Replacement for B 1.

BELL TELEPHONE LABORATORIES
INCORPORATED

SUBJECT: A Survey of the Software for the GE636 DATE: April 15, 1965

FROM: P. G. Neumann
V. A. Vyssotsky

Table of Contents

1. Introduction

The present document is intended for limited circulation as a survey of the present state of the software effort for the dual processor GE636 (augmented 635) system. (For references on the 636 see the 635 Programmer's Manual and the description of the augmentation hardware, software committee document 71 by J. F. Couleur.) We hope that in addition to provoking comments, this document will inspire the writing of those documents which are herein referred to as being forthcoming. It will then be revised and reissued together with the inspired documents as a detailed treatise on the foundations of the software effort.

The present document is divided into three parts. The first deals with the central software strategy and the use of the facilities available. The second considers the translators and utility packages. The third considers problems in the use of the system. Throughout the entire document, the aims of reliability, efficiency, flexibility, modularity and understandability recur. Standards of programming and documentation satisfying BTL, MIT and GE will be established and enforced.

The basic system outlined here uses many notions of the MIT Compatible Time Sharing System (see for example MIT document MAC-TR-16 by J. H. Saltzer), and of the relocatable system for the 7090 evolved at BTL.

2. Central Software Strategy in the Time-Shared Multiprogramming Environment

2.1 The Environment

The software system will provide for a wide variety of users, with varying requirements for turn-around time, desired completion time, storage usage, and input-output. There are two basic types of users -- over-the-counter users similar to those in the present batch environment, and on-line interactive users, initiating and interacting with jobs from terminals such as typewriters and teletypewriters, scopes with light pens, acoustical digitizers, subcomputers (such as the PB250 and PDP5 in the present shop), or processing in real time data received from remote terminals such as recording or measuring apparatus. A fundamental hypothesis of the system is that the two types of users be as indistinguishable as possible within the machine. For example, it is highly desirable, although not quite achievable, that all language facilities of the system be available from typewriters.

To avoid inefficient use of the central processors, any requirement for storage, input-output, etc., which cannot be immediately fulfilled by some part of the processor should cause the transfer of control of that program to another program which at that moment appears to be ready to run. This is the notion of multiprogramming.

Any one interactive terminal will normally require computing activity a small fraction of the time. That is, most of the time a program responding to a particular terminal will be waiting for information from the terminal about what must be done next. By taking advantage of this fact in a multiprogrammed system, many terminals can be simultaneously attached to the computer, and their demands for service fulfilled on request. This technique is known as time sharing.

The typewriter is the simplest computer terminal capable of general purpose interaction. It is expected that time-shared computer access from typewriters will become a common, if not the predominant, way of using computers in BTL. It is expected further that the resulting new style of computing will noticeably advance over-all BTL productivity.

For a detailed discussion on using typewriters as remote terminals see "The Characteristics, Behavior, Attachment, and Use of Typewriters as Remote Computer Terminals," by J. F. Ossanna (a forthcoming software committee document), and "Preliminary Estimates of Quantities and Traffic Statistics for Typewriter Remote Terminals for Bell Telephone Laboratories," by J. F. Ossanna (software committee document 73), plus addendum.(document 80).

2.2 Memory Considerations

In this section the problems of storage organization, storage management, and storage allocation are considered. Detailed documents on these subjects are forthcoming.

2.2.1 Storage Organization

Programmers are accustomed to thinking about two dissimilar forms of storage. One is primary storage, where each word is directly addressable, and thought of as being obtainable without delay. The other is secondary storage, where data is organized in files and records rather than words, and where the access time to an item is expected to be much greater than one instruction time.

These concepts will not continue to be used by programmers on the 636, despite the fact that the 636 does indeed have primary and secondary storage with these properties. Sometimes data which the programmer thinks is in primary storage will actually be in secondary, and vice versa. Broadly speaking, words are words and files are files regardless of where they happen to be.

On the 636 in a time-shared multiprogrammed environment there may be dozens, or even conceivably hundreds, of programs which have started execution and which have not yet completed. This is because a program which requires input from a remote terminal cannot proceed until the person or the equipment at the terminal gets around to supplying the required input data. Not all of these programs with their data can be simultaneously held in core. Therefore, at any given moment, some of the programs, or parts of some of the programs, will be on drum (4,000,000 words maximum capacity) or on disk (33,000,000 words per unit, two units planned).

A program which receives input, processes it, and then hangs up waiting for more input, will very typically use only a small part of its allocated procedure and data storage in doing so. Hence it is very inefficient to swap all the program and data in and out of core for each such occasion. However, it is virtually impossible for a programmer to know in detail before a run which program paths and data words will be needed for processing the various input items. So the selection of what is to go in and out of core, and when, is best done dynamically, during execution. This selection could in principle be programmed by each user as a part of his program. In practice, most users won't do it, because the effort required is so great. They quite properly insist on thinking of all the program and all the data arrays as being addressable at once. So the dynamic selection must be done by the operating system. On the 636, the operating system, using the segmentation and paging hardware, will place in core those parts of programs and data arrays which are needed at the moment, retaining the rest on secondary. To the user, his program and data are logically addressable, although in terms of physical hardware that program and data may be spread across core, drum, disk, and conceivably even data cell and tape, if some of his information has been unused for a very long time.

