention to the procedure called.

In other words, when leaving the SHELL, one can get access to all the arguments returned from the particular procedure called, and the "value" of the procedure is also available in some machine registers, or some location specified according to the general conventions followed in the software. How to use these data will be discussed in paragraph 5.10 and 6.3.

5.7 Unless specified otherwise by the user (see paragraph 6.3) the SHELL stores in the stack all the data specified by the arguments. In other words, the SHELL contains a CALL macro of the form:

CALL sp PRØC, * ($\hat{\mu}$ sp † AL)

where PRØC and AL are the addresses, relative to the current pointer in the stack, where the SHELL has stored the pointers to the procedure to be called, and to its arguments, as explained in the following. (For understanding of the symbolisms used here, one should refer to section III of the design notebook: A Proposal for a Minimal Assembler, GAP, for the GE636 by R. M. Graham.)

Both PRØC and AL can be constants of assembly in the SHELL. But variables determined at execution time may be used if this turns out more convenient for implementing the SHELL. The CALL macro then comes:

CALL sp(0, X1 * (@ sp (0, X2))

where X1 and X2 are the index registers used to carry the variable addresses.

sp PRØC points to the <u>linkage information</u> to the procedure to be executed. Indeed, one does not know at assembly time the name of all possible procedures that the SHELL may have to call upon. Consequently, the SHELL sets up in the stack the linkage information according to the user requirements. The more general setting is:

spî PRØC	SEGAD	spî STP,F
	ARG	
	ARG	spî SSTP,F
	ARG	EXP,M
	•••	
spÎSTP	STRING	m, name of the procedure segment
sp İ SSTP	STRING	n, name of the selected entry
	•••	

EXP,n is any expression, whose <u>value is provided</u> on calling the SHELL, to be applied as modifier to the pair: PROCEDURE + ENTRY. The SHELL stores in sp \uparrow STP and sp \uparrow SSTP the BCD strings of the procedure segment and entry names, and stores in sp \uparrow PR \emptyset C + 3 the value provided as modifier.

The simplest case is: PROCEDURE where nothing but the procedure segment name is sent to the SHELL. Then the entry name is assumed to be identical to the segment name, and the modifier is set to zero.

An example of a somewhat sophisticated case is the following:

PRØCEDURE (ØRIGIN) [ENTRY]-5,*7

The group [ENTRY]-5,*7 is a single string. The SHELL uses brackets and commas for breaking it into proper components.

 $sp^{\uparrow}AL$ points to the <u>list of pointers</u> to the data. They are set up as ITS pairs by STP's instructions this allows the specification of data stored either in the stack (usual case), or in other segments indicated by the user.

5.8 The number of pointers making up the argument list may be determined by the SHELL as a result of the scanning of the input message. But the procedure which will be called may expect variable number of arguments, and so far there is no information which tells when to stop picking up the arguments from the SHELL. Therefore, we are proposing a classic method, which could also be generalized in the software.

The SHELL stores at the end of the argument list a special pointer, (commonly known as a "fence"), which can be checked by the called procedure, and indicates the end of the relevant argument pointers. A fence is preferred to an argument count, because it may be ignored by procedures that are written for expecting always a fixed number of arguments.

The pattern of the fence pointer is all but arbitrary, as long as it cannot be confused with an ITS pointer. However, the following suggestion might lead to economic coding.

The fence could be an EP pair as:

ARG **,EP ARG LØC

when ** has been set from the SHELL procedure base register. Am instruction as LDA apin,* which attempts to pick up this pair from the called procedure, will execute the pair of instructions stored in LØC, in the SHELL. But LØC contains

TRA 0,0

which return to the procedure at the location specified by index register 0. Thus a procedure expecting a variable number of arguments could execute a loop for getting blindly the successive arguments. The fence would automatically transfer control to the location specified by index register 0.

Such a method is not mandatory, since the procedure could as well pick up the successive pointers, and check for the EP modifier, as an arbitrary data word.

