A Generalized Strategy for Building Resident Database Interfaces

Final Report
for
Contract NAS5-30304

May 12, 1990

by
Marsha Moroh & Ken Wanderman
Ken Wanderman & Associates, Inc
160 Bement Avenue
Staten Island, NY 10310

prepared for
NASA Goddard Space Flight Center
Greenbelt, MD 20771

File: \kwai\nasa\report8\report8

Table of Contents

Table of Figures

Preface

Objectives

In Phase I of this project, we developed a theoretical method for constructing interfaces of database management systems (DBMSs) to host distributed heterogeneous database management systems. In Phase II, we sought to incorporate the strategies and techniques learned in Phase I to streamline the process of creating interfaces. Our ultimate goal is the creation if the Database Interface Design System (DIDS) to totally automate the interface creation process. We envision this system as an important software product for the 1990s.

While full implementation of DIDS was beyond the scope of this project, we had the certain technical objectives as part of the Phase II effort. The major objects are outlined as follows:

Interface Guidelines

Our initial goal was to examine a large number of commercially available database management systems and to develop a hand book for builders of interfaces. These guidelines are a step by step approach to constructing interfaces. A subsequent goal is to construct a totally automated system of building interfaces using the guidelines.

Inner vs. Outer Interfaces

In the course of examining various commercial DBMS we determined there were two distinct approaches to building interfaces; viz, **inner interfaces** in which the interface software is embedded directly into the code of the heterogeneous distributed host system, and **outer interfaces** in which the interface software is external to the host and a series of templates are used to execute the database commands.

Refinement of Interface Templates

The original templates were designed in Phase I of this project. The symbols in these templates, which represent DBMS-specific items for each resident interface were simple names to be replaced by strings in the actual interface. Our objective in Phase II was to refine these templates as a result of testing, and expand the template syntax to encompass more complex data structures and integrate these templates into our overall system.

Front-end Processor Software

An essential part of our automated system to build interfaces is a user session to capture the DBMS-specific items to be used in the template. The solicited items are rarely just a single word, and can be quite complex in structure. In particular, for each item solicited in the user session, not only its value but its meaning or semantic use must be captured. A major objective was to develop software to capture the information and store it for use by the interface driver software. Our original idea was to code this software in a language such as C, but further research revealed that an expert system was more efficient. This front-end processor became known as the Template Generator Software as we developed it.

Interface Driver Software

The interface driver software is the engine of the entire project. The objective in building this program is to develop a process to extract the information captured by the user session and build database management system dependent templates from the skeleton templates and the information captured by the front end processor.

Building of Interfaces as Test Cases

The final objective was to apply our methodology and actually build some database interfaces to the **DAVID** system. The actual interfaces constitute the required deliverables for this project.

Scope of work

Personnel, Materials & Facilities

Ken Wanderman & Associates, Inc. has provided the personnel, materials, and facilities necessary to develop a generalized strategy for building resident database interfaces as stated in our proposal number 86-I-II 07.09-6211. In particular, we have purchased a SUN 4, Sparcstation I computer to do all the development work on. The only work not performed on our premises is that work as done by subcontractors as specified in article C-2 of the contract.

Deliverables

We have delivered considerably more than the contract specifies. In particular, the contract calls for the construction of interfaces to:

UNIFY
ORACLE
INGRES
R BASE

We have included these interfaces on the main deliverable tape except for R BASE which runs only on micro computers and thus awaits the transport of **DAVID** to personal computers. The R BASE interface is included on a separate diskette.

During the course of the project, we have come to believe that other interfaces would be of considerable more value to NASA. Therefore we have included interfaces for the following.

FILE
FITS
FOCUS
IDMS

In particular the general FILE interface and the FITS interface will allow NASA scientists to immediately do important work.

In addition we are delivering the source and executable code for the Template Generator Software portion of DIDS. These programs run on PC compatibles and are found on 3.5 inch diskettes.

Conclusions

We have developed an interface method, which for large classes of commercial resident DBMS's's, will yield usable interfaces to the **DAVID** system and by extension to any heterogeneous distributed DBMS. In addition these interfaces can be built within a few minutes or hours at the most. See the User Guide Lines below for a detailed account of how to do this. We anticipate that this will encourage greater exchange of data among NASA scientists.

Report Body

Overview

There are a multitude of database formats in use at NASA today. There are a variety of commercial and in-house database management systems, each supporting a number of databases. In addition, there are large quantities of data stored in sequential files; data which can only be accessed by specially written programs. This situation has led to difficulties for scientists trying to access information stored in a different format than their own. The **DAVID** system was developed at NASA to act, in part, as a central database management system. One could "hook-in" to **DAVID** various database management systems and **DAVID** could translate instructions and data from one DBMS to **DAVID**, or from one DBMS to another via DAVID. This "hooking-in" process is called a **resident interface** and is at the heart of this project. Prior to this project, the process of building an interface was long and tedious and more work than most scientists were willing to do to obtain data. What we have done is to streamline this process, first with a step-by-step guide to building interfaces and secondly with the beginnings of a totally automated system for building interfaces. We have tested our strategy on a number of commercial products and have found it most workable. The remainder of this report details various parts of our system.

Work Performed

Inner Interfaces

As part of this project, we developed a technique of building *inner* interfaces between resident and host DBMSs, and investigated the feasibility of our technique by designing and building four representative interfaces to resident DBMSs from **DAVID**. We are pleased by our results; they point us in the direction of a prototype for the design of fast, easily-developed interfaces in the future.

We define an inner interface to a host DBMS as one in which the resident DBMS facilities are used only for access at the lowest level, i.e., the I/O of a single entity (record or table). All other database operations are performed by the host on the data after it has been moved to buffers belonging to the host DBMS.

This low-level-access-only capability is used in the construction of interfaces to 1) DBMSs which support a high-level language interface, such as FORTRAN or C, but provide neither a query language nor access at the path level; and 2) file systems, which contain no database processing routines of their own. The former condition occurs in many network and hierarchically organized DBMSs; we describe interfaces to IDMS and FOCUS below, both of which possess these characteristics. The latter set of circumstances is described in our interfaces to FITS and to general files, both described below.

In the inner interface technique, requests for information and transactions submitted via the host are processed by the host until the request is reduced to a call for a single row of a single table. At that level, the request is handled by the DBMS under which the data is sitting. If the request includes a boolean evaluation, the boolean is performed by the host on the data after it has reached the host buffer. The routines in an inner interface include a set of access routines, and a set of opening and closing routines.

Accessing and Transaction routines. There are a set of routines that perform low-level access on records (tables) of the database which form the building blocks of the interface.

First. The first row of a table of the resident database is read into a buffer of the host DBMS. Any boolean evaluation is done in the host buffer.

Next. The next row of a table of the resident database is read into a buffer of the host DBMS. Any boolean evaluation is done in the host buffer.

Insert, Delete, Update. A row of data in the resident database is inserted, deleted, or updated.

Connect. A row of data which has just been inserted into a table in the resident database, is connected to the proper parent. This routine only has applicability in a database system of network structure.

Disconnect. A row of data is disconnected from the designated parent and, if this is the last parent, the row is deleted from the database. This routine only has applicability in a DBMS of network structure.

Parent. Given a row of a table, and the name of a parent table of that table, this routine returns a row of data from the parent table. This routine only has applicability in a database of network structure.

Assigning (opening) and *Deassigning* (closing) Resident Databases. Before table-row routines can access a resident database, certain initialization functions must be performed. Similarly, after table-row access to a resident database has occurred, some termination functions must take place before the resident DBMS is exited. These include:

Assign Database. This routine performs "housekeeping" functions necessary to allow the resident to communicate with the host: it establishes the necessary data buffers and variables needed by the resident, logs on to the resident DBMS, provides the necessary security, and opens the resident database.

Deassign Database. This routine performs "housekeeping" functions necessary to terminate host interaction with a resident: it closes the resident database, logs off of the resident DBMS, and deallocates any special data areas set aside by Assign Database.

In our research, in order to investigate the feasibility of the inner interface approach, we designed and built four interfaces. These are described below. The first two, FITS and general files, fall into the category of file systems, i.e., they have no DBMS capabilities at all. The last two, FOCUS and IDMS, are both full-fledged DBMSs; one (FOCUS) hierarchical in organization, one (IDMS) network in organization. Although IDMS has an on-line query facility, OLQ, and FOCUS has a 4th generation front-end, we bypassed these facilities to implement these interfaces.

We chose FOCUS and IDMS because each was representative of a class of DBMSs, i.e., network and hierarchical DBMSs. We felt that in solving the problems inherent in building inner interfaces to systems with each of these kinds of data organizations, we would learn a lot about how to deal with other DBMSs of similar organization.

We chose the FITS and general file systems to interface because they represent a large portion of the data currently being held at NASA for analysis, and we felt that interfacing them to a DBMS would make a real contribution to the scientific community.

All four of the inner interfaces are described in the text below.

FITS interface

FITS (Flexible Image Transport System) is a file interchange standard developed by a group of scientists interested in the exchange of astrophysical data. It provides a simple but powerful mechanism for the unambiguous transmission of large data arrays and of catalogs describing that data. The FITS format has been adopted for the transmission of astronomical image data by several large observatories including the Very Large Array, the Westerbork synthesis telescope, the Kitt Peak observatory and the Anglo-Australian observatory. It is fast becoming a standard for the cataloging and transmission of astronomical images and their catalogs.

FITS files consist of a set of header records describing the data in the file, followed by the data itself. The header information is used by application programs which read the data. Figure 3 contains a typical FITS file header for a star catalog. It should be noted that by "header", we do not mean operating-system-specific information associated with a file; file access routines deal with those. We are referring to information contained in the first few records which describe the data in the remaining records. The header information is strictly for the interpretation of the data.

The FITS file system is not a DBMS. The access and interpretation of FITS data is performed by a set of FORTRAN routines which are either "home grown" or shared by other sites among the astrophysics community. There is no query facility supplied for FITS data and catalogues; nor is there a browsing tool. Users of FITS cannot interface FITS data with that of any other file or database system.

We have developed an interface technique for FITS files, and have used the technique to connect the FITS file system to the **DAVID** system. The effect of this is to provide FITS users with a SQL-like query language front-end, so that they can perform selection-projection operations on FITS data. They can select certain records and fields from those records, applying boolean conditions on the operations, and forming new files or databases for the results (the results can be stored as FITS files, as **DAVID** databases, or even represented in another database

system, such as ORACLE or INGRES). More important, data from FITS files can be JOINed on some common trait with other FITS files, with **DAVID** databases, and with data from other commercially available DBMSs, thereby giving scientists access to data never before available.

The components of a FITS interface are described below.

Defining and Deleting FITS Files through an Interface

These routines define and drop FITS files through the host DBMS (here, the **DAVID** system), and establish and break the connection between the host DBMS and the FITS file.

For a FITS file to be "defined" to the host DBMS, a database definition must be supplied to the host's database directory. For the file to be "dropped", the definition must be expunged from the host. In neither case is the data changed in any way.

Our FITS interface treats the FITS file header as a database schema, from which the description of the attached data can be derived. Our software reads the FITS header; then translates it into a host DBMS definition. Any information in the FITS header not used by the **DAVID** system is preserved in the **DAVID** definition by being represented as a comment in a special format in the **DAVID** definition. That way, if the user wants to create another FITS file containing all the attributes of the first one by issuing a "SELECT * " (select and copy all information in the file) query, the attributes of the first file can be obtained from the **DAVID** definition.

After our software constructs the required definition, it installs that definition onto the **DAVID** system. The host then "knows about" the FITS file, and it can be processed like any other database: questions and reports in the form of queries can be solicited from it, the browsing tool can read it a record at a time, and data can be inserted into it via a query.

Query and Transaction Processing through an Interface

The records of a FITS file are accessed through the host DBMS a row at a time, using the first and next FITS file access routines built by us as part of the

file interface and linked into the **DAVID** system. To process a query on a FITS file, the query must first be decomposed into its primitive parts, optimized, and translated into a set of subroutine calls to be issued on the file.

At the lowest file-access level, FITS file access routines are identical to those of general files, described below.

File interface

A file can be defined as a related collection of data stored on some external electronic medium such as a disk or magnetic tape. Files in general differ from the description of FITS files, above, only in that they do not possess a header containing their descriptions, or record layouts. The descriptions of files generally exist in some documentary form.

Once interfaced to a DBMS system such as **DAVID**, these file could be treated as if they were databases: they could be queried, selection-projections can be performed on them, and they can be joined with information from other database systems. They can also be converted into other representations, such as **DAVID** or another DBMS.

Defining and Deleting Files through an Interface

These routines define and drop files through the host DBMS (here, the **DAVID** system), and establish and break the connection between the host DBMS and the file.

For a file to be "defined" to the host DBMS, a database definition must be supplied to the host's database directory. For the file to be "dropped", the definition must be expunged from the host. In neither case is the data changed in any way.

The problem of creating a database "schema" for a general file is approached differently than it is for FITS; instead of decoding an existing header, the system must rely on the user to provide the description of the file. This can be done in one of two ways in our system: via a DEFINE statement in **DAVID**'s query language, or via an interactive session with **DAVID**'s browsing tool. In either case, the submission of the definition triggers an interactive session with the user, in which he/she is asked to provide any informa-

tion required by the host system but not provided by the definition (in the case of **DAVID**, such information as column numbers for the individual fields, and data types not available on the **DAVID** system). After the definition is solicited, software constructs the required definition.

Query and Transaction Processing through an Interface

The records of a file are accessed through the host DBMS a row at a time, using the first and next file access routines built by us as part of the file interface and linked into the **DAVID** system. To process a query on a file, it must be decomposed into its primitive parts, optimized, and translated into a set of subroutine calls to be issued on the file.

Figures 4 through 8 illustrate the use of the file access routines to access a general file. The examples are written in C, for a file on a SUN 4 computer under UNIX. Figure 4, assign database, performs the preliminary tasks on the file so that it can be read through the host DBMS. Note that instead of opening a database, this routine opens the file. Figure 5, table-row first and table-row next, provides the host with the capability of reading a record at a time of the resident. Figure 6 provides for inserting one record a time and figure 7, Deassign File, illustrates closing the file and performing any other necessary housekeeping routines after file processing is finished.

The IDMS Database Management System Interface

The IDMS Database Management System, developed by the Cullinet Corporation, is a network-structured DBMS that runs on a variety of manufacturers' equipment, including the IBM 327x series, and the DEC VAX series. Later versions contain a query language (OLQ) as well as the traditional access methods, via IDMS calls embedded in an applications program written in PL/I, COBOL, C or assembler.

IDMS was chosen for our research because it is representative of a class of DBMS systems, namely those that conform to the standards set by the Database Task Group of the CODASYL Committee for database systems. These DBTG systems, as they are often known, support network

database organization, and have 3 languages: a schema data description language (DDL), a subschema data description language (DML) and a data manipulation language (DML). The schema languages are distinctly COBOL in flavor, since that was the committee's orientation (they were an outgrowth of the original designers of COBOL) and the data manipulation language consists of calls to be embedded in a language such as COBOL or PL/I. Compared to the query facilities of many of today's languages the DML seems quite cumbersome; however, it was the perfect vehicle for building an inner interface to a host DBMS. N.B. Because the **DAVID** system is not yet available on the platform on which IDMS runs, this interface puts the data into the host buffers, from which it is displayed.

Defining, Installing, Dropping and deleting IDMS databases through an interface

In IDMS, as in other resident DBMSs, we distinguish between the Define operation and the Install operation. *Define* creates a new IDMS database via the host system. To create the definition, the user, sitting at the host DBMS front end, submits it in the host query/definition language. Software from the interface then translates the definition into an IDMS schema and subschema representation and submits it to the IDMS DBMS, where it defines a new database in IDMS. The corresponding host definition submitted by the user is then added to the collection of database definitions belonging to the host DBMS.

Install, on the other hand, is used to connect existing IDMS databases to the host DBMS, so that they can be processed via the host. In the Install operation, the IDMS schema is read, and software creates the corresponding host database definition; then it adds the definition to those of other resident and non-resident DBMSs so that the host software can access the IDMS database system. So the Install operation is used when a user wants to be able to access an existing IDMS database through the host DBMS; the Define operation is used when the user is creating a new IDMS database via the host DBMS. It may appear as though the Define operation would be used less often than the Install, but in fact the opposite is the case: whenever a user issues a query on a database stored in the host or in

any other resident DBMS and requests for the result to be stored as an IDMS database, the result is a call to the IDMS database definition software.

The *Drop* operation does not affect the IDMS database being deleted; it merely breaks the connection between that database and the host. We can continue to do processing operations on the IDMS database; but they can only be IDMS operations; host database processing is no longer defined on this database.

The *Delete* operation actually erases the IDMS database; in addition, it *drops* its schema from the DAVID directory.

Query and transaction processing through an inner interface

As in the case of FITS and general files, and all other inner interfaces, the minimum set of operations needed to process queries and transactions on an IDMS database via the host DBMS is: *open, close, first, next,* and *insert*. These operations will be discussed below.

IDMS was designed for database access via program (COBOL or PL/I). Each query and transaction normally is performed by writing, compiling and linking a separate program specific to that query and transaction. The program contains references to a specific database, and specific records, sets (parent-child record pairs), and fields in that database. For example, the declaration section of the program contains declarations for the schema and subschema of that particular database, and there is a set of INCLUDE statements, one for the database itself. The IDMS preprocessor reads and deciphers these statements, and replaces them by the corresponding calls to IDMS external functions.

Our inner interface approach requires generic routines with no database-specific variables; otherwise every routine would have to be compiled and linked for every query, and query processing would be far too slow. To get around this problem, our solution was to bypass the IDMS preprocessor, and replace the IDMS code by conventional calls to a linked procedure called IDMS. In this way we communicate with the IDMS DBMS directly through the IDMS assembler procedure.

Examples of some of these preprocessor expansions are (we are assuming a schema called BCCIS45 with a child table called TEACHER; the fields of TEACHER are described in the declaration):

```
INCLUDE IDMS(BCCIS45-TEACHER).
```

is replaced by:

```
DECLARE 1  BCCIS45_TEACHER,5
TM_CIS45_TCHR_ID_NUM CHARACTER (6),5
TM_CIS45_TCHR_FULL_NAME,10
TM_CIS45_TCHR_LST_NAME CHARACTER (15),10
TM_CIS45_TCHR_FST_NAME CHARACTER (10),5
TM_CIS45_TCHR_SOC_SEC_NUM CHARACTER (9),5
TM_CIS45_TCHR_RANK_CD CHARACTER (1),5
FILLER0002 CHARACTER (7);
```

The preproccessor replaces the code:

```
INCLUDE IDMS(SUBSCHEMA_CTRL).
```

by:

```
DCL 1  SUBSCHEMA_CTRL STATIC BINARY,3
PROGRAM CHARACTER (8) INITIAL (' '),3
ERROR_STATUS CHARACTER (4) INITIAL ('1400'),3
DBKEY FIXED BINARY 3
RECORD_NAME CHARACTER (16) INITIAL (' '),3
AREA_NAME CHARACTER (16) INITIAL (' '),3
ERROR_SET CHARACTER (16) INITIAL (' '),3
ERROR_RECORD CHARACTER (16) INITIAL (' '),3
ERROR_AREA CHARACTER (16) INITIAL (' '),3
IDBMSCOM_AREA,5
IDBMSCOM (100) CHARACTER (1),3
DIRECT_DBKEY FIXED BINARY (31),3
DATABASE_STATUS,5
DBSTATMENT_CODE CHARACTER (2),5
DBSTATUS_CODE CHARACTER (5),3
FILLER0001 CHARACTER (1),3
RECORD_OCCUR FIXED BINARY (31),3
DML_SEQUENCE FIXED BINARY (31);
DCL 1  RECORD_NAME
BASED(ADDR(SUBSCHEMA_CTRL.RECORD_NAME))
STATIC INTERNAL,3
SSC_NODN CHARACTER (8),3
SSC_DBN CHARACTER (8);
```

In our strategy, we bypass the preprocessor, and replace the expanded calls by generic calls which can be parameterized, so that all information can be supplied to the interface routines at run time. So instead of a schema declaration with actual field names to replace the INCLUDE IDMS statement, we use a dynamic allocation with the fields supplied at run time. In a similar vein, all the expansions are established in such a way that no information is database-dependent.

Below we describe the same process with database access operations.

How do we open IDMS:

```
replace:
  INCLUDE IDMS(SUBSCHEMA_BINDS).
by:
  SUBSCHEMA_CTRL.PROGRAM = name;
replace:
  BIND RUN_UNIT.
by:
  SUBSCHEMA_CTRL.DML_SEQUENCE = SEQUENCE;
  CALL IDMS (SUBSCHEMA_CTRL,IDBMSCOM(59),SUBSCH-
EMA_CTRL
                    ,SUBSCHEMA_NAME);
  IF (ERROR_STATUS <> '0000') THEN DO;
    STATUS_CODE = ERROR_STATUS;
    GOTO END_STATUS;
  END;
replace:
  READY RETRIEVAL.
by:
  SEQUENCE := SEQUENCE + 1;
  SUBSCHEMA_CTRL.DML_SEQUENCE = SEQUENCE;
  CALL IDMS (SUBSCHEMA_CTRL,IDBMSCOM (37));
  IF (ERROR_STATUS <> '0000') THEN STATUS_CODE =
ERROR_STATUS;
```

The **open** operation for IDMS, then, consists of a set of generic preprocessor calls to which database-specific information is passed as parameters. It includes all the declarations of variables, statements to bring up IDMS, to open and ready the specific database and schema areas, and to establish the error-reporting mechanism, and the mechanism to retain currency information while navigating through the database, so that "give me the next record" has some meaning. It also sets up bindings to buffers into which the data will be transferred so that the host DBMS can pick it up.

When a user opens an IDMS database, the host system must establish a memory data structure for keeping track of the IDMS operations and their status. It also contains buffers for any data to be transferred to the host system. Each user wishing to access an IDMS database must open IDMS separately. Therefore there will be a separate memory data structure for each set of IDMS processing operations taking place. These memory data structures are used to communicate between the embedded IDMS processing calls, which are widely scattered through the host system, sometimes separated by layers of functions through which the structures must be passed. (In a totally IDMS environment, these problems do not arise; since each IDMS user is running a separate program, many of these data areas can be global.) All of these data areas had to be added to the collection of host DBMS data structures. Because it was clear that this situation will arise in all DBMSs of this class, and in some of other classes as well, these data structures were made as generic as possible, and only a pointer to one of them was stored in the **DAVID** cluster control area in the current implementation.

The *first* operation and the *next* operation both contain function calls to the corresponding IDMS external functions to correspond to the IDMS language calls **OBTAIN FIRST RECORD (name) SET (set_name)** and **OBTAIN NEXT RECORD (name) SET (set_name)**

The operations first and next have a different meaning in hierarchical and network database systems than they do in the context of relational database systems and files. In a "flat" database (with no parents or children) the meaning of "next record" is unambiguous. In database systems where there are child tables and parent tables, and particularly in network database

systems where there can be multiple parents for a single child table, the request for a "next" record must supply the identity of the parent of that record, and keep track of the current owner at all times.

The *close* operation contains IDMS function calls to close the database and check the error status of the preceding operations. It must also perform such "housekeeping" functions as deallocating the storage for the data structures used in processing the database. After a close has been executed for a particular IDMS database, no further processing of that database can occur unless another open command is first issued.

The code for the IDMS database access operations appears in the appendix.

FOCUS Interface

The FOCUS Database Management System, developed by Information Builders, Inc., is a hierarchically structured DBMS that runs on a variety of platforms, including an IBM personal computer and several mainframes. There is an upload/download facility, whereby database information can be transmitted from a PC to a mainframe and vice veers. FOCUS has no query language per se; however, a menu-driven 4th generation language front end is available on the PC version to aid the user in building reports and requesting transactions. FOCUS databases can also be accessed and updating using HLI (Host Language Interface) commands, a set of function calls which can be embedded in high level language programs.

FOCUS was chosen for our research because of the fact that it is a hierarchically structured DBMS which runs on a wide variety of computers; also because we feel it is representative of a class of database systems, i.e., those which support tree-structured data organization. Our interface did not use the menu facilities of FOCUS; rather it was constructed of calls to the FOCUS HLI routines for the retrieval of information. Those routines, which are generic for the processing of any database, can be linked into the host system a single time; then instantiated for a specific database via parameters passed to the routine. N.B. Because the **DAVID** system is not yet available on the platforms on which FOCUS runs, this interface currently puts the data being retrieved into the host buffers, from which it is displayed.

FOCUS databases are made up of a series of files, each containing a logical record of the database. The files (records) are connected by pointers. The interface loads HLI (the FOCUS host language interface routines) and accesses multiple files dynamically. In the interface, files can not be opened more than once even though HLI allows it. The number of files that can be opened concurrently is limited only by the amount of RAM memory.

Defining, Installing, Dropping and Deleting FOCUS Databases via an Interface

Like that of its counterpart, the network DBMS, the Define operation for a FOCUS database consists of translating a host database definition into a set of FOCUS definition calls that are then used to create a new FOCUS database (called a "master file" in FOCUS); the host version of the database definition is then entered into the host's database directory. The Install operation translates an existing FOCUS database definition to a corresponding host definition for the same FOCUS database. That definition is stored, as it is in the Define operation, in the host's database directory. The difference between the Define and the Install operation is that Define is for new databases; Install is for existing FOCUS databases to be connected to the host system.

For the Install operation, the FOCUS Master File Description (schema) is parsed; then the corresponding host definition is generated. Fields that do not hold data meaningful to the host (such as OCCURS COUNTER fields) are ignored in the translation process. A master file description contains file attributes, segment attributes and field attributes. The file attributes appear once in a master file description, and contain information about the physical file, which maps to the database name for the host. The segment attributes supply information about the parent relationships, which is what gives FOCUS its hierarchical structure. The field attributes correspond to the individual fields of records of any file or database. There are many parameters describing each field in a FOCUS database; those which are meaningful to the host are translated to the host data definition; those which are not are ignored.

The Define operation does the reverse: it creates a FOCUS master file description from the host database definition. This description will of necessity contain very few FOCUS field parameters; only those which can be defined in the host DBMS will be carried over; the result is a "vanilla" FOCUS master file description.

The Drop operation simply removes the host's version of the FOCUS database definition from the directory; the FOCUS database is left alone, but it can no longer be processed via the host. The Delete operation performs a Drop; it also erases the actual FOCUS database.

Processing Queries and Transactions via an Inner Interface

As in other inner interfaces discussed here, an inner interface between FOCUS and the host DBMS requires five major components: *open, close, first, next* and *insert*.

In order for the interface to deal dynamically with databases, it must maintain a memory data structure that contains data about the names, type, and length of each field in the database (a full view of the database is applied). These mapping tables are allocated from memory and stored in linked list form. Every opened file has a linked list, and every file is identified by an index of linked lists. These are referenced by the array list which is an array of pointers to lists. The main elements of a list are HLI system structures of the type INFOF, which is a structure that holds information about one field in the file. These structures are linked together by a structure that has two pointers: one pointer to INFOF structure, and a second pointer to the next structure.

Each file has an entry in an array of files called filename[], and a flag, open_flag[], that has a true value when the file is open. For each file a work area is allocated dynamically as a character string. When a retrieve operation is done such as get first record, the data that HLI stores in the work area can be retrieved by using the mapping tables stored in the linked list. When a file is closed, all the memory that was allocated for it is released.

So the **open** inner interface operation invokes FOCUS (N.B. The PC FOCUS product can be made to become memory-resident when it is invoked. So the more likely scenario is to make it continuously memory-resident; another option is for the *open* to make it memory-resident. In this case it will reside in memory only until the *close* removes it. In either case its status must be checked; attempts to start up an already memory-resident process can lead to unpredictable results!

The *open* operation also performs some house-keeping chores. The following is a representative set of these chores: it allocates a work area for the data being retrieved, it builds a control area for the opened database by creating field mapping tables; it allocates a file control block for the database

As in the network DBMS described above, *first* and *next* have different meanings in hierarchical database systems than they do in relational systems or flat files. Both operations call upon the generic HLI function call "get_seg", which accepts as parameters the information

database operation (first, next, etc.)
database name
segment to be retrieved
parent of segment to be retrieved

and retrieves the required information. The database name, segment name and parent segment name are supplied by the host system as the request for data is issued by the host system. The "currency", or the proper parent for each segment, is kept in the work area allocated at *open* time. The choice of database operation is determined by the calling host routine.

The **close** operation closes the focus database and, if FOCUS itself is not to be memory-resident, removes it from memory. It also deallocates the many work areas and temporary files set up to process the FOCUS database.

Summary and Evaluation of the Inner Interface Method

The inner interface method proved to be a workable method for building interfaces; indeed, it is the *only* feasible way to build interfaces to those DBMSs with no

query or path access facilities. It gives great power to a resident DBMS -- all the power of the host DBMS -- with just the coding of a few routines, and is, from a computing resources point of view, by far the more efficient method for an interface than the alternative method, by which a batch job containing data-base specific requests is created, then compiled and linked, then executed. We rejected the latter approach out-of-hand when a few preliminary tests showed it to be intolerably slow.

Moving most of the processing that the resident database performs on the records (such as boolean evaluation) to the host from the resident will generally improve performance over that of a similar operation on the resident DBMS alone; the host system is designed for faster processing than many of the older DBMSs.

However, there are difficulties with the inner interface approach. For each interface, a total of about 15 routines must be compiled and linked to the host system, and the resulting system must be debugged and tested. The potential for error is great, and the interface builder must be an experienced programmer. A contrasting approach is presented in the next section, in which we discuss outer interfaces.

Outer Interfaces

For database systems with query languages, a very fast way to build interfaces is using the *outer* interface approach. In this method, the *Define, Install, Drop and Delete* work the same way as the same operations in the inner interface method. However, the query and transaction processing interface component is quite different.

When queries are submitted to the host system, they are parsed; then translated into the query language of the resident system. The query is executed on the resident, and the results are captured to a file, which can then be read by the host system by a single standard inner interface.

This method of building interface is particularly attractive because it is so straightforward: we have designed an expert system to facilitate their construction. Using this system, a person familiar with a resident DBMS can build an interface in a short amount of time, sitting at a personal computer. No programming is required.

The outer interface construction process uses an approach we call the *template* approach; its components are explained in detail in a subsequent section. Briefly, here's how it works: Each database operation for every DBMS is depicted in a block of text called a template. The template contains DBMS-specific commands, such as CREATE, or SELECT, or OPEN, and symbols for data-base specific information: for example, the name of a table would be replaced by the symbol @tablename.

Here is a sample template for creating a single table in the ORACLE database system:

```
CREATE TABLE @tablename (
    @BEGINFIELDS
      <,><><> @fieldname @fieldtype @fieldlength
    @ENDFIELDS )
```

The symbols beginning with "@BEGIN..." And "@END..." Connote loop structures for repeating text; the symbols <><><> connote left and right delimiters and separators for repeating text.

At execution time, the template is filled in by replacing the symbols (those beginning with @) with database-specific information. The result is an executable module which performs the required database operation.

The template approach is used for all outer interface components; it is used also for the inner interface components *Define, Drop, Install* and *Delete*. