MCTS CUSTOMER TASK ENVIRONMENT

R. R. Brown
Computer Science Department
Research Laboratories
General Motors Corporation
Warren, Michigan

THE MCTS CUSTOMER TASK ENVIRONMENT

The customer task† environment on the MCTS system (Multiple Console Time Sharing System) was designed to attack many of the problems which the MCTS designers thought were important to a large, multi-faceted corporation such as General Motors. As such the designers considered such problems as providing adequate computing capacity for performing significant numerical analysis computations, providing high interaction rates for attached graphical (or other) terminals, providing flexible, safe, and secure access to adequate volumes of on-line or off-line data, and providing a programming environment in which it would be very easy to work and which would allow a very broad range of scientific and commercial problems to be solved. As important as these considerations are they must be provided in such an efficient form that the cost of computing in the customer task environment would be significantly lower than in previous systems.

The application problem chosen as the model of a large portion of the work load expected on MCTS was one with which GMR already had considerable experience: doing automobile design at graphics terminals. In fact the cost/benefit studies which justified the construction of MCTS assumed that the entire cost of running the system would be borne by such graphics design applications. It was hoped that many other sorts of applications would use MCTS, some at terminals, some unattended (i.e. batch), and thus nothing in the system restricted its use to graphics design applications. It was felt that the load characteristics imposed on a system by a graphics design application were so severe that most other applications would appear easier. In the paragraphs to follow the major topics to be described are:

1. Customer Task Space Management
2. Programming Language Support
3. File System Usage in Virtual Memory
4. Support of Physical I/O Devices
5. Customer Task Initialization
6. Communication Between the Customer Task and the Console User
7. Requests for System Services

---

† As used in this paper customer task means a dispatchable entity which supports the execution of applications programs. Said in a slightly different way, it is the MCTS-maintained control information plus the executable procedures and the data to be referenced by those procedures, comprising all portions of the execution support for an interactive terminal or unattended batch job, which are not provided as parts of the MCTS system.

As used in this paper customer (also referred to as customer task programmer) refers to the person who programs the procedures which execute as a part of a customer task.

The order in which these topics are discussed, while partially arbitrary, does have some significance. The first four topics are properly discussed together because the system designers did their designs with certain prejudices explicitly stated as guiding principles, and these led to the MCTS design being created so that the file system, the programming language, and the space management aspects of the customer task were all designed as a unit, and therefore are reasonably discussed as a unit. The discussion of these first four topics shows how system control is retained over several routine programming chores, and topic six shows something of the control considerations involved in supporting communication between the customer task and the interactive terminal which is (ordinarily) connected to it. Topic five precedes the graphics discussion because some mention of task initialization is useful in describing the flexibility of the graphics environment†.

---

† Most of this paper is written with the viewpoint that the user is a person sitting in front of a graphic console, trying to solve an application problem, e.g. selecting a windshield wiper system which will clean a windshield adequately to provide good forward vision for an automobile driver. Writing in this way is convenient, because it allows words to be chosen which reflect this circumstance, rather than attempting to choose words which reflect the more general situation which is in fact supported.

Due largely to the fact that a customer task's environment is one which is self-initialized by executable code found in a standard file, there is actually great variety possible in the choices of both terminals to be supported and the collection of commands which are to be the command language for some particular application. In fact, there need not even be a terminal involved at all, since a predefined sequence of character strings to be found in some stored file can serve as a "user" and can control a customer task in batch mode.

This self-initialization process also results in establishing the Console Operating Subsystem as a set of procedures which obey the same execution rules as any set of customer-written procedures. It communicates to the user via CALLs to a small number of interface modules which pass data on to the MCTS system proper. By changing the file which is attached to supply the Console Operating Subsystem procedures, different terminal types can be supported. Similarly, by changing the file which is attached to supply the set of commands for a customer task, its command language can be made anything one desires. It is entirely feasible to support a variety of terminals with a variety of command languages, simultaneously.

A partial list of the aforementioned design prejudices follows.

> MCTS should be designed as an integrated whole: Every aspect of the design must support and be supported by each other aspect. As corollaries of this: one file management technique in the system, that one accessible through constructs within the programming language; one programming language; space management performed without customer programmer involvement, and as part of file management and language support.
>
> Simplicity in all design aspects is necessary, which leads to two important results: a smaller total system which then is easier and cheaper to implement, document, understand, and use; and few options exist for the customer programmer to consider when he is designing his application program.

In the usual case, one well supported means of accomplishing an objective exists for the customer programmer to use. In keeping with this situation:

> Control over as many as possible of the routine, "housekeeping" matters involved in supporting the execution of a customer task should be retained by the various aspects of MCTS.

Finally, probably the most significant of all the design prejudices was:

> Design the customer task programming environment first, and then design the MCTS system to support it well. The system should serve the customer, not vice versa.

Overview

The environment described in this paper is that in which the customer tasks of MCTS run. It is a software environment designed to support the chosen programming language, APPLE, and its subset MALUS III. The task environment features a very large (2^{48} bits) virtual memory /13 - 17/ which is demand paged by the operating system, and which holds images of the files which the customer task uses. Thus access to data in a file is gained by executing language statements which reference virtual memory, such as FIND, INSERT, OPEN, and assignment statements. No input or output statement is needed in the language.

The command language is a collection of operators (interpretable files) for processing by COS (the Console Operating Subsystem), which is an interpreter. The displays managed by this operator allowed the user sitting at a console to act in a very flexible manner, interacting with the system and with his programs in ways which assist both checkout and productive use. The programmers doing work to develop MCTS found that the customer environment was a very convenient working environment. The command language most widely used by the MCTS development programmers made accessible to the user at a

console an editor, a compiler, a debugger, the file system, and any programs which were stored in procedure files. One could require that a named procedure in a named file be executed, and one could pass parameters to this top-level procedure just as if it were being called from another procedure.

Customer Task Space Management

Studies of programmer time distributions on a predecessor operating system /1 - 9/ showed that the application programmers were spending more than half of their programming time arranging for space management functions within their address space. Data gathered from monitoring execution of these programs when in productive use by engineering designers showed that 65% of the charged CPU time was being spent in performing these same space management functions. Accordingly a primary goal of customer task environment design was to eliminate or reduce this oppressive, expensive burden. The approach chosen was to provide each customer task with a virtual memory of adequate size so that space management wasn't necessary in customer code, thus putting the burden of space management onto the operating system. In other words, each customer task was to be supplied with a virtual memory which was so large that an application program could have all of its programs and data simultaneously occupying portions of its address space. No address space region would ever need to be emptied for reuse. Thus one resource, customer task address space, was removed from the control of the customer task code, and simultaneously removed were the problems of managing this resource.