Clearly, with such a scheme in operation, a block of information which is not referenced will tend to slide down the hierarchy of core, drum, disk, data cell and tape. The time

required to recover the information thus increases with lack of use. There will be certain types of information which cannot be permitted to regress like this; hence facilities will be available in the software for guaranteeing that programs with special requirements do not get pushed back in secondary.

Just as programs and arrays may reside in media other than core, so data files, or parts of data files, may reside temporarily on media other than their nominal ones. Indeed one important type of file is that whose medium is specified only as "secondary," without any further limitation. This will be the default attribute for files not otherwise declared, and the file will be kept on whatever medium is dicated by efficiency of use. It is essential that there be widespread facilities for self-identification of data structures, and that the use of these facilities be strongly urged. This greatly aids the debugging effort.

For reasons of memory hygiene, some effort must be made to keep useless information from remaining in the memory hierarchy forever. For ease of garbage collection, the user should specify the nature of a file when he creates it -- whether it may be deleted when first read (temporary), deleted by explicit command, kept forever, and so on. It is felt that the burden of deleting files is a responsibility of the user, and that the user should be charged for all storage usage, whether core, disk, tape, data

cell, or whatever. Although every effort should be made to induce the user to release useless files, the system must expect to have files which must be pushed down indefinitely in the hierarchy without ever being recovered, as indeed we now keep files indefinitely without ever using them.

Access to some of the data in storage must be restricted. Not all files may be rewritten by all users. Thus, provision must be made in the software to protect information against unauthorized reading, execution, overwriting and deletion. Proposals for file structures incorporating flexible protection mechanisms are contained in a forthcoming software committee document by R. Morris and in MIT CC-241 by R. C. Daley. The augmentation hardware has features which will aid implementation of such proposals.

Despite the need for privacy, ease of use is a paramount consideration. Passwords (programmed combination locks) should be optional on access to files, so that the user who wishes safeguards can obtain them easily, but the user who does not wish any is not encumbered. File naming should be simple, and both names and passwords should be chosen by the user, not assigned by the system. The file routines can inform him if he has chosen a name already in his file directory and ask if he wishes to delete the previous file with that name. (We must assume that for any files whose contents and names can be modified by two or more users, the relevant users will cooperate with one another.

No automatic features can prevent chaos if the users authorized to modify the file do not cooperate.)

Closely related to the problem of guaranteeing privacy and freedom from tampering is the problem of guaranteeing immunity to inadvertent clobbering of files due to hardware and/or software errors in the system. Core failures, for example, can be harmful if the only map of the contents of disk is in core. Thus it becomes mandatory for disk to contain a file directory of its contents, although this probably does not need to be a complete map. It obviously must be updated frequently, but probably not every time the core map is altered.

The notion of common files leads to the need for interlocks on common data. For example, user 1 takes file A and begins to modify it. He sometimes wants to be able to assure that user 2 cannot access A until he has completed his modification, and does so by making it a nonread, nonwrite file for the time being. (He is then the only user who may change the mode of privacy.) Suppose user 2 is in the process of modifying file B (with exclusive access), but cannot complete the process until he has access to file A. Since the hardware provides no help here, there must be software interlocks or bypasses to avoid the freezeout that ensues when user 1 must access file B before he can finish. There are presently various proposals for surmounting this problem.

2.2.2 Storage Allocation

The need for dynamic storage allocation arises when blocks of storage are required whose sizes or very existence cannot be determined until run time. In such cases it is often not feasible to allocate a maximum expected block size to each such block, simply because the sum of the maxima is too large. In the 636 system, dynamic storage allocation is greatly facilitated by the segmentation hardware (see software committee document 71). For each such variable size file of data, the user may assign a segment which may be expanded or contracted as desired, during the execution of the program.

All data addresses (and instruction addresses - see below) are relative to the beginning of the segment in which they occur. The relocation is accomplished with the aid of a segment description which contains the address at which the beginning of the segment is located, and the size of the segment. It is easy to change the size of any segment without having to change any other segment.

2.2.3 Storage Management

In order to obtain an efficient multiprogramming environment, it is necessary to keep to a minimum the amount of time spent swapping information in and out of core. The crucial questions here are how and when does information get swapped, what should be kept in core, and how is it found when it is needed. In the 636, paging provides considerable help

in answering these questions. Swapping in and out of core is performed in blocks of 64 or 1024 words (pages), and the location of the page is kept in a page table. Addressing is done relative to the page table, pages being located by the segmentation hardware. Thus it is not necessary to put a program in consecutive locations in core; instead, pages of the program may be splattered around in whatever space is available. (The saving in relocation time required to arrange consecutive core locations is of course significant, especially in a highly interactive environment.)

If a page is required which is not in core, the thread becomes dormant while that page is brought into core. At this point it may become necessary to swap another page out. Here page table usage statistics must be employed to determine whom to throw out. One useful tool for this is the use bit in each page table entry, which is set to 1 by the hardware if the page is accessed through this page table entry and if the bit is 0. By software convention, no page will be listed in more than one page table entry, so every access to a given page will set the same use bit.