5.9 As said in 5.6, the procedure called by the SHELL may return results by storing new data into areas specified by the arguments of the call. This classic method is unfortunately restrictive in the sense that it requires that the calling program knows enough about the number of results, if not their size. This technique is not suitable for the SHELL, which must call arbitrary procedures. Therefore, in addition to the classic method, we are proposing a systematic way of handling resulting arguments, whose number and size cannot be known before execution.

The principle is that the called procedure modifies the contents of the old stack pointer, so that <u>after return</u> the area of the stack belonging to the calling program contains then the pointer to the return arguments list. The return argument list is a list of pointers to the data, terminated by a fence pointer. The return argument list and the data may be stored in any segments that the called procedure decides to use for that purpose. Only the pointer to the argument list needs to be in the stack. But nothing keeps from putting everything (argument list and pointers) in the stack. The only requirement is that the procedure, which is going to return this information, modifies the old stack pointer according to the amount of extra storage required. Using systematically the stack, rather than creating segments, would insure that no conflict arise between informations left by the same procedure called at different levels of recursion.

The following instructions perform the updating of the old stack pointer, when transferring back to the calling program:

EAPbp	sp†16,*	saving bases address
ADBsp	EXSTOR	EXSTOR contains the amount
STBsp	bp[18	of extra storage. Update top
LDB	bpî 0	restore bases
LDR	spî 8	restore registers
RTD	sp120	return

The calling program may ignore this setting, if it decides not to process the return arguments. However, they are available until control returns to a higher calling program. If the return arguments are processed (and the SHELL does), the following sequence is suitable in order to pick up the arguments:

EAPap	sp118,*
EAPap	ap^-2,*

Then ap points to the argument list returned by the procedure the more recently called. Indeed, the pointer to the argument list is stored just below the new top of the stack.

Other methods can be imagined, as a pointer left in one pair of base registers; but this would create an exception in the restoring of machine conditions, hence open the door to restrictions in the calls across independent procedures. Consequently, the stack is preferred to any other form of storage in order to preserve independence and recursivity among the system tools.

We suggest that the expansion of the stack and the making of pointers and argument lists be handled through a standard set of instructions generated by a RETURN macro. Indeed, there are three proposed options for the CALL macro, viz.

CALL	ENTRY				
CALL	ENTRY	(@ARGI	LIST)		
CALL	ENTRY	(ARG1,	ARG2,	• • •	ARGn)

Similarly, there could be the same set of RETURN macros replacing CALL by RETURN, and ENTRY by the location specifying the amount of extra storage for stack expansion.

RETURN without argument would be assembled without stack expansion, as shown in the GAP paper.

RETURN	EXSTØR	expands
	EAPbp	sp†16,*
	ADBsp	EXSTØR
	STBsp	bp118
	LDB	bp†0
	LDR	sp†8
	RTD	sp120
RETURN	EXSTØR	(@ARGLIST) expands:
	EAPbp	sp [†] 16,*
	ADBsp	EXSTOR
	STBsp	Ър∱ 18
	EAPap	ARGLIST
	STPap	spĺ-2
	LDB	Ҍҏӏ҅҄Ѹ
	LDR	sp18
	RTD	sp120
RETURN	EXSTØR	(ARG1, ARG2,, ARGn) expands:
	EAPbp	spĺ16,*
	ADBsp	EXSTOR
	STBsp	bp ¹ 18
	EAPap	ARG1
	STPap	spÌ-2*n-4
	EAPap	ARG2
	STPap	sp↑-2*n-2

•••	
EAPap	ARGn
STPap	sp]-6
EAPap	FENCE
STPap	sp↑-4
EAPap	sp' -2*n-4
STPap	spî-2
LDB	ър↑О
LDR	sp18
RTD	sp 20

5.10 Thus, after a call to the SHELL, all possible results produced by the procedure called, are available in the stack. We have seen in the previous paragraph 5.9 how the SHELL can retrieve those unpredictable resulting arguments which might be returned, in addition to the normal ones. We assume now being at the supervisor level, after a procedure has been executed under the SHELL's control.