It was further decided that the MCTS system would manage each customer's virtual memory on the basis of demand paging, thus keeping the total control responsibility within the system. Some consideration was given to supporting "advisory functions" which a customer task could use in exceptional cases to request some specific pre-paging, but the need was never proved and the experiment was never tried.

Programming Language Support

Another major decision about the customer task environment had to do with the choice of programming language(s) to be made available to customer programmers. For several reasons, some of which are stated below, the choice was to define a language which was a dialect of PL/I, and to produce a compiler for it.

PL/I was felt to be suitable for coding a much wider range of useful applications than other languages, and thus a suitably chosen subset of PL/I should retain that characteristic. In further support of this position, it is observable that PL/I has achieved a wide usage in General Motors, and for many sorts of applications. This appears to be a continuing trend.

This department of the General Motors Research Laboratories and a cooperating department of another General Motors Staff /18/ already had designed, implemented a translator for, and extensively used a language which was an extension of PL/I.

The extensions provided for the manipulation of associative relationships among data, and this language (named APL, Associative Programming Language /10/) had been used very successfully in graphical automobile design applications.

Both departments felt that this language had advantages over PL/I which would be useful in a wide range of applications. Accordingly the decision was made to extend a subset of PL/I by including the associative data handling facilities from APL. Since there was a naming conflict with the more heavily publicized APL by Iverson, the name APPLE was chosen for the language to be supported by MCTS. The first versions of this language to be implemented were subsets of APPLE chosen to support the buildup of MCTS and to ignore such things as floating point arithmetic capability. These versions of APPLE were known as MALUS I, II, AND III.

A decision to implement a compiler for the APPLE language in order to be able to optimize the generated code was originally made when it was intended to implement MCTS on an IBM 360/67. Had there not been such strong concern about optimizing the generated code sequences, a preprocessor could have been built (as done earlier for APL) to generate source code for some available PL/I compiler. Later of course, when it was decided to implement MCTS on the Control Data Corporation STAR-100, there was no available PL/I compiler for such a new machine, so a compiler would have had to be built in any case.

The other significant decision made about customer task programming languages was that there would be only one language supported in MCTS. In this way the language support codes, i.e. the execution library, and the MCTS operating system could be coded to know about each other's data passing and receiving conventions, and thus could communicate more readily and more economically. In contrast to a more usual situation in which the operating system provides a general, non-specific set of language support functions, and in which the languages require some particularized version, the particular support needed by APPLE is the only support provided by the MCTS system to its customer tasks. The biggest gains attributable to such a strategy come from reduced needs for operating system tests to see what options are needed to support which customer tasks; and hence the operating system runs more efficiently, leaving more CPU time available to the customer tasks.

The major disadvantage of such a strategy is of course having to solve the problem of supporting customers who wish to execute programs which have previously been written, perhaps by other people and at other locations, in other languages. This problem received some study with particular reference to running programs written in FORTRAN IV, but no decision as to the solution technique to employ was made. The writer's recommendation was for source language translation into APPLE source code, to continue the advantages aimed for with the system design, as described earlier.

File System Usage in Virtual Memory

The existence of an adequately large virtual memory address space, managed by the MCTS system on a demand paged basis, provided the possibility of avoiding

the use of "access methods" for data files. That option was studied: one access method (paging) versus several access methods (typically one for each type of data file organization), and finally it was asked why programmers in the MCTS customer task environment might need various data file organizations.

It was concluded that one sort of data file organization was enough if that was an adequately powerful organization, and it was concluded that the data organization provided by the GMR-developed Associative Programming Language was powerful enough to eliminate the need for such specialized organizations as sequential, keyed, partitioned, indexed sequential, inverted.

Reducing all of these various file organizations to a single one has some obvious advantages in many diverse aspects of both system structure and customer code structure. The file system need no longer concern itself with the internal structures of several types of differently organized files, and therefore the catalogue structure need no longer record the internal structure of a file. The operating system need no longer provide and maintain several sets of code to search for records specified in various ways, et cetera. The programmer need no longer study the various file organizations available to him and try to decide which one to use, and perhaps also what sub-options of that organization would be best. In short, the decision to support just one type of file organization showed such tremendous implied reductions of complexity in all areas of customer and system software that it was felt that it just had to be the correct solution.

The implications for customer task programmers of the system decision to support only one file type for data files gave rise to questions such as: What is the supported data file organization? How does one store data into a file? How does one retrieve data from a file? How is it that one is allowed to put such diverse data types as source text, compiled and executable code, printable or punchable output files, arithmetic data, and other types, all into files with a single organization type? Does the data structure reflect itself in the program's structure? These five questions will be repeated and then answered in the paragraphs which follow.

Question One: Organization

What is the supported data file organization? A file is organized for accessibility via the APPLE language, and has one of two basic internal arrangements: either structured, which will be discussed briefly in the paragraphs which follow, or unstructured, in which case the programmer must have some special knowledge about the organization of the data and must do all manipulations with no particular help from the system. The unstructured type of file is used in a few special places by the customer task support codes, and ordinarily the unstructured file is not intended for customer programmer use. Such a file has no imposed structural requirements except for its maximum length, which is limited to 2^{32} bits.

A structured file contains its own directories, a description of the data associations it is recording, records of its own free space, and a set of pointers to these directories, descriptions, and records so that all of these structure data can be found from knowing only one original address within the file: that being the address of the list of pointers. In addition to these structure data the file will usually contain some application data, which will be contained in records. The records will be (logically) clustered into sets, with each record being a member of one or more sets, and with some records being perhaps heads of some number of sets. The internal structure of such a file is thus seen to be a collection of threaded lists, but with a difference reflected in the name "Associative Programming Language." Depending on the whims of the customer task programmer, the relationship of association between certain data can be recorded in such a file in many ways by use of APPLE language constructs (as opposed to subroutine calls). For example, a programmer could choose to build a file to hold a number of compiled procedures and some relevant data about them. The programmer might wish to put into this file the following data about each such procedure: the source code; the compiled, executable code; a separate text file describing the procedure and its flow chart; and some notes about the history of amendments made to each procedure. The programmer might then choose to record these data in a structure similar to the one sketched in Figure 1. Both sets and records are objects which are declared in DECLARE statements in the APPLE language, and records (and some kinds of sets) can carry names. A set ends up being a collection of associated records, whereas each record is a STRUCTURE and will typically hold some application data within itself.

Question Two: Store

How does one store data into a file? Within an APPLE PROCEDURE one DECLAREs a record with subfields which match the data types and quantities one wishes to store. Then one ALLOCATEs the record in the particular file of interest, perhaps by specifying the file name in the ALLOCATE statement. Then one puts user data into the record by assignment statements. Finally, one organizes this record to be with associated records by INSERTing the record IN some set of the user's choice. This puts data into the virtual memory image of the file. To make permanent changes in the file one must SAVE the virtual memory file by means of a CALL to the SAVE procedure. This copies all changed pages of the virtual memory image back to permanent storage in the file system.