If a page has been swapped in, but has not yet been used, it is almost always desirable not to swap it out, for its use was demanded by a thread which became dormant because that page was missing. (The exception involves the user who happens to quit at precisely this point.) Pages which have been accessed,

however, are fair game for swapping. Some sort of a software priority scheme is required for these, in order to give infrequently used pages less chance of avoiding swapping. Proposed here is some sort of weighting scheme such as

$$\text{weight} = (\text{previous weight})(\text{constant}) + (\text{use bit} - \tfrac{1}{2})$$
$$\text{initial weight} = 0$$

evaluated for each page whenever a swap out becomes necessary (with all use bits being reset to zero at that time); weights are stored in the core map, with swapping resulting for the page(s) with the lowest weight. (The hardware sets a use bit to one when its page is accessed.) An appropriate value of the constant appears to be around .9 (weight range ±5) or .95 (weight range ±10).

2.2.4 Reentrance and Pure Procedure

In order to avoid multiple copies of a frequently used routine (e.g. input-output), the 636 segmentation philosophy permits a routine to be reentrant, i.e., many users may be in various stages of completion of the same copy of the routine at the same time. In order to accomplish this elegantly, a single copy of the procedure part of the program is made available, and may not be altered (pure procedure). All variable information is found in a second (data) segment, of which each user has his own copy. In this way no interlocks are needed on the reentrant routine, and as many users may have access to the routine as require it.

A routine may be divided into a pure procedure part and a data part for various reasons besides a desire to use the routine in reentrant fashion. Debugging is easier if no changes occur in the procedure part during execution; reinitialization is easier, recursive subroutines are conveniently written this way. For these reasons among others, hardware facilities are provided in the 636 for making a segment read-only, and the 636 software will be designed to permit programs to be written easily in this manner. However, there are many cases when it is clearly desirable to intermix the constant and variable parts of a procedure, so no restrictions will be imposed to impede the writing of impure procedure.

The standard 636 CALL, SAVE and RETURN macros will use a stack (push-down store) for saving contents of registers. Any routine will be able to use this stack for temporary data storage, so recursive programs will be easy to write. The user will be able to ignore the stack completely if he wishes, even to the point of saving registers elsewhere (or not at all) by redefining the CALL, SAVE and RESTORE macros. He cannot discard the stack, however, since it will be used by any system routines he calls. The contents of the stack should be self-identifying for debugging purposes. Users who abide by system conventions will have the full benefit of system debugging routines.

2.3 Time Considerations - Scheduling

There are many general decisions which can greatly influence the time response of a time-shared multiprogramming computer system to its user needs. For example, what types of service are to be offered and to what degree? In addition to normal batch processing, there will be batch-processing facilities with short turn-around times and with guaranteed delivery, which may have to be negotiated for in advance - hopefully by dialog directly with the computer from a console rather than through a clerk or operator. (In any event, both of these human functions should be eliminated from the batch processing as much as possible. The current necessity of mounting tapes will hopefully be considerably relieved by the increased size of storage.) Batch processing will differ from its present form, however, in that jobs will be stored on secondary (disk), and taken as desired.

In addition to normal take-your-chances interaction facilities for a variety of consoles, there will also be special interaction facilities including guaranteed response time and guaranteed percentages of steady-state central processor time, again by advance negotiation. The question of how many interactive users to allow at any time, or how to draw the line on the basis of load rather than numbers of users can greatly influence the time response.

The implementation of the various commitments outlined above is the task of the scheduler, which will be described in detail in a subsequent document. The user should be reminded that the time-shared multiprogrammed environment cannot produce essentially zero turn-around time for all users. It is the task of the scheduler to attempt to satisfy the varied user needs as best as possible. Thus it becomes highly desirable that each user specify in advance of his run (possibly by default) the type of service he desires (e.g., turn-around time, guaranteed delivery, special memory or input-output requirements), along with time, page, storage and possibly other bounds (see software committee document 56 by W. S. Brown). In the event his running time exceeds the time bound, or his output exceeds the page bound, etc., he is so informed, and his run is halted, pending instructions.

In addition to priorities engendered by user desires and negotiations with the scheduler, there are various high priorities required by the system. Highest among these is the operator's absolute interrupt, which must override all overrides. Here the override is not so much instantaneous but rather decisive. Input-output routines, on the other hand, must have a high priority in order to keep the (expensive) input and output devices working constantly. Other examples are charging routines and disk dump facilities, both with guaranteed completion requirements.

2.4 Connective Tissue

In addition to specifications for the identifiable program modules of which the software is composed, design of the 636 must specify the way in which these various programs interconnect and hang together. This connective tissue consists partly of programming conventions - for subroutine calls, for use of the 636 augmentation hardware, methods of storing and accessing temporary data, and so on (see software committee document 68 by R. M. Graham for details). In part it will consist of code, as for example the instructions which reside in the fault vector. In part it consists of doctrines, such as an arbitrary assertion that the amount of master mode program will be reduced to the smallest possible amount.

It is essential that the 636 software should be easy to build, debug and modify, and that it be free of errors. It should be possible to partition or reconfigure the (hardware and/or software) system on the fly. Further, the system should have the highest efficiency and the shortest reflex time (time to react to urgent external signals) consistent with the other objectives. The most important guideline to follow in achieving these goals, of course, is careful advance planning and documentation. Other than this it is essential that the software be modular and systematic. The impact of this on the hard core software is that the hard core software should deviate as little as possible from the standard conventions for user programs.

Thus, the amount of code which runs in absolute mode or master mode should be minimized, and insofar as possible all programs should appear to be a part of some user run or runs. It turns out that this can be accomplished to a surprisingly high degree; a more extended outline of how to do this is contained in a forthcoming software committee document.

