Transaction applications have specialized requirements for scheduling and data base support. This paper describes the procedure by which those requirements were identified during the design of a data base and transaction management program for the IBM 8100 Information System. It also provides an overview of the program structure that evolved to satisfy the functional requirements.

Design of the IBM 8100 Data Base and Transaction Management System—DTMS

by F. C. H. Waters

The IBM 8100 Information System is a new minicomputer for distributed applications. Architecturally, it is an extension of the IBM 3790 Communication System. To ease the transfer of applications to the 8100 from the 3790, there is an operating system, the Distributed Processing Control Executive (DPCX), that is compatible with the 3790 control program. To handle the larger configurations and more diverse applications of the 8100, however, a new operating system, the Distributed Processing Program Executive (DPPX), has been developed. Unlike DPCX, DPPX is a full multiprogramming operating system, supporting both batch and interactive applications.

DTMS

In the initial stages of DPPX development, its planners realized that many 8100 applications would need data base facilities and a specialized transaction-processing scheduler if they were to make efficient use of the 8100. Because of our experience with Data Base/Data Communications (DB/DC) products such as IMS[1] and CICS[2], the Santa Teresa Laboratory was asked to design and implement a small DB/DC product, consisting of a Transaction Processor and a Data Base Manager. The resulting program is called the IBM 8100 Data Base and Transaction Management System, which is usually abbreviated DTMS. The progress of DTMS from initial design criteria to completed program structure is the topic of this paper.

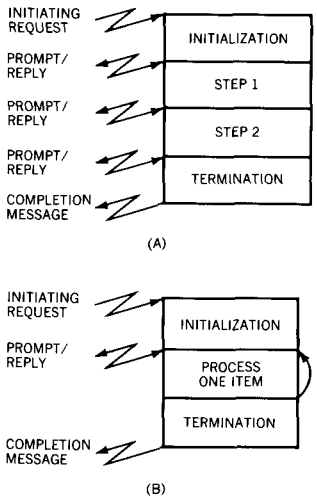
Copyright 1979 by International Business Machines Corporation. Copying is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract may be used without further permission in computer-based and other information-service systems. Permission to *republish* other excerpts should be obtained from the Editor.

This project gave us for the first time the opportunity to build a transaction processor and data base manager in concert with the design and implementation of the base operating system. By contrast, most of the transaction processors that we built for the System/360 and System/370 were designed after the operating system structure was in place. To avoid changing the operating system in ways that would have been incompatible with existing application programs, the System/360 DB/DC products had to add duplicate functions to create a transaction-oriented environment. We felt that they were also functionally richer than necessary for the 8100.

design objectives

As a basis for program design, our planners and consultants constructed several scenarios that they considered representative of the classes of use the proposed data base and transaction management system would receive. Each of these scenarios described a hypothetical company and the application or family of related applications that such a company would probably implement on the 8100.

Figure 1 Complete transactions are typically either (A) serial conversations or (B) repetitive conversations



At the same time, we adopted a number of overall ground rules for the design phase of the project. The objectives that had the most significant effect on DTMS structure were the following:

- The basic program should run in a minimal amount of space, but should be able to make efficient use of additional space to improve performance;
- A terminal should be able to stay logged on to DTMS with little overhead, until it submits a transaction;
- Data base recovery should occur by default if the system or the application program fails detectably;
- DTMS should be capable of running twenty-four hours a day indefinitely without significant performance degradation or wasted space during periods of light load;
- No detailed understanding of the internal logic of DTMS should be required for its installation and efficient execution;
- Start-up should require little effort beyond that needed to start the operating system.

Application program structures

Study of the application scenarios confirmed our experience with transaction environments and provided some new insights. Most of the expected applications were conversations between a terminal user and an application program that consisted of short bursts of processing, separated by comparatively long periods of idle time waiting for the terminal user to respond. There were comparatively few applications whose programs ran continuously for more than a second or so—and those were really batch or CPU-

intensive conversational programs for which efficient DPPX mechanisms had already been designed.

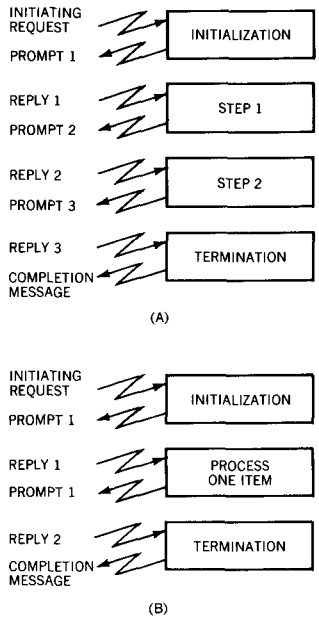
Since we were specializing in handling large numbers of short programs, we concentrated on speed of allocation and recovery of resources. We had to give the application program its execution resources and then retrieve those resources for use by someone else, with a minimum of effort.