When the $BR \emptyset \emptyset M$ is on, any request may be typed from the console, while keeping all previous information gathered so far into the stack. By using some conventional notations, it is possible to reenter as arguments of a request some of the results left by the previous request.

Assuming for example, that a procedure segment, called CØMPUTE, prints, and returns, the result of an arithmetic expression given as argument, one might type:

	CØMPUTE	25 + 7 - 10 (SAVE)
which prints:	22		
followed by:			
	CØMPUTE	* 3 - 1	
which prints:	65		

The saving of the stack might be a systematic property of COMPUTE, rather than controlled by a meta-argument. It is a matter of preference.

In the previous example, we show that the name of the procedure may be assumed having the "value" of the preceding request. But, in general cases any of the previous arguments can be mentioned anywhere. For that purpose we use the conventional notation

to specify the nth previous argument. The value of the request itself is given the rank 0; the name of the request is given the rank L (for label).

i.e. RQUEST ALFA #3 #0

means: use previous third argument and previous value as second and third arguments

while $+ \neq L \neq 1 \neq 2$

means: execute the request set by the previous segment. (It may be the same, if the arguments have not been altered.)

Return arguments (if any) need not a special handling; they are considered as extra-arguments with a rank extending the list of the input arguments. Thus, if a procedure is called with three arguments, and returns two extra arguments, #1 # 2 and # 3 are the input arguments (possibly modified by execution of the procedure); # 4 and # 5 are the return arguments.

If an argument is mentioned and does not exist, the SHELL should send some diagnostic message, and take the error exit two, with the BROOM ON. Then, the user could retype a correct request without losing his status.

How this is going to work is quite straightforward. Upon entry from the supervisor, the SHELL notices by scanning, that previous arguments should be retrieved, so does it, by tracing back into the stack. If there are no previous arguments, (stack not saved) an error exit occurs.

N.B. The notations offered here are clearly arbitrary, and opened to any better suggestion. Furthermore, as we have said, there is still for a user who really does not like the system convention, the possibility of supplying his own SHELL.

VI. Arguments Management

6.1 Up to now we have considered only the possibility of putting in BCD arguments. This restriction of the present command system should not be carried through the system. In effect, strictly speaking, there will not be any more commands, but only procedure segments, made up from subroutines, and nothing else. The broad denomination of commands will likely remain however, but merely as applying to console oriented procedures rather than program oriented ones. In terms of programming and bringing into the system, this will not make any difference.

Henceforth, the scheme would not be complete if it did not allow the putting in and out of any kind of arguments, as subroutines generally deal with. The various types which one may encounter are:

- single word values (octal, floating, integers, BCD)
- double word values (double precision)
- arrays
- addresses and double word pointers
- procedure names
- BCD strings
- lists
- structured blocks of arbitrary data etc.

An evident conclusion is that it does not seem easy to close out a comprehensive list of all possible types of arguments. Consequently, we assume a reasonable compromise, whereby arguments will be handled directly up to a certain complexity; beyond, only pointers will be handled leaving to some specific routines the charge of appropriating themselves to the relevant data. For example, in the above list, the line could be between BCD strings and list structures. In other words, only linear structures could be handled directly.

6.2 Specifying arguments requires a somewhat more detailed description than a plain list of BCD words. Therefore, a special mode of formatted input is entered through a meta-argument: "(MANUAL)". Then the user may type his input data, in MAD-like messages. N.B. The MAD technique is only suggested as the more flexible in such a context. Evidently, the development of other languages and the GE $1/\emptyset$ package may bring about some modifications if they turn out more suitable.

Each formal argument is associated with the data block (single, double, multiple words) specified in the consecutive data specifications. The "*" marks the end of the input phase.

Two more data definitions are necessary, since data may already be present in the user's storage area.

A = SEGMENT segment names A = FILE file names

In case of a file, the SHELL enters automatically the common procedure provided in the system for generating segments. This does not apply when the file is to be handled explicitly as a file, via the regular I/O package.

Files, and segments, which are a specific class of files, are the only possible sources of data to feed procedures with elaborate structures as lists, trees, packed components, etc...