3. Languages and Language Processors

3.1 Uniform Input Syntax and Uniform Input Processors

In a computing system with many types of input terminals and many different language translators, it is highly desirable to establish certain conventions on syntax of input text, to give users of the system a chance of remembering how to talk to it. These conventions will be called a "uniform input syntax." Furthermore, a program (e.g., a language translator) which may receive input from a variety of terminals cannot be expected to cope with all the eccentricities of all types of input devices. Hence it is necessary to have a "unform input processor" to take input text from the various devices and render the text into a standard form for use by language processors.

The most basic aspect of input syntax is the character set. Different devices permit character sets of different sizes, and of course the choice of graphics is essentially arbitrary. The basic character set for input to the new software will be the 64 character ASCII set, with what are

essentially the ASCII graphics. (For proposals, see software committee document 50, by M. D. McIlroy. See also MIT Computation Center document CC250 by F. J. Corbato.) This will be a standard in the following sense: a language translator or other input program may assume that all input devices are capable of generating any character from this set, and need not be usable with a more restricted character set. Requirements for availability of characters other than these 64 will be considered as limitations on the applicability of the program, to be explicitly stated in documentation. This standard, of course, implies that our current key punches must be replaced by 64 character key punches. Standard internal character representations for the 64 characters will be those specified by GE for the 9 bit character mode. Six bit characters will not normally be used internally.

Extended character sets (more than 64 characters) will be available on many input and output devices; eventually a larger character set may be available on all I-O devices. No standard set of larger size will be specified at present. However, to allow for future extensions, all generally used programs will use 9 bit characters, and all programs must assume that any combination of bits can occur in a 9 bit input character. The treatment of input characters of unspecified meaning will be decided on the basis of context.

Another aspect of standard input syntax for language translators is concerned with the conventions for indication of comments, literals, sentence boundaries, labels, field boundaries and so on. It is important for tutorial purposes that an attempt at unification be made in this area; the standard guidelines are being developed. Current proposals in this area are contained in software committee documents 33 by R. Morris, and 39 by N. M. Haller.

The objective of a uniform input syntax is to ease the learning and memory task of all programmers. The objective of a uniform input processor is to ease the efforts of compiler writers. Slightly differing proposals for a uniform input processor are contained in software document 34 by R. Morris, and document 39 by N. M. Haller. A resolution of the differences is now in progress, and the resulting specification will be implemented.

In addition to the forms of input provided by the unit record input and the uniform input processor, there will be available to the user program a basic input mechanism which permits him to examine all the input information received by the computer, without alteration. This form of input permits the user program to interpret the input stream in any arbitrary fashion; it is necessarily highly device-dependent. For example, teletype input can be passed to the user program character by character, as it is received; this is not possible with input from magnetic tape.

3.2 The Assembler, Binder and Linker

In the 7000 series machines a program in some source language is prepared for execution in two steps. First it is translated into an intermediate language format, and then it is loaded and linked to other programs. This two stage process was devised to satisfy requirements of convenience and efficiency. For the same reasons of convenience and efficiency, the corresponding process on the 636 will be broken into three stages rather than two. These are translation, binding and linking. The advent of automatic relocation and segmentation hardware makes it possible to form a final version of the program text without knowing where in core the program will reside. The desire for linking during execution, coupled with the requirement of reasonable efficiency, makes it imperative that linking should be performed separately from the other tasks of instantiating a program. On the other hand, convenience and efficiency require that programs be translatable in pieces smaller than the units handled by the linker. Hence, in the 636, the translation process will usually be followed by the binding of translator output texts into segments; the bound text normally will be in its final executable form. Then, as execution proceeds, the linker will be invoked at appropriate times to establish intersegment references.

3.2.1 The Assembler

The basic function of an assembler is to provide a convenient flexible means of specifying machine language code. For this purpose a one-to-one, field-by-field translation of symbolic to binary is insufficient, because the amount of repetition required would be unduly high. Hence, a facility for handling macros and text strings is required. It turns out that much of the apparatus for pseudo-operations and generation of output can be subsumed within an appropriate macro and string facility, and this, of course, increases the flexibility of the assembler while decreasing the effort required to write and maintain it.

Another function of the assembler as used at BTL is the generation of code for machines other than the computation center computer. This can be done, if the assembler is properly constructed, by substituting new tables and subroutines into the assembler in place of the standard ones. In order that these substitutions be possible, the assembler must be designed so that the relevant tables, subroutines and decision points reside in clearly identifiable places, and are not dispersed throughout the entire structure of the assembler.

Thus, the assembler will consist of a skeleton padded out by appropriate subroutines and tables, which

accepts input text and puts out text for use by the binder. For further details, see software committee documents 29 by D. J. Farber, 37 by R. E. Archer, 33 by N. M. Haller, 48 by D. E. Eastwood and 64 by J. P. Hyde.

3.2.2 The Interface Between Assembler and Binder

The text which passes between assembler and binder carries a variety of different kinds of information. Among these, for instance, are text strings in final form (e.g., an assembled constant or op code), text strings requiring alteration (e.g., a relocatable address) together with a specification of what alteration is to be made (relocation bits), external symbols whose equivalences must be supplied at binding time, internal symbols with equivalences to be supplied to other routines at binding time, storage allocation information (e.g., loading origins), and other miscellaneous information. The format to be used as interface between translators and binder has not yet been chosen, but some of the desiderata are as follows:

(1) Tables, or isolated table entries, should be explicitly identified by header information; e.g., relocation information should be preceded by a header indicating that relocation information follows, and how much of it there is.