Fortunately, the job was somewhat simplified by another common characteristic of transactions: they seldom need resources beyond the instigating terminal, space to run in, and some data from the data base. That meant that we could optimize our resource allocation mechanism for these requirements. A transaction that needs a tape drive is a very special case, and separate processing to handle it is not very expensive.

Seeking to minimize resource allocation overhead, we looked at the program structure of a typical transaction. Figure 1 illustrates the fact that most complete transactions are either (A) serial conversations with the terminal operator soliciting various pieces of information; or (B) repetitive conversations, for example, when entering all of the items on a list.

Both of these structures can be troublesome in a transaction-oriented system. The "think" time between query and response can be long, particularly if there is some physical act to be carried out by the terminal operator, or if the transaction involves two people (say a sales clerk and a customer). All that time, the conversational application program is occupying precious space.

Figure 2 Segmented conversations: (A) serial segmented conversation and (B) repetitive conversation



segmented conversations

A close look at the typical application structures in Figure 1 shows that they can be broken up into segments, as in Figure 2. Each segment can be a small, separate program. Then the space and other resources needed by the conversation are required only when a conversation segment is actually being executed. When each conversation segment application program ends, its resources can be reassigned to a segment of another conversation that is ready to resume execution. Figure 2 shows the abstract conversations from Figure 1, restructured into segmented conversations.

As a result of our initial studies, we decided to optimize the transaction processing mechanism for the execution of a segment of a conversational transaction. This is not an entirely new concept, of course; the same general approach was used, for example, in the Airlines Control Program (ACP).[3]

Figure 3 Resource requirements of a conversation segment

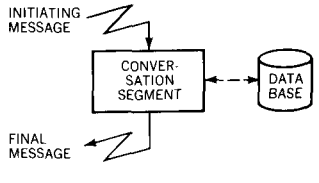
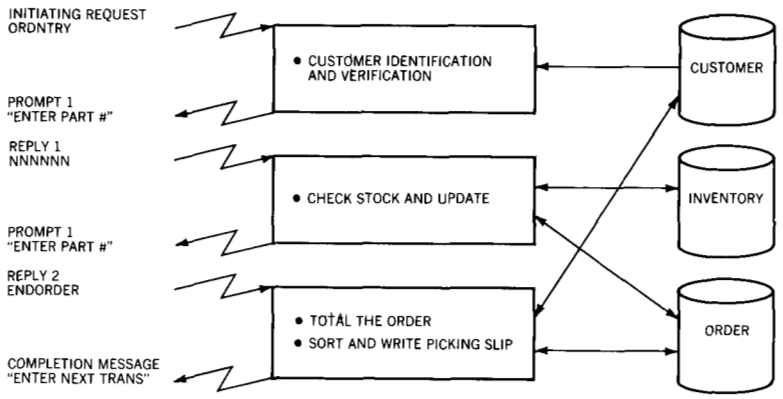


Figure 3 illustrates the modest resource requirements of the unit of DTMS processing, a *conversation segment*. The conversation

Figure 4 Structure of the APEX order entry application



segment is loaded into storage and given control. It accepts the terminal message that caused it to be started, processes the message, and returns an output message to the terminal. Data base I/O may or may not be required.

Starting from the basic element of the program segment, we identified the functions that application programmers would probably need to allow them to string together segments to form full-fledged applications. We then tried to design an efficient program for scheduling and executing these units of work: the DTMS Transaction Processing Manager.

an application scenario

To analyze function requirements, we used the application scenarios. One in particular seemed to include most of the requirements of the others, and it became our touchstone during functional design. The scenario is known as APEX because it deals with a fictitious company called "APEX Auto Parts." The structure of this fictitious company was abstracted from our experience with various distributing and retailing enterprises, and the APEX applications are our planners' and consultants' assessments of the ways a typical company of this type would use the 8100.

The APEX applications are primarily the computerizing of the clerical tasks of a wholesale parts distribution outlet: order entry, shipping and receiving, and accounts receivable. In terms of volume, the order-entry activity dominates the system. Since it also demonstrates most of the functional requirements we identified, it is used here as an example. Figure 4 shows the overall structure of the APEX order entry application.

A customer in an APEX outlet is typically from an auto repair shop and is looking for a specific list of parts to complete a job. Since

the shop usually has a running account with APEX, the first activity is to verify the customer's account number and its standing. Assuming that the account number is valid, the next step is to open a new order and direct the terminal operator to start requesting parts by number. This seemingly straightforward process may involve some complex exception routines, because the customer does not always know the part number. Even if he does know the number, that number may have been superseded. Each item ordered constitutes a separate transaction to DTMS, and an incorrect part number or obsolete item may start a digression that is a conversation within a conversation. To keep the illustration simple, we ignore exception processing.

If the part number is valid, the order entry application program obtains the inventory data base record of that part and, if the stock is sufficient to fill the order, decreases the inventory by the quantity requested. It also records in the order the part number, quantity, price, and warehouse location. It then sends a message back to the terminal to indicate that the part is available and to request another part number.