In order to specify names inside segments there are some modifiers recognizable by their particular pattern. These notations are similar to those proposed in paragraph 5.7, and intend to keep as close as possible to the GAP symbolism.

E.g. A = SEGMENT segment names $[Z\emptyset\emptysetM]+5$ means that the data of argument A starts at 5 locations after the symbol ZOOM defined inside the specified segment.

The mechanism used by the SHELL to get access to those data is almost identical to the one used for establishing calls to procedures (as seen in

5.7). The SHELL sets in the stack the followin vector

spÎA	SEGAD	sp [†] STP,F
	ARG	
	ARG	sp [↑] SSTP,F
	ARG	5
	• • •	
sp ¹ STP	STRING	m,segment names
spTSSTP	STRING	4, ZØØM

6.3 After execution of the requested procedure, the results, if any, are in the stack. Indeed, the MANUAL mode provides for a breakpoint in the SHELL before returning to the supervisor or the calling program, whatever. Then the user may give requests to print some results, or to store them away for later use. The requests are:

> PRINT, arguments PRINT OCTAL, arguments PRINT BCD, arguments PRINT FORMAT (valid format), arguments

followed by the formal argument names requested.

E.g. PRINT FØRMAT (306, F10.9), C, A, B * The "*" marks the last request.

Again there is a need for two more requests providing the means of equating files, or segments, to data blocks. They are:

SEGMENT segment names, arguments

FILE file names, arguments

And the SHELL enters the procedure of creating a segment or a file, out of the specified collection of arguments.

6.4 Another variation is proposed in the definition of arguments, when the execution may be repeated. Instead of (MANUAL) one may type (AUTØ) followed immediately by the name of a BCD file containing all the requests of the type mentioned above. The execution of the procedure may thus become entirely automatic. Furthermore, (MANUAL) encountered in the file, switches the mode back to the console, and (AUTØ) typed on the console switches back to the file. Intertwining preset arguments and console input is thus possible. If several requests attempt to define the same argument, the last one (dynamically) is the only valid one.

VII. Requests Stacking

7.1 The chaining of requests, similar to those typed at the console, is straightforward. Consecutive calls to the SHELL, from any procedure, and at any level of recursion, allows an unlimited chaining of requests.

7.2 Another feature commonly used on the present system is the execution of a stack of requests stored into a BCD file. This mode is a easy variation, as it consists in reading a block of several BCD request strings, and postpone the return to the calling program until the block has been exhausted. Due to the present system conventions, the SHELL selects this mode of execution when the name of the request is RUNCØM, while the first argument is the BCD name of the file. But any other convention may work as well.

As a matter of fact, the SHELL calls the procedure RUNCØM, which is responsible for macro-expansion and substitution of actual parameters. The RUNCØM turns control back to the SHELL with a data block of requests to execute in a row. This data block may be the pure contents of the BCD file if no macro processing is required.

7.3 Stacking requests typed from the console is also possible, as long as a character is recognized as separator between successive requests. The comma, for example, would mean the beginning of another request.

7.4 Whatever the various ways used to put in a list of requests, the SHELL manages eventually the storage of the list, extracts one set of arguments at a time for calling one procedure, gets control back and repeats the same step until the list of requests is exhausted.

7.5 Needless to say, any input device may be thought of a potential source for lists of requests: tapes, card readers, high speed lines. A typical batch process may be turned easily into a list of requests, which

are a much more elaborate form of "control cards." The fact of being able to invoke all the tools available in the system, (as opposed to the only features of a particular monitor) provides a very high flexibility for chaining and interspersing tasks, and relieves from the handcuffs of the linear stream of control cards and associated data.

7.6 The SHELL must not be seen as merely a console oriented tool, even though some parts of it are specifically console oriented. As a point of fact the SHELL is first of all a general procedure-linking tool, and as such fits equally the batch-type process requirements, where no console is attached during execution.