(2) The structure is to be open-ended, in the sense that header formats should allow later inclusion of an arbitrarily large number of new types of tables.

(3) Check features (e.g., check sums, sequence numbers) will not be included as part of the format, but will be inserted, verified and deleted by I-O routines as may be appropriate to the various recording media.

(4) The interface text will not be constrained to be intelligible to humans.

(5) The interface text will be formatted so that binder output text is acceptable as binder input text, in order that a segment may be built up in multiple passes through the binder. Text strings whose final form in core can be completely determined at translation time should appear in essentially that form as input to the binder.

3.2.3 The Binder

The job of the binder is to accept one or more modules of text produced by translators (or by previous passes through the binder) and combine them into a single larger module, which will typically be a segment plus its linkage information. The binder also provides the interface between source languages and the system for generation of debugging dictionaries, data structure descriptions, etc.

The two major tasks performed by our current loader are establishment of inter-module references and relocation of addresses. These will also be done by the binder; however, inter-module references will be direct (as in a linking loader)

rather than via transfer vectors. This will permit assembly of programs in modules which do not have to be complete subroutines, and will also allow source language layout of data storage to be compiled independently of the procedures which reference the data.

The binder will have a new, and initially quite limited, facility for expanding macros at binding time. In the initial version such expansions must obey the restriction that the length of text to be generated at binding time be known at translation time. It is possible that this restriction may be removed at some later date.

The binder may also include, either initially or at some later date, a facility for accepting assembly language source text, shipping it to the assembler, and then binding the resulting module with other modules. Whether this facility is included will depend primarily on how easy it is to implement.

3.2.4 The Linker

A procedure segment produced by the binder is ready for execution. Inter-segment references have not been established, and will not normally be established before execution begins. Instead, the linkage information of the bound segment will be placed in a linkage segment, and will remain unlinked until an attempted inter-segment reference causes a fault. (See software committee document 68 for details of linkage segment format.) When the fault occurs, the linker

will be invoked, will retrieve the referenced segment if it is not already available, will implant the linkage information of the referenced segment into the linkage segment, will link the desired intersegment reference, and then return control to the program which caused the fault. If desired, it is also possible to link at one time all references in a given segment to a particular outside point.

In addition to this automatic mechanism a number of explicit calls will invoke the linker. One of these will be a call which requests "Link such-and-such a reference, and do it now." This will cause the same linking action as would a fault, but under explicit program control. This call has various uses, and is essential in some cases as a substitute for the automatic linking. Another call which involves a part of the linking mechanism is a request for a new data block in an existing segment. This requires establishing or changing an intersegment pointer. This type of call will be provided primarily as an aid to handling the NPL ALLOCATE statement.

Since the input to the linker will be in a form which could also serve as input to the binder, a segment being instantiated by the linker will be accompanied by much information that the linker need not make use of. To what extent this information should be discarded and to what extent it should be put in some convenient spot for the use of debugging

aids is as yet unclear. However, this extra information must be retained somewhere to permit unlinking and subsequent binding or relinking of segments already linked.

3.3 Bootstrap GEM

Because the new assembler and binder cannot reasonably be made available before late 1965, it is essential to have some preliminary mechanism which will generate code for the 636. This will be a combined assembler-binder, currently known as bootstrap GEM. Bootstrap GEM will be a slightly revised GEM assembler with a loader grafted onto it, to produce 636 code ready for linking. Because the binder is included in the assembler, bootstrap GEM will necessarily produce one and only one segment per source deck.

Bootstrap GEM is being produced by GE, and should be completed in May or June of 1965.

3.4 Conventional Narrative Algebraic Languages

Among the languages available on the 636 will be Fortran IV, COBOL, and NPL (or MPPL, or whatever it is called).

COBOL will be available only through the GECOS submonitor, and will conform to the specifications in the GE 635 COBOL Reference Manual CPB-1007. It will be implemented by GE.

Fortran IV will be available through the GECOS submonitor, and may also be accessible directly from the main

operating system; GE is now considering whether this is feasible. GE will implement Fortran IV as described in the GE 635 Fortran IV reference manual CPB-1006.

An initial version of NPL will be implemented for BTL by Digitek, for April 1966 delivery. It will follow the IBM NPL Technical Report 320-0908 except for certain IBM modifications not included in the report, minor deletions by BTL (e.g., sterling currency arithmetic), and certain BTL extensions. For details on the 636 NPL and its differences from the IBM technical report, see software committee documents 65, by Digitek, and 66 by M. D. McIlroy. A second version of NPL will be delivered by Digitek about six months after the first version. It is to be expected that further versions of NPL will be required, since NPL is a novel language in some respects, so that both extensions to the language and improvements in compiling techniques are likely to be forthcoming.

In addition to COBOL, Fortran IV and NPL, a MAD compiler and an ALGOL compiler will probably be available for the 636, implemented by MIT and GE. No details are yet available, and there is little reason to believe that these languages will be widely used at BTL.

Fortran II will not be available on the 636 in any form. The extreme difficulty of implementing a Fortran II compatible with our current compiler, together with the virtual coincidence of function of Fortran II and Fortran IV, preclude

the existence of Fortran II. Some program will be provided to aid conversion of Fortran II programs to Fortran IV.