This loop continues until the entire order has been entered. Then the total amount of the charge is calculated and the customer is asked whether it is acceptable. If so, a picking slip is printed out on the warehouse floor asking that the order be collected and shipped or held for customer pickup.

Function analysis

When viewed at such a high level of abstraction, APEX seems simple. One can hardly understand why it needs help from DTMS. But consider the application programmer's implementation problems.

transaction processing functions

In the interests of efficiency and better program design, we are asking the programmer to divide the application into discrete steps. In an unsegmented conversational application program, he could keep data, such as the accumulated order record, in storage until the order is complete. But his segmented application does not have any storage during think time. To solve this problem, we decided to allow the conversation segment to pass a scratch pad of up to 4096 bytes to DTMS before terminating. DTMS retains the scratch pad during think time and then passes it to the next segment of the conversation as an input parameter when it starts execution. This process is shown in Figure 5.

Another problem with stringing together conversation segments is that of identifying the program to be run next. Since the conversation segments are logical subroutines of the overall application,

Figure 5 Use of scratch pad in order entry (ORDNTRY)

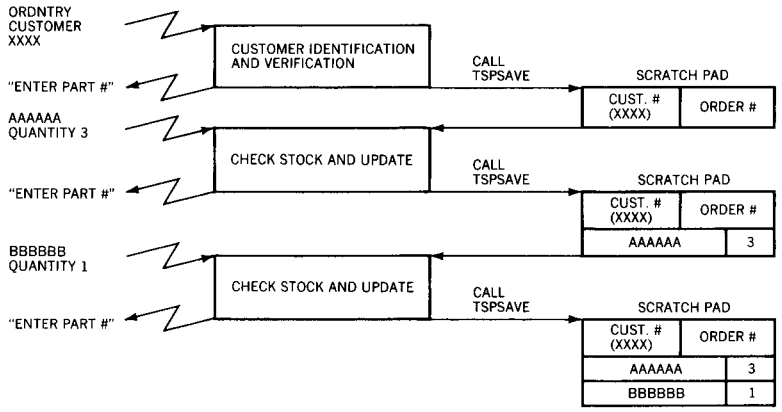
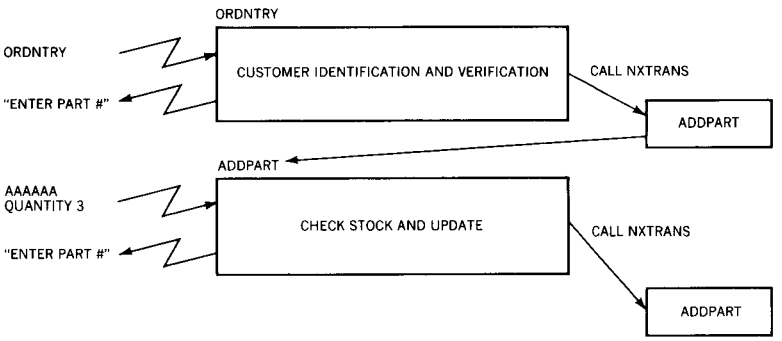