VIII. Meta-arguments

8.1 It is a commonly used technique to use <u>conventional words</u> having <u>no meaning as values</u>, <u>but only as modifiers</u> to the procedure to be executed. In the present system, each command may have its own conventions, as there is usually a complete independence between the various commands. By using the SHELL, it turns out that a set of meta-arguments can be defined as general conventions of the system. The SHELL strips off these arguments that it recognizes, so relieving each particular common procedure of the corresponding overhead.

8.2 We do not intend to give a complete and definite list, because the names are arbitrary, and their only justification is primarily a large acceptance by the users.

(MØRE) as last word means that the next line is a normal continuation of the BCD string of arguments. When a list of request are not given at the console level, (MØRE) may be also used as the first word of a request, meaning that it is merely a continuation of the previous request. Because this facility requires a look-ahead, it cannot be used at the console level for obvious reasons.

(SAVE) as any word means: keep stack for the next request, i.e. return with the BROOM ON. As first word is recognized by the supervisor for keeping the current stack.

(MANUAL) as any word means: enter the interactive process implied in the present context.

(AUTØ) enters the automatic process implied as the other alternative in that context. The first time the automatic process is entered, the file name (if any used) must follow immediately (AUTØ). If nothing follows the (AUT, argument, when no file has been yet specified, both (MANUAL) and (AUTØ) phases are considered through, and the process continues, if it can, or control is returned to the caller. This is the way to get out of a (MANUAL) mode entered by mistake.

(NIL) Stands for a void argument which cannot be omitted.

(BRIEF) Only emergency, or fatal error diagnostics messages are printed on the console.

(LOUD) Restores printing of all messages on the console. It is the responsibility of the program which creates a message to determine whether it is of emergency, or routine. When in BRIEF mode, routine messages are discarded.

(MESPOT) Appends all console messages to a file.

(MESOUT) Restores printing messages on the console. The name of the file may be set through a call to the supervisor, and is stored in a readonly part of the user's storage. The initial setting is :

MESPOT BCD

When in MESPOT mode, all messages are written into a file regardless of the BRIEF or LOUD mode, (which controls only the number of messages.)

(MESPIT) Reads all console messages from a file.

(MESIN) Restores reading messages from the console. The name of the file is set through a call to the supervisor, and is stored in a read-only portion of the user's storage area. The initial setting is:

MESPIT BCD

There is no provision for emergency input messages. This means that a process running in MESPIT and MESPOT modes will likely fail if it has to read in a message not supplied in the input file. On the other hand, the user has the possibility to switch to the MESOUT and MESIN modes for those parts of the process where some emergency action might be anticipated. Having

two classes of input messages acting simultaneously would require a tedious preparation of the job as to which message should go into the file, and which not, assuming in addition that a reliable documentation exists for that matter.

(LIST) controls a general option commonly used by assemblers, interpreters and compilers. Creates listings, whatever they are, according to the particular meaning of the running process.

(NOLIST) Suppress listings. Same context as above.

(DEBUG) Another option frequently available in compilers, assemblers, loaders, etc... Creates source information (symbol tables, patch areas, trapping instructions, tracing links, etc...) used by debugging tools.

(NOBUG) Cut-off the generation of the above debug information.

(QUOTE) Applies to the following argument, including (QUOTE), so that it will not be interpreted, but taken as a literal value.

N.B. With the exception of (MORE), (SAVE) and (QUOTE) all metaarguments can be placed anywhere in requests. Their position is meaningless; only their existence matters.

8.3 All the above meta-arguments will be identified by the SHELL, which strips them off the list of arguments, and sets accordingly the associated bits of the permanent options. (see paragraph 9 for a more complete discussion). As a matter of fact, the permanent options are not modified, in their read-only box. Only their carbon copy, used by the procedures in execution, is updated for the <u>temporary duration of the current procedure</u>. Meta-arguments are thus a way of overriding temporarily a permanent option for the current procedure (not the whole process), and all deeper levels of calls. Before returning control, any procedure restores to its initial value the carbon copy of the permanent options.

8.4 As a general policy, all system tools should keep the same rules as the SHELL does with respect to meta-arguments. This would bring about a clean consistency, and a logical behavior among all procedures.