3.4.1 Incompatibility of Algebraic Languages

It appears highly probable that no procedure written in Fortran IV, COBOL, NPL, MAD or ALGOL will be able to call directly any procedure written in another of these languages. This state of affairs appears to be largely inherent in the detailed structure of the source languages themselves, which require different mechanisms and different kinds of information for transmission of subroutine arguments.

It is possible that interface subroutines can be provided to allow argument transmission in many cases; the most hopeful being a call from a Fortran IV program to an NPL subprogram. This technique and others will be made available wherever reasonably possible, but details will not be available until late 1965.

3.4.2 NPL Subroutines

In order for the full NPL Language to be available, a large number of subroutines will have to be provided for use by NPL object programs. These include various numerical packages to perform arithmetic of various types and precisions, a large package of type conversion routines, dynamic storage allocation, diagnostic and debugging routines, stack usage and IO routines. These routines will be so written and embedded in the software as to make them available also to programs

written in assebly language. Arguments and argument transmission techniques will be specified by Digitek in the course of NPL development. Further, mechanisms comparable to the asynchronous facilities of NPL will be available for use by programs written in assembly language.

3.5 Microfilm and Graphic Display Programs

Because of the diversity of visual display devices which may eventually be attached to the 636, and because of the transmission and buffering requirements of visual display devices, it is desirable that the format of data for visual display should be standardized in a form that is compact and convenient from the viewpoint of a programmer oriented toward display hardware.

However, for the user who doesn't care about the intricacies of display hardware, and just wants pictures, a hardware-oriented language format is inappropriate. For the user there must be a set of primitives available in Fortran, NPL and assembly language, analogous to our current system plot routines. The detailed form which such facilities will have is not yet clear; one proposal is that of F. W. Sinden in a subsequent software committee document. There are no current plans to provide a compatible replica of the microfilm routines in BE SYS7.

The use of such primitives is unduly laborious for many standard applications, and it will be necessary to

provide packages of standard plot routines to do, for instance, automatic scaling, labeling and plotting of one dimensional arrays. This job, however, is of comparatively low priority, and will probably not be started until much other software effort is complete.

3.6 GECOS (Comprehensive Operating Supervisor)

GE will provide for the 636 a version of GECOS III which runs the 636 as if it were a 635. This monitor is essential as a fail-safe against software schedule delays in checking out the new 636 software, and particularly in efforts to determine whether a particular trouble is due to hardware or software. This stand-alone GECOS, however, will not be useful after cutover to new software. At that time, facilities compatible with GECOS will have to be provided within the framework of the new software.

Two different proposals have been contemplated for doing this. The first envisions an "encapsulated" GECOS, which is essentially GECOS III with almost no recoding, subordinated to the new monitor by forcing GECOS to run in slave mode as a single segment throughout an entire batch of user programs. GECOS master mode functions would be performed by the new monitor as it catches faults caused by GECOS' attempts to issue master mode instructions. This alternative has the advantage of being simple to implement, easy to debug, completely compatible with GECOS III for the 635, and easy to update to maintain compatibility with 635 GECOS.

The second proposal envisions an "articulated" GECOS, partly rewritten to take advantage of the segmentation features of the 636, and the file handling strategies of the new software. This would permit GECOS to be scheduled more flexibly, and GECOS storage to be allocated more flexibly, than would the encapsulated version. The articulated version would also permit users of GECOS to take advantage of certain of the new software features. For example, GECOS runs could be started from typewriter consoles; data could be passed to GECOS runs from files constructed under the new monitor facilities; some subroutine packages constructed for the new software could be used with GECOS.

Compatibility of 635 GECOS with articulated GECOS will be harder to achieve and maintain than with encapsulated GECOS. However, since GE proposes to do the implementation and maintenance, and since GE favors the articulated version, it is advantageous to proceed with the articulated GECOS. GECOS users, however, should not assume that any particular new feature will be available in 636 GECOS until that feature is specifically guaranteed.

3.7 Special Languages

The 636 software will include SNOBOL, ALPAK, Simscript and BLODI. For none of these is a detailed specification yet available. Simscript will be essentially

the same as the version now in use on the 7094. BLODI will be basically the version recently completed by B. J. Karafin for the 7094 (MM-65-1359-2). ALPAK will be ALPAK B or some extended version thereof. SNOBOL will initially be an adaptation of SNOBOL 3, and will subsequently be extended.

Of these translators, the one of highest urgency is SNOBOL, which is needed at initial installation date. Simscript will also be available at initial installation, because there will by then be a version of Simscript supplied by GE and running under GECOS.

3.8 Command Language

The command (control card) language of the new software will be a narrative language without looping capabilities, i.e., it will be similar to the control card language of BE SYS7 and to the command language of CTSS (MIT's Compatible Time-Sharing System); it will not assume (as does GECOS, for instance) that all control cards for a run will be read and interpreted before execution begins.

Command format will conform to the uniform input syntax (i.e., free fields), and therefore the line layout will look much like the current layout of CTSS commands. In general, the command language can be expected to include the sorts of commands currently available in CTSS, plus other commands for handling files (e.g., rewind), and a number of

miscellaneous commands (e.g., unload). There will also be various declarations, including file declarations like the DISC and TAPE declarations of GECOS, and various control declarations (e.g., a declaration like the LIMITS declaration of GECOS). More details will be available in May 1965. A detailed discussion of a framework for command language implementation is contained in software committee dbcument 77 (the SHELL: A Global Tool for Calling and Chaining Procedures in the System, by L. Pouzin, MIT Design Notebook, Section IV).