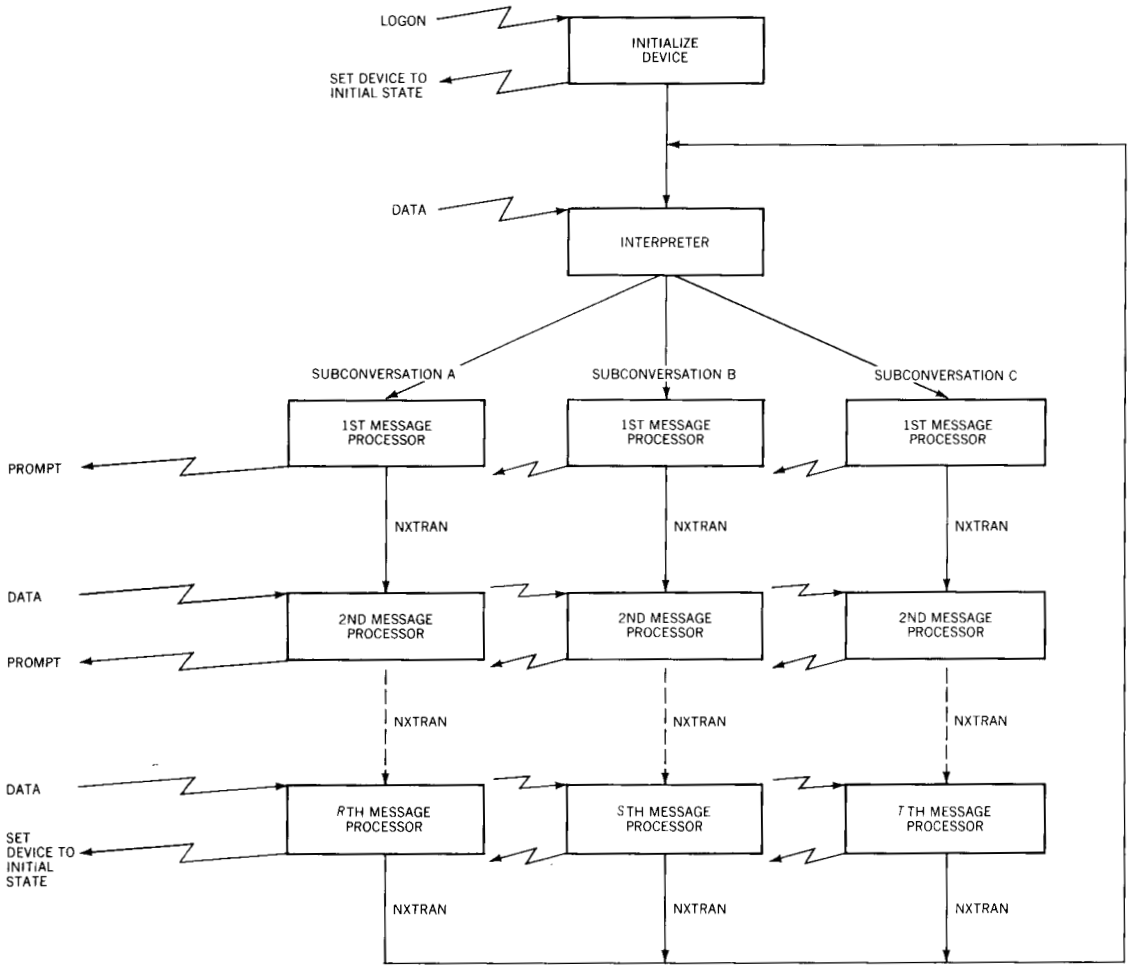
Figure 6 Use of the next transaction (NXTRANS) service



the determination of the sequence of segments should be under program control. Also, from a human factors viewpoint, the terminal operator should not have to enter the transaction name each time he wants to enter some part numbers. Yet a part number is not very helpful in determining what application program to run.

In response to this requirement, DTMS provides an interface to allow the application program to identify the transaction that is to process the next input from the terminal, as illustrated in Figure 6. Since the application programmer knows what data should be elicited by his next terminal prompt, he can specify the program to process it. If a next transaction (NXTRANS) has been indicated, the usual DTMS process of looking for the transaction name at the beginning of the input record is bypassed, so the end of the repeti-

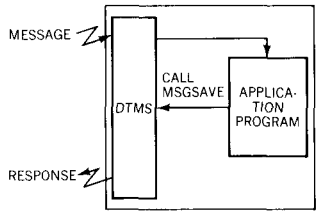
Figure 7 The never-ending segmented conversation



tive conversation cannot be indicated to DTMS by the user's reply. The ADDPART program tests for the ENDORDER reply and proceeds directly to that program.

The next-transaction function is also used in the design for one of our industrial application scenarios. Some limited-function terminals, such as magnetic stripe readers or analog-to-digital converters, are not capable of supplying a transaction name in character form. Figure 7 shows the mechanism that we designed to allow segmented conversations with such devices. We call it the never-ending segmented conversation. The first segment is an initializing transaction that is run when the terminal is logged on. In addition to initializing the device, it requests as next transaction an interpreter that is capable of accepting raw input from the terminal and starting a segmented conversation. The final segment

Figure 8 Message input and output interface



of each conversation always requests the interpreter as the next transaction. To ensure that such segmented conversations are never irretrievably lost, we designed DTMS to return to the initializing transaction (if the session has one) whenever a segment fails or ends without giving a next transaction.

A conversation segment normally has a single message as input and a single message as output, although the message may consist of a screen full of text. This fact led us to create the interface shown in Figure 8, in which the segment receives its input—the message that caused it to be executed—as a parameter of invocation. Its output message can be transferred to DTMS via a service call.

This structure has the following purposes: (1) it saves the application programmer from having to code any terminal I/O macros in his program, and (2) it allows the program to terminate without waiting for the result of terminal I/O. DTMS takes responsibility for delivery of the message, and if the transmission fails or the system comes down before the transmission is completed, the data base activity is all reset. Since the line delay in sending the message to the terminal and receiving a response back can be substantial, we save considerable space on the average by not requiring the application to wait for the response. This does not mean that the program cannot perform terminal I/O, just that it does not have to.

In addition to transaction requests entered from terminals, we have provided a facility by which an executing program can request a transaction. This service, CREATX, is available to any program in the DPPX system, whether transaction or not. The requesting application program passes DTMS a buffer containing the transaction in the same form in which it would be received from a terminal. DTMS enqueues the transaction for execution resources as it would any other. Unlike a terminal-entered transaction, however, a CREATX transaction is not released for execution before its requester has terminated successfully. If an application program fails, its CREATX requests, like its data base changes, are backed out.