Question Three: Retrieve

How does one retrieve data from a file? One begins by making the file accessible in his virtual memory by a CALL to the OPEN procedure. At that time one's access privileges are checked by the file system (the following paragraph comments briefly on access privileges available to people) and if one is permitted to do at least as much as one asks to be able to do, then the file is made accessible to one's programs at some system-chosen range of one's address space. As a result of this

EXAMPLE: PROCEDURES AND RELATED DATA IN A STRUCTURED FILE

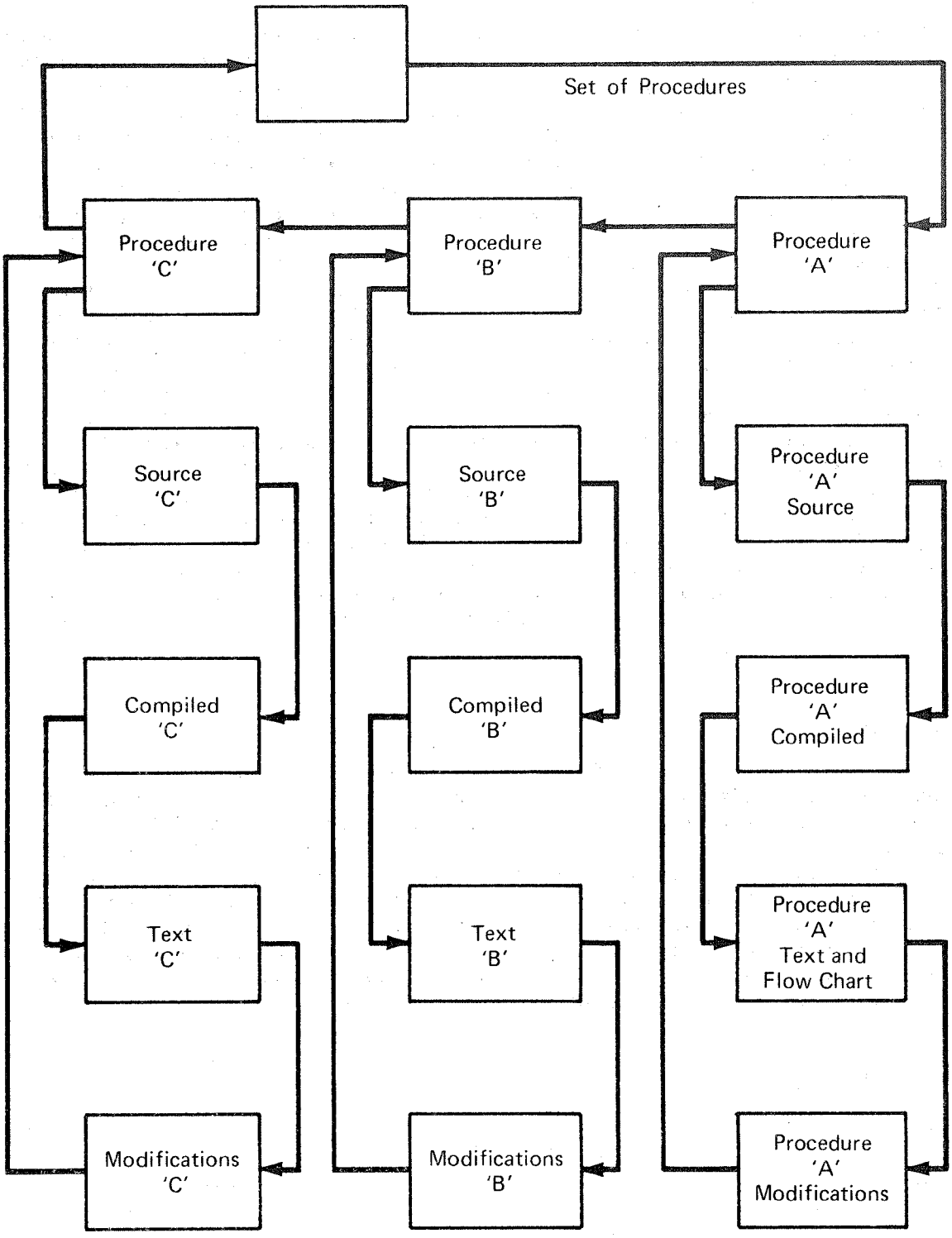


Figure 1

CALL, the program in execution is supplied with a returned value contained in a location which has been declared to be a FILE variable. Thereafter, the use of this variable in an APPLE statement will cause the statement's action to be performed on this file's image in virtual memory. Direct computational reference to all the data in the image of the file is then available via APPLE statements such as FIND, FOR EACH, INSERT, REMOVE, et cetera, each of which references some file variable. The actual structure of a file variable is of course just an address; in particular it is that address at which the directory of the named structured file begins. This sort of technique for fetching and storing data by direct reference is widely known as Virtual I/O, and it is the only technique available to a customer task. Less refined techniques, such as might invoke "access methods", are unnecessary.

The file system defined several different sorts of access privilege which a customer task might wish to use on a file, and it appeared to be useful and practical to implement the full range of access types specified. This range allowed a non-hierarchical access attribute structure, with attributes including independent specification of the ability to read, or write and store permanently, or execute, or write but not store permanently, and also the ability to change the access authorization list specifying who has what authorized access privileges. One could also (according to plan) ask for various combinations of private access or shared access to files. However, the early implementation of the file system allowed only private access to files, and with access combinations of only 1) read and write-with-permission-to-store-changes, and 2) execute and read and write-with-permission-to-store-changes.

It is hoped that this brief discussion, backed up as one wishes by more documentation on the features of APL /10/ or the APPLE languages /12/ will serve to indicate the sort of access tools which were available to a customer programmer.

Question Four: Diversity

How is it that one is allowed to put such diverse data types as source text, compiled and executable code, printable or punchable output files, arithmetic data, and other types, all into files with a single organization type? In other words, of course, why doesn't one need sequential files, partitioned files, keyed files, inverted files, random access files, et cetera? Clearly all the mentioned data forms need to be stored and retrieved, so why are these specialized file organizations unnecessary?

The proximate cause is that the relevant system support processors were programmed to do their work knowing that MCTS-style files, i.e. structured files, held their data, rather than these other amazingly specialized file types. More fundamentally, there is no reason why a generalized file organization cannot be home to any form of data. The data are, after all, only bits which retain some special internal organization, and that internal organization need not be known to anyone except

ones who need to interpret portions of the internal organization. This was recognized in the MCTS file system design in the requirement that nothing be known by the file system about the contents of any file (except the file system catalogue) or about the organization internal to a file (except the one-word pointer associated with each file and retained by the catalogue, which located the file's internal directories). The file system managed physical space and permitted bits to be stored there.