3.9 Utility Packages

3.9.1 Elementary Functions

The elementary function routines now incorporated in GECOS may or may not use satisfactory algorithms. This is yet to be determined. However, they must be modified for the new software in any event, to conform to calling sequence modifications dicatated by 636 hardware. In addition, since NPL and Fortran will use different conventions for argument transmission, some or all of the elementary function routines will have to exist in two versions, one for Fortran and one for NPL. Moreover, routines for multiple precision arithmetic and for multiple precision elementary functions will be required for NPL.

Since the number of routines required for elementary functions will thus be quite large, and since the speed of the routines is important, they cannot be made to work correctly

by conventional techniques of hand coding and manual debugging. Hence, it is very desirable to have generators for elementary function routines, and semi-automatic testing procedures for the resulting programs. Whether adequate generators can be developed on a time scale consistent with hardware development is doubtful. Nonetheless, the development of generators will be pursued, since a good generator will be very useful even if it is completed somewhat late.

3.9.2 Type Conversion and Multiple Precision

For NPL it is necessary to have an extensive package of routines for type conversion (e.g., BDC and DBC) and for multiple precision arithmetic. These routines will have to be written from scratch; whether they will be done by Digitek or by BTL is not yet decided.

It is possible that these routines will also be usable for Fortran I-O conversion in place of the package currently used by GE 635 Fortran. If the Fortran I-O package can be thus eliminated, system maintenance will be somewhat simplified.

3.9.3 Other Numerical Routines

Depak (differential equations) will be running on the 635 under GECOS by the time of 636 installation. Because of this, it is not essential to subordinate Depak directly to the new monitor at first, though it will be done eventually. A

matrix package must be provided, including eigenvalue and eigenvector and matrix inversion routines. These routines must be constructed and tested with great care, and BTL will have to devote substantial effort to them, even if GE does the code. Similarly, BTL will have to exert effort on techniques for root finding, since good root finding techniques are not easy to program. The same statement holds for any routines we may need for such jobs as evaluating the hypergeometric function or finding solutions of simultaneous non-linear equations. Such programs involve a substantial amount of research in numerical analysis.

3.9.4 Statistical Routines

The development of statistical routines for such purposes as analysis of variance and multiple regression is beyond the competence of the computation center software groups, and must therefore be left to the various user groups sophisticated in practical statistical methods. Development of a set of routines for use in data laundering is an urgent task, and could be undertaken by any of a variety of groups; it is hoped that competent people with both the enthusiasm and the time to do this job will appear.

3.9.5 User Input-Output

Input-output routines in the new software will be invoked in two different manners. The automatic filing and retrieval required for operation of the single-level store will,

of course, use the basic I-O routines. There must also be facilities available for user programs to control I-O explicitly. The calls available for this purpose will be constructed for specification by exception; that is, any parameter which is not specified by the user program will be given some default value. The full specification of the characteristics of a data file might be, for instance, "tape, 200 BPI, even parity, labeled, 14 word records." However, for most files the user doesn't need to or wish to specify such detailed characteristics. He may wish to declare simply that the file is serial, or he may wish to make no statement at all about the properties of the file, and let the operating system choose. It is intended that the new software will allow the user to employ any of these degrees of specificity.

It is worth noting that eventually, although perhaps not in the first version, the I-O may pass through some intermediate transcription medium, at the convenience of the operating system. This gambit, which may be viewed as an extended buffering strategy, would, for example, deposit an output file on disk until a seven-track tape unit became available, and then copy the file. This kind of strategy can help with scheduling of peripherals.

The type conversion which is now associated with some forms of I-O is in principle an entirely separate function, and

will reside in the package of routines used for type conversion. From the point of view of the user, of course, conversion will continue to be available as part of I-O operations, as well as separately.

3.9.6 Other Utility Programs

A sort-merge package and linear programming package will be available under GECOS, provided by GE. There are no plans to make either of these packages function in the new software except via the GECOS submonitor.

A context editor like the one currently available in CTSS will be part of the new software. It will be programmed at MIT.

4.1 Performance Statistics and Accounting

For the benefit of computation center staff and systems programmers it is important that provision should be incorporated in the software for gathering of operational statistics at various levels of detail. This is even more desirable with a multi-access, multiprogrammed system than with our current batch processing mode. Such parameters as percentage of idle time, percentage of CPU time consumed in paging and current drum and disk occupancy will be very hard to obtain unless plans are made from the inception of the software effort to have them gathered. Estimating the effectiveness of scheduling and paging algorithms involves obtaining information on such items as queue length and

waiting time distributions. In addition, it should be possible to keep track of users of certain routines (e.g., NPL, the microfilm package), obtaining general information such as who did what, when, and how did it turn out.

Unfortunately, past experience has indicated that the measurements one wants to take are frequently those that were not thought of in advance, and are therefore hard to make. In this area BTL can profit by the experience of Project MAC on their current time-sharing system.

Accounting and charging are related to performance statistics from a programming point of view, since much of the information developed for either application is relevant to the other. Accounting and charging practices as such enter into software design only peripherally. It is necessary for the software task force to ensure that the information required for accounting is available and is developed, and It is of great concern to the software designers that charging practices should conform with software design objectives in the sense of tending to balance the load from the system design point and not causing substantial inefficiency. (Incidentally it is desirable to have lower rates for low priority use in order to help equalize the load.)

4.2 Documentation

Documentation of the 636 software will be a large part of the total software effort. Six categories of documentation need to be distinguished:

1) Interim working documents, generated as a part of the design and implementation process.

2) Specification documents.

3) Maintenance documentation, intended for use by the programmers who have to keep the software running and make changes to it.

4) User reference manuals.

5) Tutorial documents for new users.

6) Published papers.

The interim working documents are currently being produced in considerable numbers, and this process will continue throughout the project. Such documents as this one are intended to be of strictly temporary interest to a rather limited group.

It would be unreasonable to expect that the other classes of documents will be produced entirely by the programmers who write the software. Such documents will be written by a group consisting partly of the programmers who produce software, and partly of trained documenters from such areas as the manufacturing information groups at BTL.

Our objective should be to have user reference manuals produced on the same time scale as the software itself; maintenance documentation will lag somewhat behind, but the lag should be as short as possible (a few months). Tutorial documents will undoubtedly come along on a slower

time scale, since they will have to be revised after experience in training new users on a functioning system.

A sixth type of document which we must strive to produce is published papers describing any new and interesting aspects of the software. It would probably be unwise to invest the project with an aura of "publish or perish." It would be equally unwise to ignore the importance of readable published accounts of new developments embedded in the software.

4.3 Programmer Education

Some four to six months before the new system becomes generally available to users it will be necessary to begin explaining in detail to prospective users what facilities will be available and how to use those facilities. This will be at a time when reference manuals and tutorial manuals will be available only in part, and probably in a not very satisfactory form. In view of this, the system programmers must consider giving preliminary courses to users, and generally helping users to learn about the new system. This effort should be beneficial, in that it will provide feedback from users on unacceptable or awkward aspects of the software. It will also be a considerable hardship to programmers engaged in the final stages of coding and debugging.

4.4 Software Maintenance Responsibility

The degree of acceptance which the new software receives will be directly related to its usefulness. Its

usefulness, in turn, will be considerably reduced if it doesn't work. Since a great deal of experience has shown that no large program is ever free of errors, the software will have to be maintained.

General Electric intends to assume primary responsibility for maintenance of the 636 software, including portions written at BTL as well as portions written by MIT and by GE itself. However, much past experience has indicated that programs as large and complicated as those involved in the 636 software can only be maintained properly if the original designers and implementers are available to assist in the work. Thus, we must expect that for a period of a year or more after initial cutover of the new software, all the BTL people involved in its production will have to spend part of their time on consulting and maintenance.

4.5 Debugging

Debugging seems to be one of the fundamental problems in the efficient operation of the entire system. (See software committee document 57 by W. S. Brown.) Its effective implementation influences almost all parts of the system, and thus cannot be isolated to a single part of the system. The most effective techniques of all, however, are those devoted to the prevention of bugs in the first place, and thus it seems paramount to enforce certain standards on the system itself as well as on its users. These include imposing stringent

requirements on documentation (see 4.2), having available analytic tools such as RUFUS and FORTRACE, and generally taking out of the hand of the user tedious details in which he is likely to make mistakes, such as subroutine calls, input-output handling, etc. It is, of course, also to the user's advantage to use that language which is most suited to his program.

To help in the detection, location and extermination of bugs, various software facilities are desired in addition to sets of standards (the latter including the requirements of error checks, self-identifying structures, the ability to replicate a run at a later time). Two types of facilities are required. One is an editing and debugging facility peculiar to each translator and language being used. An example of this type is MADBUG (see MIT document CC-247, MAC-205), for use with the MAD language. MADBUG in its current form provides for controlled execution with insertable break-points, source language editing, provisional changes, and interrogation of the resulting machine code at a symbolic level. (A successor of MADBUG for the GE 636 will exist, but presumably will have little use at BTL.) There presumably should be one such facility for each translator in the system, i.e., one for NPL, FORTRAN, etc. They will all have in common the table of the assembler, linker and binder, and will probably

require other tables as well. Each translator should have appropriate symbolic print routines, geared to the data structures by providing the appropriate conversion and format, and inserting identifying names where desired. Thus the design of the respective translators depends to some extent on debugging needs in order to assure that relevant information is available.

A second type of facility involves the system directly, and is an extension of OEDIPUS (W. S. Brown, Comm. ACM, June, 1965). The translator and utility packages depend on a considerable amount of software, here called the supervisor. Some of this software is directly concerned with debugging, such as mechanisms for symbolic snaps, for setting and printing of remarks and for post-mortems. The supervisor will also contain coarse and fine dynamic storage allocation, the stack, and scheduler. The contents of the stack, the data structures which have been dynamically allocated, and the scheduling information must be available to the symbolic snap routine. The symbolic snap routine will locate a data structure, identify it (data structures will be self-identifying), and will consult a table to find an appropriate conversion and output routine. The table must of course be provided by the translator or processor being used. If no appropriate

entry is found, there will be a standard default output routine. It is up to the writers of the translators and utility packages to take advantage of the above debugging facilities. The facilities for debugging and their implications on the rest of the operating system will be discussed in a forthcoming document.

P. G. Neumann

P. G. NEUMANN

V. A. Vyssotsky

V. A. VYSSOTSKY

MH-1271-PGN
1273-VAV-AK