Figure 9 shows the final segment of the order entry process submitting a CREATX for a transaction to be run in the session between DTMS and a terminal printer in the warehouse. When the printer becomes available, the transaction is executed and prints the picking slip.

Data base functions

So far, we have looked only at the transaction processing aspect of DTMS. The data base services were designed to serve all appli-

Figure 9 Creating a transaction (CREATX)

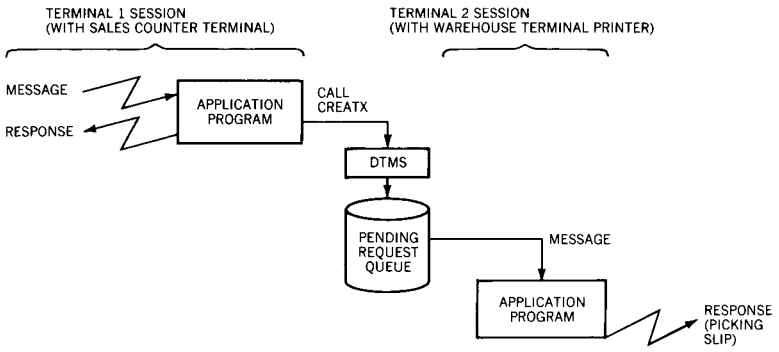
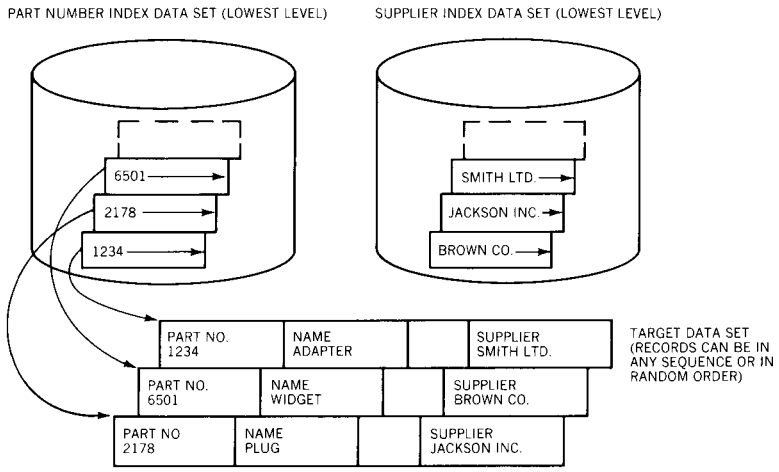


Figure 10 DTMS data base structure



cation programs in DPPX, not just those that run under the Transaction Processing Manager. After studying a number of possible data base organizations, we decided that an indexed data structure would allow a wide spectrum of transaction, interactive, and batch applications. An indexed organization would also be compatible with COBOL and could be implemented in a reasonable period of time. Figure 10 is a simplified depiction of the chosen structure.

organization

The target data set contains the user's data records. Each index data set contains a tree-structure index[4] that provides a pointer to the target data set record (or records) that contains a given value of the key field. There is one index for each key, up to a maximum

of eight indexes per target data set. The indexes are updated whenever a target data set record is inserted, deleted, or altered in such a way that a key field changes.

data base manager

Records are always processed by key, either randomly or sequentially. We decided not to allow reference to the records by relative record number (the DPPX interface used within the Data Base Manager itself) so as to avoid interference with the data recovery mechanism.

indexed access method

The target data set and index structures are the same as those for the DPPX/BASE Distributed Indexed Access Method (DXAM). Also, the programming interfaces to the Data Base Manager are compatible with those of DXAM. When DTMS is installed on a DPPX/BASE system, two DPX/BASE control blocks are expanded with the Linkage Editor to add identifiers for the Data Base Manager. Thereafter, the operating system deals with the Data Base Manager as with any other access method. A program can be written to process data bases and indexed data sets interchangeably, or debugged using indexed data sets and then run in production using data bases. To smooth the transition, we have provided the ability to convert most indexed data sets to data bases simply by identifying them to the Data Base Manager and describing their processing requirements.

The Data Base Manager operates a central buffer pool and record locking structure for all data base users, regardless of where in the operating system they are running. This reduces the total amount of storage required for buffers and the probability of data integrity loss.

Data integrity has been one of our major concerns in the design of DTMS. Complex data structures and elaborate search mechanisms could be implemented later, if customers indicated a need for them. Data integrity, though, is so fundamental to Data Base Manager design that it is very cumbersome to retrofit it to an existing product that must continue to serve existing applications and data bases compatibly.

levels of data recovery

In addition to such integrity measures as record locking and protected buffer pools, we decided to allow three levels of data recovery. The first is none. That is, there are some cases for which one may want to use data base management services, such as central buffer pools, but not be concerned about the integrity of the records, for example, when the data base is temporary scratch space. In that kind of situation, one does not want to pay the record keeping price that data recovery involves.