Question Five: Program

Finally the most crucial question: Does the data structure reflect itself in the program's organization? This of course hearkens back to the earliest days of computing when many of the previously mentioned properties of data files did tnen also appertain. At that time different data structures required different programs for their manipulation. The advent of "access methods" was intended as a break with that requirement, and indeed it was, from the customer programmer's point of view.

The data-structure-dependent codes still exist when access methods are used, but they have become subroutines; selected, loaded, and executed by operating system support codes in response to rather general statements written by customer programmers, such as READ, WRITE, GET, and PUT.

The MCTS file organization allows yet another forward step to be taken, because in addition to all of the aforementioned advantages, no very significant reflection of a file's structure need be imaged in a customer program. The data file itself retains the knowledge of its structure. Programs do of course need to know what data they should process but the files retain adequate structural information to allow programs to locate those data. Thus a customer program can ask to process all RECORDs of a given type, where that type is specified in a DECLARE statement, and a general language support subroutine will be automatically invoked to search the data file (named) along relevant sets (specified in a natural manner) to find all such records. The contents of a specified record can be anything one wishes: compiled programs, source text, floating point numbers, et cetera. Control at this internal level is handled by the programs which process the data, as guided by their DECLARE statements. There is some similarity between the FIND statement of APPLE and the GET or READ statements of more primitive languages, since these statements allow data to be located and prepared for subsequent processing. Much less programmer involvement is required if one uses the FIND logic, however.

In summary one should probably conclude that the MCTS strategy of supporting one file structure for customer task use, and of mapping accessible files into an extent of customer task address space, was good for the customer programmer since it relieved him of some problems and imposed no new restrictions. It was also good for the operating system structure since fewer options and special cases had to be supported; thus the operating system could be simpler and could execute more rapidly.

Support of Physical I/O Devices

Another question which should be attended to is the question of physical devices. How does the file system map data between virtual memory and physical devices, such as disks, tapes, cards, printers, et cetera?

Remembering that the file system only manages space and that it knows essentially nothing of the internal structures of the files which occupy the space; the mapping of a file between virtual memory and, for example a card punch, can be recognized as an easy two step transformation. The first step is to copy the virtual memory space onto some space managed by the file system, and the second step is to require that a system support program which is especially written to handle the card punch be invoked to move the data bytes from the file system space to the card punch. In this way the unformatted space (of a disk, typically) is used as the intermediate storage form for all data files, no matter what their origin nor their destination. Thus one can arrange to find data in virtual memory which are the images of data as they were on cards, or on a magnetic tape, if one wishes. Of course the system support program which drives the physical device in question could be programmed to do some data reformatting as it moves data between the file system space and its device, if such reformatting seems useful. The MCTS system actually supported only three external file formats: one for printable files and two for files going to a card punch or from a card reader. As an aside, it might be useful to note that in MCTS the device drivers which moved data between their devices and the file system's space operated in the stations, i.e. the peripheral mini-computers clustered around the main STAR-100 processor /13 - 17/.

Customer Task Initialization

The process of initializing the MCTS customer task is triggered when a customer presents himself to the system, as for example by trying to logon at a terminal. Most of the process is performed from within the embryo customer task itself, as supported by requested system services.

When the system observes the customer attempting to logon, it creates an embryo customer task and starts it into execution. At the instant when the customer task is about to start executing its first instruction (of self-initialization code) it has available to it three things: (1) the self-initialization code in a system-owned file, shared for read and execute access with all other customer tasks; (2) a system-owned collection of control information about this task which the customer task can never access; and (3) one page of virtual space it can both read and write, viz. the page starting at address 0 and spanning the address space 0-4095 bytes. This latter item is referred to as the task's "Page Zero," and the hardware definition of the STAR-100 processor specifies that the bottom half of each task's page zero contains that task's 256 registers. In summary, the customer task starts with some code, its registers, 2048 bytes of space, and some inaccessible control tables.

This self-initialization code then issues requests to the operating system to OPEN the various files needed to support the execution of standard APPLE procedures: the APPLE support library of subroutines, a scratch file for variables of the "automatic" storage class and another for variables of "static" storage class. It OPENs a file for printable output for regular execution, and it OPENs the file containing the command language code and control tables. When this has been accomplished there is a full APPLE execution environment available to support execution of the command language (an interpreter) and any program it calls.

The debugger is automatically started up by the MCTS operating system if the customer program generates an error severe enough to cause an interrupt. The debugger startup process involves OPENing the debugger executable code file, print, automatic, and static files for the debugger separate from the ones used by the interrupted customer task, and executing enough checking code for the customer task to verify that all the language support facilities needed for error-free debugger execution are available and contain accurate information. The debugger then executes and allows many diagnostic and corrective actions to be taken, and will also allow the attempted resumption of the main program from the point of interrupt. The debugger has no special privileges and thus cannot be used to perform any action that a normal customer program cannot perform.

Communication Between the Customer Task and the Console User

The preceding sections have spoken about the portions of the customer task environment which reflect the mediation of MCTS between the executing customer task procedures and the central computer. They indicate that that mediation process retains the mundane control aspects over the CPU, the execution store (virtual memory), and the file system, relieving the customer task procedures of those tiresome chores.

This section will speak about the analogous mediation of MCTS between the executing customer task procedures and the application oriented user† who controls the task. This section will also take the view that this mediation scenario involves the customer task procedures, the mediator (a collection of MCTS procedures known as the Console Operating Subsystem), and the user. It will concentrate primarily on the interface between the executing customer task procedures and the mediator, the Console Operating Subsystem, attempting to explain what functions the mediator performs. The Appendix describes the mediation process primarily at the interface of the mediator and the user, i.e. it describes 1) the Console Operating Subsystem (COS), 2) its associated compiler for writing command language graphics interfaces for the individual commands, the Operator Programming Language (OPL), and 3) the user's interactions with the command interface.

---

†The term user is explained in the footnote on page 12.

Background

At the start of a complete user interaction the customer task procedures are idle, awaiting information from MCTS which will specify what computation the user has requested upon his data files. In an illustrative case (not truly typical of production uses) there is a display drawn on the screen of the user's graphic console which has been computed as the perspective projection of some collection of space curves which are represented in the data files with which the user wishes to work. Also on the display screen may well be drawn some words or other controls indicating what sort of action the user has chosen to perform next upon the data in his data files. He may, for instance, be intent upon computing a vector tangent to one of the displayed curves at a chosen point, and the words may say "CHOOSE A CURVE" and "CHOOSE A POINT" and "RESTART" in case the user has made an error, and "GO" in case he hasn't.

The functions of the console controlling portion of MCTS are to observe the physical actions of the user with his input device(s), e.g. a light pen, and translate those into identities of data elements in the user's data files so that the appropriate computation can be performed by the appropriate procedure, as written by a customer programmer.

After the requested computation has been performed, the displayed image projected on the user's console must be modified as determined by the customer task process, after which the customer task again awaits the specification of another set of input parameters.

The console controlling portion of MCTS retains the mundane control aspects over the customer task during the input collection phase of this operation. In what follows there will be presented descriptions of the problems encountered in performing this chore, and the solutions incorporated into the MCTS customer task environment.

Just as MCTS strove to simplify the customer programmer's environment by eliminating his requirements for managing his own address space and his own I/O to and from various devices; so was the attempt made to retain system, rather than customer program, control over the terminal through which a customer task's programs interact with the person using them. The strategy employed to retain system control over the user's console without preventing the customer task programs from using the console interface as they please for effective interaction with the user was to assign control responsibility in two distinct portions. MCTS assigned 1) to the customer task the duty and authority of creating the terminal display and of specifying to the console control modules which display items were created as representations of which items in the user's data files, and assigned 2) to the MCTS console control modules the duty of managing the terminal display and of recording the user's selections of display items so that the addresses of the corresponding user data file items can be passed to the appropriate customer task procedure as parameters, so that the called procedure can perform the computation which the user requires. The result of the

strategy was that the customer task environment, as seen by the programmer in that environment, was one in which he would receive a CALL to his APPLE data manipulation subroutine with a passed parameter list specifying the console user's latest choices of data elements to compute with, and he was expected to eventually CALL some APPLE program provided by the customer task which would update the user's data files and his console display, and then RETURN control to the MCTS environment which would manage his user's console until another complete interaction had been specified by the user (Figure 2). The portions of MCTS which provide this console controlling environment for the customer task are named Console Operating Subsystem and Operator Programming Language, COS and OPL.

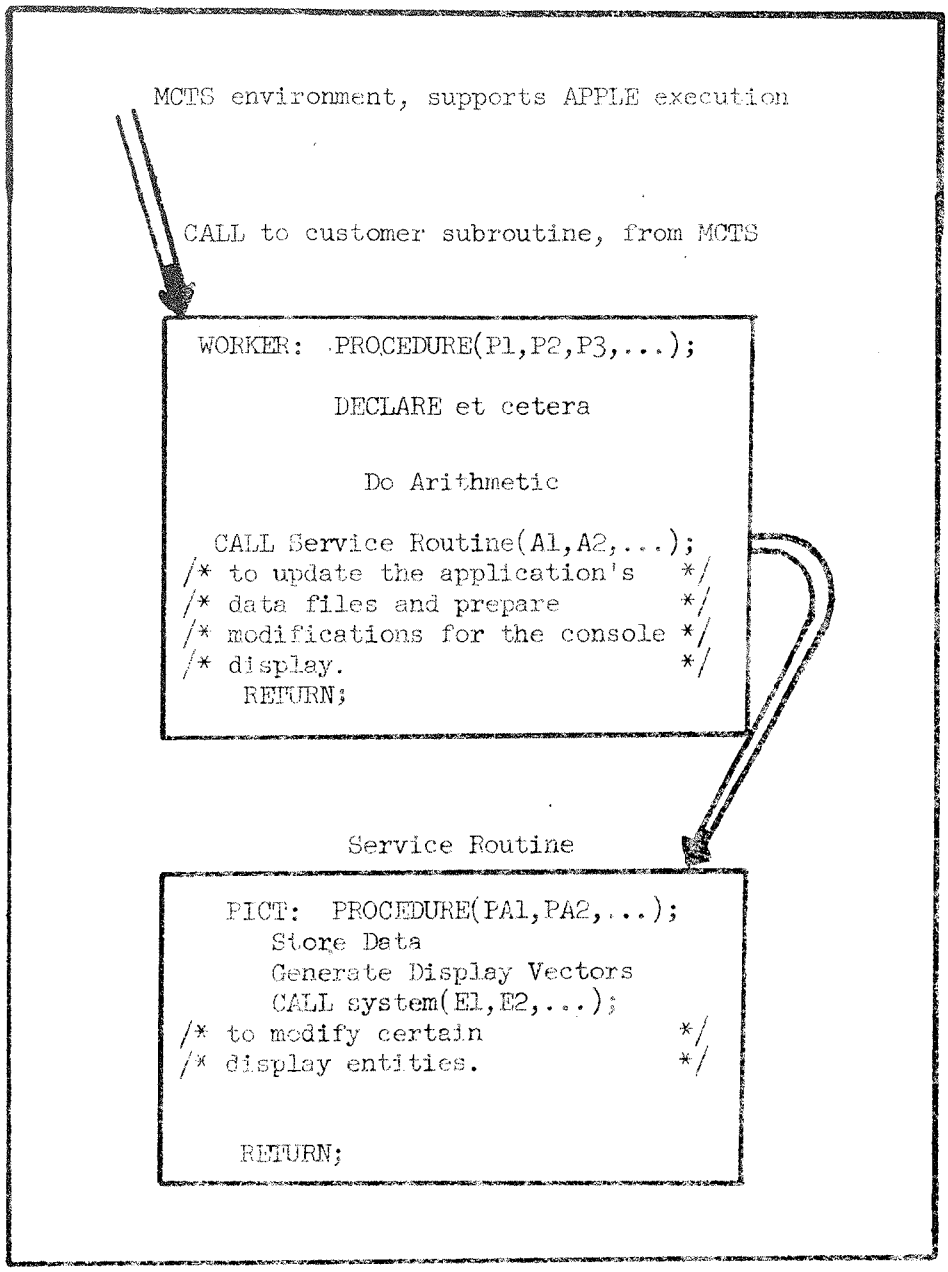
As noted above, the executing customer task is obliged to compute the output representation of the user's data files, thus clearly taking part of the console control function to itself. Since these output computations depend from both certain console utilization decisions as well as from the specific nature of the display terminal hardware, it would have been necessary to separate these various dependencies somehow if the customer task programs were to retain all their usage options while MCTS was to attend to all the terminal dependencies. In MCTS the conscious decision was made to leave both parts of this chore to customer task procedures, thereby insuring that no unforeseen design implication could deprive the customer task procedures of full control over the user's console.

An ameliorating factor in this situation, reducing the amount of effort required of the customer task programmers was that the computation could easily be done by a service routine which would be subsystem-dependent, not requiring a different service routine for each different user program, but rather one sharable by all the user programs of a single subsystem.

The choices for console control thus made available to the subsystem were such as: What technique should be used to select things from the data files to display (all, ones with special flags, text strings only,...)? What projections of the selected things should be shown (orthographic, perspective, stereo,...)? What screen format should be used (one area where everything is shown, subareas for selected subsets,...)?

A Little Deeper Look: The foregoing discussion about graphics support in the MCTS customer task has purposely kept to a very high, phenomenological level, and in an attempt to avoid mentioning any but the most essential details, it has even been slightly misleading. In the sections to follow, up to the heading Requests for System Services, this topic is additionally discussed in a little greater depth.

The major misleading aspect of the foregoing discussion concerns ignoring the moment and method of the selection of the customer task procedure which is to receive the parameter list gathered and passed to it by COS. Also ignored has been the method whereby the user at his console can direct the sequential flow of commands executed by the customer task he is controlling, and finally the mechanism which is available to a customer task programmer whereby he shares with COS the control responsibility over the console, i.e. creating output displays and collecting the user's input data selections. All of these topics are properly



Control Flow in the MCTS Customer Task

Figure 2

addressed together, since they all concern the cooperative functioning of COS and OPL, performing their portion of the terminal control chores.

OPL is a compiler whose input language allows the OPL source language programmer (assumed here for convenience to be the same person who is the customer task programmer) to create a single command, which the OPL compiler can then compile and store in an MCTS file. The results of the compilation are pieces of executable APPLE code, and a data structure which can be interpreted later by COS.

COS collects the input selections made by the user at his terminal, and treats each such selection as the specification of one datum needed by the particular customer task procedure whose execution will produce the command's effect on the user's data files. Note that the selection of a particular command causes COS to process a particular OPL-compiled file, and within that file is the information required to generate the command's terminal display and also the identity of the customer task procedure to be executed. The very considerable flexibility available to the programmer of such an OPL command arises because the OPL interpreter considers the data structure to be a tree, whose branch tips are these user-specified data and whose interior nodes are treated as "AND" and "OR" logical requirements which the input data must satisfy before the entire tree is satisfied. Even more flexibility is gained due to the inclusion of the aforementioned "pieces of executable APPLE code" which are considered to be associated with the branches between the interior logical nodes. A code piece is executed when the corresponding requirement node has been satisfied, and it is allowed to manipulate the picture being shown to the user at his terminal, and is even allowed to manipulate various of the data required at the input branch tips, whether or not they have already been specified.

When all of the requirement nodes of a command's tree representation have been satisfied, the associated customer task procedure will be CALLed with the specified input data converted and sorted as necessary into the parameter list which that procedure expects. Thus it is seen that the parameter input phase of a command is handled by COS and OPL, the compute phase is handled by an associated customer task procedure, and the output phase, in which the command's effects are imposed on the user's data files and on his display terminal, is handled by the customer task procedures, usually by a widely used subsystem service module.

Getting Even Nearer to Details

The final set of topics described in these sections about graphics-related system support, relate certain control aspects which are even nearer to the detail level. They describe three aspects of the graphics support for the MCTS customer task environment, namely: 1) identifying output images of meaningful objects in the user's data files; 2) selecting data file items as a result of interpreting the user's physical actions at the terminal, e.g. light pen selections, key strokes; and 3) specifying a complete user interaction, which requires collecting a desired set of data file item selections and passing them as arguments to a customer task data manipulation procedure.

Identifying Output Images: In preparation for the input selection phase in which the system will translate user selections of component parts of the display into selections of corresponding items in his files, this phase builds a temporary data file representing the console display. It contains information indicating which atomic portions of the display should be considered together to represent a display of each item in a file, and an identifier of that item in the user's files.

The actual implementation of the Console Operating Subsystem supported a very rich structure within this picture data file, allowing specification of many display modifying options, but the only one to be mentioned here is the capability of grouping the display elements into sets (such as "the set of all selectable sentences in the display") which have some commonality for the user's purposes.

The Console Operating Subsystem is able to create and maintain this picture data file because the service routine which computes the atomic elements of the display is required to communicate to COS in terms of "display entities" and "display sets", et cetera. Thus the contents of a display entity include the displayable atoms which represent some item in the user's files, an identifier of that item, data about display set membership, and other data controlling the other display options. CALLs to the various subroutines supporting COS allow the service routine to create entities, delete entities, change their contents, change their set memberships, create sets, et cetera.

Selecting Data File Items: Given the picture description data file built by the first phase supporting subroutines of the Console Operating Subsystem, the requirement of translating a user's physical actions (such as a light pen poke) into a selection of an item in the user's files is easily satisfied once the physical act can be related to a displayed entity. This can be a simple chore if the display has but few, well-separated items, but if the display is full of overlapping displayed items then the chore can be very difficult.

There are two principal tools which the Console Operating Subsystem uses to insure correctness in its choice of a display entity to correspond to the user's action. One is the continual, instantaneous feedback to the user of the Console Operating Subsystem's best guess as to which entity is being selected at that instant. The other is bound up with the set(s) of which an entity is a member, in that the Console Operating Subsystem can "know" that only the entities on certain sets are meant to be selectable at some particular time, so it makes its "best guesses" from only the entities on those sets. It then is up to the user to attempt to select the display entity he chooses until he sees that that is the current choice of the Console Operating Subsystem, at which time he allows that selection to stand.

In the actual MCTS implementation, the feedback was accomplished by displaying the "best guess" display entity in its entirety with considerably brighter intensity than the other displayed entities. When the user lifted his light pen from the screen, it was assumed that he was content with the latest choice.

Specifying A Complete User Interaction: The questions to be answered here are these. How can the Console Operating Subsystem be made to know that at a particular time the user is attempting to select only the display entities on one particular display set; for example, only those display entities which correspond to "center points of circles"? How can the Console Operating Subsystem know what customer task program should be called to process the data selected by the user during his interaction? How can the Console Operating Subsystem know how to pass arguments to the customer task data processing program; i.e. how can it know the number, type, and sequence of arguments expected by the customer task program? All of the answers to all of these questions form the answer to: How can the Console Operating Subsystem specify a complete user interaction to the customer task he is using?

The answers to these questions lie with the programming language mentioned earlier: Operator Programming Language (OPL). OPL would allow all these factors to be specified in its source language, and would then compile into an "operator file" a mixture of executable code and some data structures to be used by the Console Operating Subsystem, an interpreter. See Appendix.

Thus, to add a new data processing program to an existing application subsystem, the application programmer would write two things: the program itself in APPLE, and the operator which would control its console in OPL.

Requests for System Services

Some comments about requesting system services are in order here. The system services can be loosely classified into two sets: those services which can be accomplished without any special system privileges (such as finding the next entity on a known set in a known file), and those which use some system privileges (such as adding a new file to the virtual memory, which requires system checks for access privileges, virtual memory space allocation, and some table building by the operating system). Both sets of services are implemented from a customer program by issuing a subroutine call with proper arguments. For the non-privileged set of functions, that subroutine will provide the service and return control, but for the privileged set of functions control must be passed to the operating system to perform at least part of the function. That is accomplished by the called subroutine. It sets up a list of arguments which the system support codes will need, points a particular register at that list (the same register as is used to point to a normal argument list for a normal CALL statement), and issues a programmed interrupt (an EXIT FORCE instruction). The first argument in the list specifies the identifying number of the system service needed, and the remainder of the list is specified by agreement between the writers of the system support code and the subroutine which builds the argument list.

Returned arguments usually come back in the same list. All system services are reachable via subroutine call from executing customer programs, and almost all are also reachable from the command language. The command language arranges for this by issuing the proper subroutine call, just as if it were an ordinary customer program.

A slightly different environment was planned for the customer callable services in the production version of MCTS, in that in addition to the conditions already described, a small additional condition was to be imposed. It was the intention to collect all callable system service interface modules into one particular file, and then to attach that in execute only, shared mode into each customer task. Then only procedures in that segment would be allowed to issue the EXIT FORCE instruction. That rule would allow certain checking of input arguments to be performed safely within the customer task rather than in the system. This production environment was not used in the daily MCTS operation, since it would have hampered the testing of new customer callable system service interface procedures.

The simplicity of the environment was a valuable asset, allowing the customer to concentrate on his problem, not on trying to arrange that his procedures would have enough resources to run. His procedures would of course be compiled by the MALUS III compiler (which ran with acceptable rapidity even on the rather small STAR emulator computer we were using) and would automatically be assured of a MALUS III support environment in which to run.

The unanimous judgment of those who used MCTS was that it provided a powerful, flexible customer task environment, and that the system support, the programming language, the command language, and the file system all fit together very well to provide this environment. The experiment of designing all these aspects together, with the idea that they should all work in harmony to solve customer task problems should be considered as a successful experiment. Hopefully some future system will incorporate this sort of total environment planning.

APPENDIX

by John T. Murray

Communication Between the Customer Task and the Console User

The principal users of MCTS were expected to be engineers dealing with line drawing models of automotive parts and surfaces, using graphic terminals. Since the information to be processed was graphic rather than textual and since engineers are generally poor typists anyway, the user interface was not to be based on typed commands. Instead the light pen was to be used as much as possible so that the engineer might communicate with the system by simply choosing items from the options displayed. Failing that, forms would be displayed on the screen so that the user might simply "fill-in-the-blanks" with whatever typed information might be required. For example, to add a hole to a line drawing model of the inner panel of an automobile trunk lid, the engineer would first select CIRCLE from a displayed list of geometry building programs. He would then be prompted with a display identifying the input data required by that program, e.g. a CENTER POINT and RADIUS. By selecting 'CENTER POINT' and then selecting a particular point from the display of his trunk lid, the engineer could bind a particular datum to the input parameter for the program. Similarly, by keying a number into a blank displayed next to RADIUS, the engineer could supply this input datum with a minimum of typing difficulty.

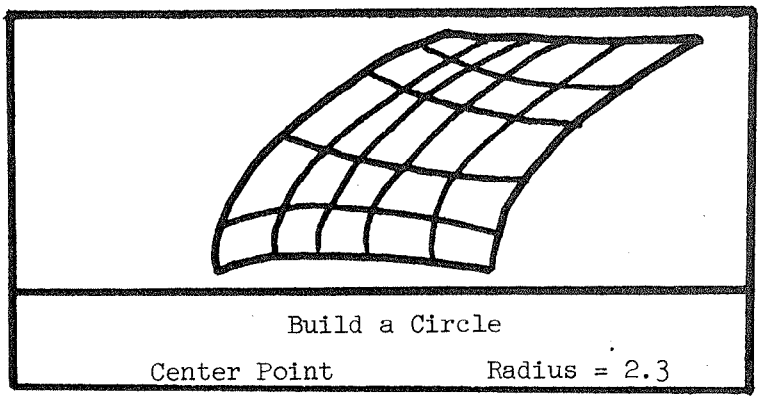
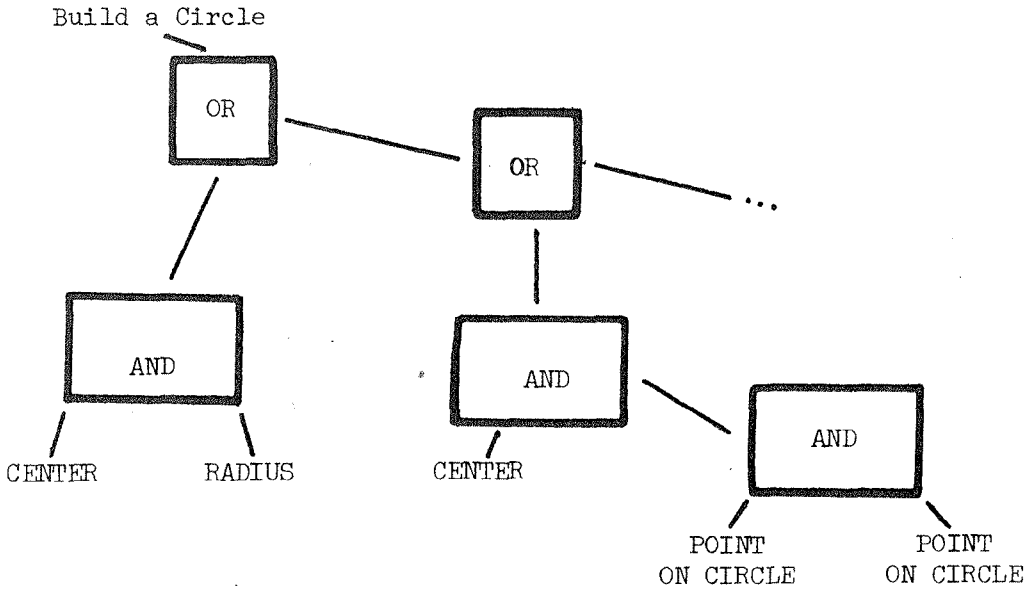


Figure 3

Experience /18/ writing programs to create this sort of user interface had shown the need for separating the programs which actually build circles (semantic programs) from the programs which communicate with the customer to obtain the required input data (syntactic programs called operator programs). The semantic programs were to be written as APPLE subroutines so that they could be invoked directly by higher level programs not requiring communication with the user. Accepting input data as parameters, these semantic programs would "build a circle" by calling service routines to add the corresponding data into the list structure representing the customer's line drawing. The service routines would be responsible for updating both the line drawing model and the display of that model, as continuously presented to the customer. The problem of updating the display involves generating the corresponding display vectors and calling the appropriate system utility to transmit them to the graphic terminal buffer memory from which the display is regenerated.

The operator programs which communicate with the customer in order to obtain input data for semantic programs were found to require inordinate programming effort /18/ to develop and maintain when written using conventional programming languages like APPLE. The example shown here requires only two input parameters (CENTER and RADIUS) and therefore oversimplifies this problem. In fact, most operator programs deal with a dozen or more input parameters. (A circle can be constructed not only from a CENTER and RADIUS but also from a CENTER and two POINTS ON THE CIRCLE, three POINTS ON THE CIRCLE, a CENTER and TANGENT LINE, two TANGENT LINES and an APPROXIMATE CENTER, etc).

The programming effort required to develop and maintain operator programs can be greatly reduced using interpretive techniques. Specifically the operator program itself has been represented as an AND/OR graph:



Operator Program AND/OR Graph

Figure 4

A general purpose interpreter program can then manage the operator display and process the user's light pen selections and type-ins to insure that the required input data (and only the required input data) are provided by the user.

In order to gain experience with this approach toward man-machine interaction on MCTS, it was decided to implement the command language, text editor and debugger along these lines. The initial MCTS terminals did not have light pens, but the command language, text editor and debugger operators were written as if they did so as to require no change when full graphic terminals became available. The initial alphanumeric terminals did have full screen read and write capability; thus it was possible to use a "fill-in-the-blank" approach. This worked out especially well in the text editor where a full page of text could be edited, and characters and lines inserted and deleted in place on the terminal screen. The approach did not work out so well in the debugger, however, where a conventional typewriter-oriented debugger was written to "short-stop" the full debug operator in order to improve response.

In the MCTS implementation, the program which interpreted the AND/OR graphs defining the command language and text editor programs was called the Console Operating Subsystem (COS) of MCTS. This subsystem became the top level program for the MCTS customer task, handling all communication with the user from LOGON to LOGOFF. As sets of input data became complete (such as the name of a program to be moved from one file to another, and the input and output file names) COS would invoke semantic routines to actually carry out the user's computational requests.

The AND/OR graphs defining a particular operator program were written in a dialect of Bachus Naur Form called the Operator Programming Language (OPL). A compiler translated the source programs into the object data structures, i.e. the AND/OR graphs to be interpreted by COS. The OPL compiler accepted semantic statements written in APPLE as associated with portions of the AND/OR graph. The semantic statements themselves were simply collected into a DO group and copied into a semantic output file. As it was produced, each DO group was numbered and the number stored at the appropriate node in the AND/OR graph. After a complete OPL operator was compiled, the semantic output file contained source statements for a valid APPLE subroutine. This file was then compiled by the APPLE compiler to produce the actual object code for the semantic program. As described above, COS would invoke the semantic programs and cause the execution of the appropriate DO group as sets of input data became complete (according to the AND/OR graph). The semantic program supporting the execution of the MCTS text editor contained several dozen DO groups for changing, inserting, deleting, searching, saving, and scrolling text. The operator structure contained on the order of 50 AND/OR nodes.

BIBLIOGRAPHY

1. Edwin L. Jacks, "A Laboratory for the Study of Graphical Man-Machine Communications", AFIPS Conference Proceedings, Volume 26, Part 1, 1964 Fall Joint Computer Conference. Spartan Books, Inc., Baltimore Md.

2. M. Phyllis Cole, Philip H. Dorn, Richard Lewis, "Operational Software in a Disk Oriented System", AFIPS Conference Proceedings, Volume 26, Part 1, 1964 Fall Joint Computer Conference. Spartan Books, Inc., Baltimore, Md.

3. Barret Hargreaves, John D. Joyce, George L. Cole, Ernest D. Foss, Richard G. Gray, Elmer M. Sharp, Robert J. Sippel, Thomas M. Spellman, Robert A. Thorpe, "Image Processing Hardware for a Man-Machine Graphical Communication", AFIPS Conference Proceedings, Volume 26, Part 1, 1964 Fall Joint Computer Conference. Spartan Books, Inc., Baltimore, Md.

4. T. R. Allen, J. E. Foote, "Input/Output Software Capability for a Man-Machine Communication and Image Processing System", AFIPS Conference Proceedings, Volume 26, Part 1, 1964 Fall Joint Computer Conference. Spartan Books, Inc., Baltimore, Md.

5. F. N. Krull, J. E. Foote, "A Line Scanning System Controlled from an On-Line Console", AFIPS Conference Proceedings, Volume 26, Part 1, 1964 Fall Joint Computer Conference. Spartan Books, Inc. Baltimore, Md.

6. Technical Information Department, "Design Augmented by Computers - The General Motors DAC-1 System", (Search 7), Research Laboratories, General Motors Corporation, General Motors Technical Center, Warren, Mich. 48090, October 1964.

7. General Motors World, "Designer's Legman", pp. 8-11, General Motors Overseas Operations Division, 767 5th Ave., New York, January-February 1965.

8. Edwin L. Jacks, "The DAC-1: A Computer System for the Study of Graphical Man-Machine Communication", General Motors Engineering Journal, pp. 2-8, Volume 12, No. 2, Second Quarter 1965.

9. Gerald J. Devere, Barrett Hargreaves, Dennis M. Walker, "The DAC-1 System", Datamation, Volume 12, No. 6, June 1966.

10. G. G. Dodd, "APL - A Language for Associative Data Handling in PL/1", 1966 Fall Joint Computer Conference, Volume 29, AFIPS Conference Proceedings.

11. R. C. Daley and J. B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS", ACM Symposium on Operating System Principles, October 1967.

12. F. N. Krull, M. Marcotty, M. S. Pickett, J. J. Thomas, and R. F. Zeilinger, APPLE Reference Manual, General Motors Research Laboratories Publication GMR-1234, General Motors Technical Center, Warren, Mi. 48090, 1972.

13. W. C. Hohn and P. D. Jones, "The Control Data STAR-100 Paging Station", AFIPS Conference Proceedings, Volume 42, pages 421-426, 1973 National Computer Conference.

14. R. L. Curtis, "Management of High Speed Memory on the STAR-100 Computer", IEEE International; Computer Conference Digest, Boston, 1971.

15. P. D. Jones, "Implicit Storage Management in the Control Data STAR-100", IEEE CompCon 1972 Digest, 1972.

16. R. G. Hintz and D. P. Tate, "Control Data STAR-100 Processor Design", IEEE CompCon 1972 Digest, 1972.

17. G. S. Christensen and P. D. Jones, "The Control Data STAR-100 File Storage Station", Fall Joint Computer Conference Proceedings, Volume 41, 1972.

18. C. Richard Lewis, "Economic Factors in the Utilization of Reactive Graphic Consoles", Society of Manufacturing Engineers.