IX. Permanent Options

9.1 There has always been the need for a permanent setting of user's options, not for the duration of a single procedure, but for an arbitrary long period of time. By convention, a certain amount of options are assigned a permanent physical representation in the machine, and eventually tested by procedures to fit the particular user's requirements.

9.2 A full machine word may be assigned to those permanent options common through the system. Another 36-bit word may contain user's private options.

An attempted list of the general options is following:

bit 1 -	Øff –	MANUAL mode
	ØN -	AUTO/mode
bit 2 -	ØFF -	Eull printing of all console messages
	ØN –	BRIEF mode. Sifts messages
bit 3 -	Øff –	Prints on console
•	ØN –	Writes all console messages onto a file
bit 4 -	Øff -	Reads from console
	ØN –	Reads all console messages from a file
bit 5 -	Øff -	Process serialization fields, if they may have to
	Øn –	Ignore serialization fields
bit 6 -	Øff –	No BCD listings from assemblers or compilers
	ØN -	Create BCD listings
bit 7 -	Øff –	No "debug" tables from assemblers or compilers
	Øn –	Create "debug" tables
bit 8 -	Øff -	All system library, or common procedure allowed
	ØN –	Not allowed
bit 9 -	Øff –	Normal housekeeping of files
	Øn –	Question on the console, whenever an attempt
		is made to change anything to a file

bit 10	-	ØFF	-	User is accessible by "mail"
		ØN	-	User is not accessible by "mail"
bit 11	-	ØFF	-	User does not accept inter-console messages
		ØN	-	User accepts inter-console messages
bit 12	-	Øff	-	User does not accept I/Ø slaving
	_	ØN		User is in I/\emptyset slaving mode.

This list is to be completed when more options become of general use.

9.3 All user permanent options can be stored into a read only segment, in the user's file directory. The structure would be as follows:

	SYMDEF	COMOPT, PRIVOPT, MESPIT, MESPOT, ERROR,
COMOPT	OCT	user common options
PRIVOPT	ÔCT	user private options
ESPIT	STRING	names of MESPOT file
MÉSPOT	STRING	names of MESPIT file
ERROR	STRING	names of ERROR segment

The number and the variety of permanent options is somewhat arbitrary and the problem is where to stop. Since the needs will grow with the users and system sophistication, the segment of permanent options should be organized in an open manner, with new classes of options assembled in whenever necessary. Each option, or family group is externally known by a symbol, therefore each user option segment may be organized freely, and contains as many private entries as desired which are used by the user's private procedures of course, but not by the system. However, in order to prevent the frequent mistakes resulting from too large a freedom, the OPTIONS segment may not be altered directly by the user. Use of common procedure is necessary to modify, delete, or append entries to this segment.

9.4 All system functions which may be under control of an option, such as console Input-Output, files protection, mail, console slaving, should systematically check the corresponding information in the user OPTIONS segment, during execution. This must be done at such a level that it cannot be by-passed by user's programs. For example, there should not be a direct entry to the supervisor allowing printing a console without checking the MESPOT bit. Evidently, there must be a possibility of by-passing, but restricted to certain authorized users, or certain privileged procedures. This policy is required for system consistency, since there is no guarantee that user's programs play the game, when there are ways not to do so.

9.5 In paragraph 8.3 we have discussed the effect of meta-arguments on permanent options. There is no need to repeat here what has been said on precedence and evanescence of meta-arguments. But we emphasize the point that <u>meta-arguments have no effect on the OPTIONS segment</u>. Whenever the supervisor initiates a process, it activates a segment called CUROPTIONS, which is also in read-only mode for safety consideration but <u>may be altered without intervention of the supervisor</u>. This segment is a mere copy of the OPTIONS segment, and is to die with the process it is associated with. All checking and updating of options is performed in the CUROPTIONS segment.

9.6 Modifications of permanent options, i.e. the segment ØPTIONS, require using a system procedure called OPTION. The particular parameter to be modified may be specified by name, plus a modifier ON or OFF, (nothing assumes ON), or a literal value, and also by bit position for octal words

E.g.	OPTION	MESPOT	OFF	same as	OPTION MESOUT
	OPTION	MESPIT	READ	FILE	(sets file name)
	OPTION	SYSTEM	MANUAL	NOBUG	LOUD
	OPTION	USER 2	5 ON		

Names like MANUAL, NOBUG, are associated with a definite setting of the option, (the bit may be set ON or OFF, depending on the name). But, a name followed by OFF (E.G. NOBUG OFF) means: set the option to the opposite of the specified name. With bit specification, set bit ON is assumed, unless explicitly indicated by ON or OFF.

9.7 There is a further need for having a set of permanent options associated with some common procedures, and variable from a procedure to another. This facility is included in the proposed scheme in the following way.

A procedure written for using a tailor-made set of options would access the CUROPTIONS segment through a particular entry name associated with itself, and replace some parts of the CUROPTIONS setting by the special status associated with the procedure. On return, the status in CUROPTIONS should be restored to its original setting.

This implies that a procedure which enjoys the privilege of having its private options has to set them for itself. One may easily imagine an automatic setting by the supervisor, but this looses generality and creates useless overhead. Indeed, the supervisor would have to initiate an attempt for every procedure, and most of them do not require this setting. Furthermore, any procedure may be executed with a tailor-made set of options, as we are going to see now.

The user may create a file containing all the particular options valid for a procedure only, specified as meta-arguments to the SHELL, then include the procedure with its formal arguments, and wrap up the whole thing as a macro-procedure

E.G.	This is	s the file	GAP BCD	
	CHAIN	ALFA		
	(BRIEF)	(DEBUG)	(NOLIST)	(MORE)
	GAP	ALFA		

Then RUNCOM GAP BETA performs the assembly with all particular options, without altering the constants permanent options.

X. Shell Organization

10.1 The SHELL should make use as much as possible of other standard common procedures, so that users will be able to alter its behavior in some particular areas without having to provide a complete SHELL of their own.

A list of separate tools is suggested in the following, not only because they fit logical divisions of the SHELL, but because they are to be used widely by other procedures throughout the system.

10.2 String manipulation package. As BELL LABS are likely to convert to GE636 their string macro-package, (used in SNOBOL for example) some cooperative task could be undertaken in that area. Another area for borrowing techniques is the SLIP system. Indeed, since naming in the system will be string oriented, there is an obvious and general need for such functions as:

- grab a string and move it
- count characters in a string
- concatenate strings
- break a string according to delimiters
- create list of strings
- traverse list of strings
- get Nth string of a list
- delete Nth string of a list
 - etc... etc... etc...

10.3 The SCANNER is a particular implementation using a string package. It primarily breaks a string into a list of strings, according to delimiters conventions, and yields values for string count, length, etc...

10.4 LINKAGE is a procedure which converts a composite address SEGMENT [ENTRY] GAPEXPRESSION, MODIFIER into a linkage vector.

10.5 ARGLIST makes up a list of pointers to a specified list of data arguments. It also sets up the fence (or the argument count) bound-ing the list.

10.6 A symbolic I/O procedure similar to the symbolic MAD I/O. On input, accepts strings of the form used by the READ DATA statement in MAD; On output, accepts strings as the MAD statements PRINT RESULTS. I/O and conversion are automatically performed using a table of pointers to the data and their symbolic names. Following a previous setting of external devices, input-output may be associated indifferently with any file, console or other.

XI. SHELL Flowcharts

11.1 A rough flowchart is outlined in the following. It ignores a certain amount of particular situations, which are important, but at the coding level. As a flowchart is usually a cryptic nonsense from every but one's point of view, some remarks will attempt to clarrify the major steps. Figures in parenthesis match the ones posted nearby the boxes.

11.2 The SHELL has two entries. One is for the processing of one or several requests given as a single string. The first thing to do (1) is to break it down into components. Another entry by-passes this first step when requests are supplied as a list of separate requests, each one containing separate strings as arguments. All requests are stacked by (2).

Then the SHELL starts looking for meta-arguments (3), taking appropriate action on the CUROPTIONS segment. The occurence of (MORE) results in saving the request (4) and going back for getting the rest of the request.

If in (MANUAL) mode, or (AUTO) with a file to be read, the SHELL constructs the table for the symbolic I/O procedure (5), sets the appropriate vectors (6) for the source of data, and (7) calls the I/O procedure, which replaces symbolic data by their values according to the specified conversion.

There may be arguments specified (8) as results of the previous procedure.

Then, everything being settled, linkages are established (9) to the specified procedure and to those arguments given by pointers to other segments. Eventually (10) the SHELL calls the procedure, and hopefully gets control back.

The output-phase takes place, if any. In effect, the procedure just called may have already put out desirable results. It is only on user's requirements, by (AUTO) or (MANUAL), that a specific result output phase is performed. Again, the table for symbolic I/O is built (11), unless this is already done. Then a proper setting takes place (12), in order to feed the sym bolic I/\emptyset package with requests from the appropriate source (console or file). Output requests are read (13), then decoded and processed (14). Recycling the loop occurs until the output phase is terminated.

Then the SHELL saves into the stack (15) all resulting arguments which were not in yet. Those segments (data or procedures) that are now useless, are released (16), and finally the current list of pending procedure requests is accessed for a new cycling.

When there are no more waiting requests, the SHELL saves any of its final status (18), restores the calling conditions, and returns to whichever called it. In case the last request specified (MORE) arguments, the SHELL expects to be called again with the rest of the request. Consequently, the BROOM is set on (17), so that the present status will not be wiped out if we were at the supervisor level.

SHELL Flowcharts



(Cont.)

SHELL Flowcharts (Cont. 1)



XII. Further Needs for Common Procedures

12.1 Formatted input. (See also design notebook, Appendix F, paragraph 2.5.1.) One must be able to create a data file containing any kind of preset data, i.e. integer or floating point numbers, octal fields, as well as BCD strings. This may be realized by setting a mode whereby input lines are interpreted according to conventinal notations similar to the READ DATA statement in MAD.

R-34

12.2 For the same purpose, data output, as PRINT commands, or PRINT off line request, should accept a format specification in order to get a readable hard copy.

Both of such implementations should use as hard core, the symbolic I/O package already mentioned in 10.6.

12.3 The concept of segmentation creates needs for some smooth transformation between files and segments. Indeed, data segments and data files are two forms of data storage, which users will likely use concurrently to feed procedures. Furthermore, segments as long as they are in the user's file directory, appear just like one more variety of files. Unless tools for handling files are general enough so that all segments peculiarities show like ordinary file parameters, some specific procedures would be necessary.

Hence we may forese the following needs:

FILE	segment names file names
SEGMENT	file names segment names
RENAME	segment names segment names
DELETE	segment names
CHMODE	segment names mode
SEGMAP	list of segments presently accessible
SEGNET	list of segments, with all cross references
	and internal symbols known betwen segments
SUBMAP	list of subroutines under procedure segments
SUBNET	cr oss-references between subroutines in segments.
EQUATE	segment names segment names

This list is likely to be extended after further study. There is evidently more to say about that subject.

12.4 Another set of procedures is associated with the permanent options. See description in 9.6.

XIII. Summing Up

The ideas outlined in this paper come out roughly as follows:

13.1 There is no particular convention distinguishing commands from any other private or common procedures.

13.2 There is a procedure, the SHELL, intended to control the overall execution of procedures. It has as properties:

.set arguments for procedures

.initiate procedures

.extract results

.act as a link between procedures not expressly linked by program

.recursive, may be called by any program, at any level

.is automatically called by the supervisor for requests typed at the console level

.partially or entirely interchangeable with user's private tools, so that the system language may be completely remodeled by any user.

13.3 Private permanent user's options control the behavior of the system in all critical areas, particularly with regard to input-output modes and peripheral! devices used. Thus, any procedure may be run indifferently without regeneration under a batch process, or a console controlled job session.