The second level of recovery is *resettability*, by which we mean the ability to recover from an application program error and back

out all data base changes made by that program. In the case of a system problem, we back out all changes that had been made to resettable data bases by work in flight at the time of the failure. Use of resettability assumes that neither the data set we use for recovery nor the data bases themselves will be physically destroyed.

The highest level of recovery is *recreatability*. At this level, DTMS takes the precautions required for resettability, and also maintains an audit file of all changes to any recreatable data base. The audit file can be used in conjunction with a previous dump of a data base to recover from physical I/O device problems.

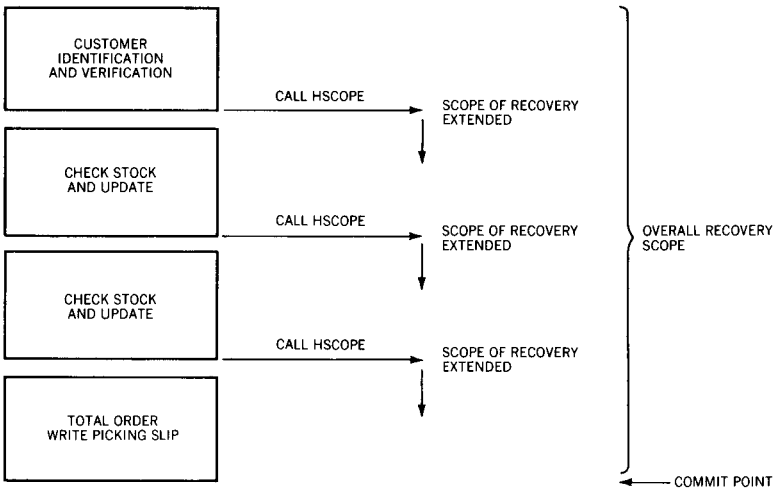
The recovery mechanisms can be used for more than just disaster recovery. In the APEX scenario, for example, the order entry transaction updates the inventory at the time each item is ordered. Obviously, if anything goes wrong at that point, the inventory data base is out of step. However, use of the resettability feature permits recovery, as follows.

Whenever a data base record in a resettable data base is updated, it is *locked* by the Data Base Manager. No other program can read or update the locked record until this program is finished with it. In the case of APEX, the terminal is not really finished with any of the inventory records until the customer gives his final acceptance of the order and its price. To allow the application to keep control of its updates, we have given the programmer the ability to request extension of the *scope of recovery*. The scope of recovery is simply the series of segments about which DTMS retains enough information to allow the resetting of all data base updates and any CREATX requests that have been made.

In the APEX case, as Figure 11 shows, the application extends the scope of recovery by calling the hold scope (HSCOPE) service at the end of each segment except the last. DTMS locks each record as it is changed and keeps accumulating back-out information in the reset information data set until a segment ends successfully, without requesting that the scope of recovery be extended. When that happens, DTMS releases the locks on all updated records and recycles the space used for the reset information. This is called the *commit point*. It is also the point at which any CREATX requests are released for execution. After the commit point, there is no automatic way to back out the results of this particular conversation.

We also allow the program to commit or reset explicitly all recorded data base changes that have occurred within this recovery scope. The reset operation is used in APEX if the customer cancels the order because the price is too high or some item is out of stock. When the application program receives a reply indicating

Figure 11 Use of hold scope (HSCOPE)

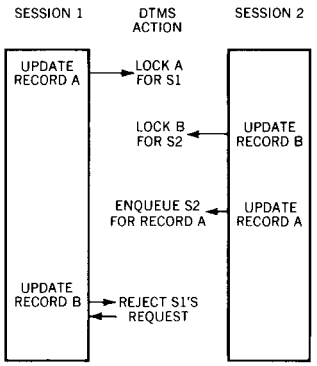


cancellation, it simply calls the reset service to return the data bases to their state at the start of the conversation.

deadlock

The locking mechanism, of course, creates the possibility of a deadlock, shown in Figure 12. Session 1 and Session 2 have updated one record each (not necessarily in the same data base). When Session 2 requests the record that Session 1 has updated, Session 2 is added to a queue to wait for Session 1 to complete and release the record. When Session 1 requests the record that Session 2 has updated, DTMS cannot put it in a queue because that would leave both sessions perpetually in a wait state. In this situation, DTMS detects the incipient deadlock—including a deadlock that involves more than two users—and gives Session 1 an appropriate return code. The Session 1 application program can then either terminate abnormally, which requires the terminal user to re-enter his request, or Session 1 can request a restart. If the decision is to restart, the Session 1 data base changes are backed out, its locks are released (allowing Session 2 to proceed), and Session 1 is put into a wait state until the record it asked for is unlocked. The Session 1 application program is then restarted from the beginning. The restart capability is limited to the current segment of the conversation, however, because that is the only program for which DTMS has the initiating message. If the deadlock involves a lock acquired by an earlier segment, the resolution has to be by resetting the whole conversation and having the Session 1 terminal user re-enter his request from the beginning.

Figure 12 An example of deadlock



DTMS program structure

In designing the program structure of DTMS, we were striving for the most economical implementation of the required functions we

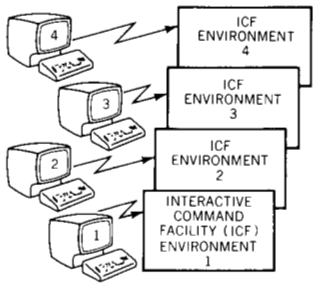
had identified. We considered briefly the possibility of seizing a large block of storage and a set of terminals and writing our own routines to manage those resources independently of the underlying operating system. That is expensive, takes up space for duplicate code, and makes it difficult to provide services to non-transaction programs. Since we were coming in at the start of the new DPPX operating system, we had the opportunity to design DTMS to work cooperatively with that operating system. We decided to make maximum use of the functions of DPPX and to make DTMS interfaces compatible with those of DPPX wherever possible.

environments

We started with the fact that the basic element of resource allocation in DPPX is as an *environment*. An environment is somewhat analogous to an OS region, a DOS partition, or an MVS virtual memory. An environment holds real storage, I/O devices, and access to data sets. It may have one or more tasks—which DPPX calls *threads*—running against those resources.

In the Interactive Command Facility (ICF) of DPPX, a separate environment is created for each terminal as it logs on, as illustrated by Figure 13. This is a desirable structure for some purposes, such as program debugging, but we did not believe that each transaction terminal would require a distinct environment. An environment, even though idle, still ties up all the resources allocated to it.

Figure 13 DPPX environment structure for terminals



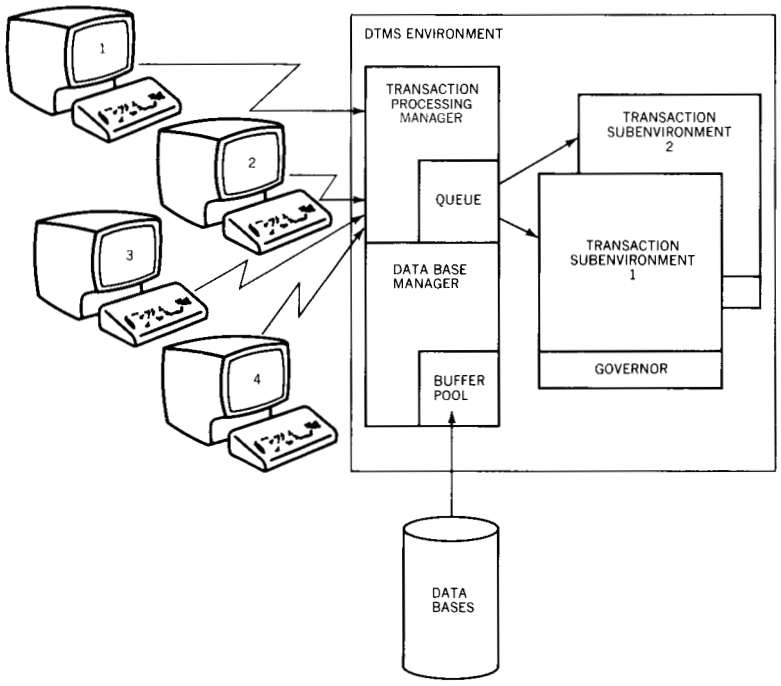
The structure of DTMS is shown in Figure 14. DTMS has an environment of its own, which persists as long as DTMS is running. The DTMS Transaction Processing Manager and Data Base Manager run in that environment. Within the DTMS environment are several subenvironments in which the transaction application programs execute. Normally, these transaction subenvironments have different DPPX address spaces from the DTMS environment and from each other. This isolates application program failures to one environment, and protects the DTMS buffer pool and control blocks from interference and unauthorized access.

the governor

To control the flow of work through the transaction subenvironments, we designed a program called the *Governor*. Whereas most of DTMS runs in the DTMS persistent environment, the Governor runs in the transaction subenvironment, and represents DTMS there. The Governor receives directions from the main part of DTMS as to which application program to execute. It loads and transfers control to that program, and control is returned to the Governor when the program ends.

In its own environment, DTMS keeps queues of messages that have been received from the terminals. When the Governor of an environment indicates that it has executed the requested appli-

Figure 14 Environment structure for DTMS



cation program, DTMS takes the next request from the queue and passes it to the Governor. If there is no request ready to be processed, that subenvironment simply waits for work.

transaction subenvironments

A key point is that these transaction subenvironments are true DPPX environments. All DPPX services that are available to a program running in an Interactive Command Facility or Batch Queue Manager environment are available to that program in a transaction subenvironment through the identical interface. DTMS also allows the execution of Interactive Command Facility commands by terminals logged on to the DTMS transaction processor, although some of them may not execute very efficiently there, because they are not segmented conversations. Nevertheless, an application programmer checking out a transaction does not have to log off DTMS and log on to the Command Facility just to use a command.

Because they are designed to work with programs that run in standard DPPX environments, many DTMS services need no additional code to accept requests from any environment in DPPX. Data Base Manager services, the CREATX service for generating a transaction, and others that are meaningful to nontransaction programs are available in any DPPX environment. Facilities that are

meaningful only for transactions, such as the transaction scratch pad, are restricted to transaction subenvironment users.

We make maximum use of DPPX operating system services in our terminal support as well. In fact, it is not strictly accurate to refer to DTMS as supporting terminals at all. Unlike a conventional data communications product, DTMS has no device-specific terminal I/O code. It deals strictly with the DPPX Presentation Services layers (the terminal access methods that provide a logical I/O interface to the terminals). No terminal is permanently reserved for DTMS. If a terminal logs on to DTMS, it is a DTMS terminal for the duration of that session. If it logs off DTMS and then logs on to the Interactive Command Facility (ICF), it becomes an ICF terminal. Since DTMS is completely session-oriented, we included a service that allows one terminal to cause another to be logged on, for example an output-only terminal. (This service is used in the APEX application to start the session with the terminal printer that writes picking slips.)

other DTMS features

Our objectives of continuous operation and efficient use of additional storage caused us to add a number of tuning functions to DTMS, and affected the design of other areas. To speed up the processing of very frequent transactions, such as order entry, the transaction descriptors of those transactions can be kept resident in storage. Also, data bases that are used frequently can be left in a partly opened state (i.e., the control block structures are not destroyed when the application program ends) so that full open processing is not necessary each time a transaction accesses such a data base.

Data bases and transactions can be defined to DTMS while it is running. The definitions can also be altered and deleted without stopping DTMS, although the particular data base affected may have to be deactivated temporarily while the change is made. Data bases can be deactivated, dumped for backup, or even recreated from a previous backup and the audit file, and then reactivated without stopping the Data Base Manager.

The Data Base Manager and the Transaction Processing Manager can be started and stopped independently of each other. This allows, for example, the Data Base Manager to run twenty-four hours a day, while the Transaction Processing Manager runs only during business hours. The space that the Transaction Processing Manager and the transaction subenvironments normally occupy is then available for large batch programs that run overnight.

The number of transaction subenvironments can be varied without stopping the Transaction Processing Manager. This allows additional subenvironments to be activated to process peak period workloads without leaving idle environments at other times.

Status commands are available to determine the number of transactions that are queued and the state of the transaction subenvironments.

The size of the Data Base Manager's central buffer pool in the persistent DTMS environment varies with the number of data bases that are active and the number of concurrent data base references that are taking place.

To simplify installation and startup of DTMS, most of the setup information required is kept in small data sets called *command lists*. Although these command lists may need customization for unusual user requirements, the default command lists that are supplied with DTMS allow the system to run effectively immediately after installation.

Summary and concluding remarks

In DTMS, we have attempted to construct a Transaction Processing Manager and a Data Base Manager that are optimized to the projected needs of the true transaction application. We envision such an application as a sequence of simple programs, each of which processes a single message from the terminal and then prompts the user to continue the conversation.

The Transaction Processing Manager receives each message from the terminal and assigns it to a transaction subenvironment for execution. The Data Base Manager provides the application program with data base records from its buffer pool. As the application program is executing, it indicates to the Transaction Processing Manager the services to be performed after it completes, including the message to be sent back to the terminal. When the application program ends successfully, all of its requests are honored and its data base changes are committed. The transaction subenvironment is released to process messages from other terminals. The specified message is sent to the terminal, and DTMS waits for more input.

We believe that the design of DTMS represents a good trade-off of storage, speed, and function, and that it provides the IBM 8100 Information System application programmer with the tools to build segmented conversation applications simply and reliably without disallowing the more complex program structures when necessary.

ACKNOWLEDGMENTS

I am indebted to my colleagues at the Santa Teresa Laboratory whose contributions this paper describes. In particular, my thanks go to Wayne R. Maple, Chief Programmer of DTMS, who

corrected my technical mistakes (and tried to correct my stylistic ones), and who has directed DTMS design from its inception.

CITED REFERENCES

1. W. C. McGee, "The information management system IMS/VS," *IBM Systems Journal* **16**, No. 2, 84–168 (1977).
2. D. J. Eade, P. Homan, and J. H. Jones, "CICS/VS and its role in Systems Network Architecture," *IBM Systems Journal* **16,** No. 3, 258–286 (1977).
3. J. E. Siwiec, "A high-performance DB/DC system," *IBM Systems Journal* **16,** No. 2, 169–195 (1977).
4. D. Comer, "The ubiquitous B-tree," *ACM Computing Surveys* **11,** No. 2, 121–137 (June, 1979).

The author is located at the IBM General Products Division Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, CA 95150.

Reprint Order No. G321-5110.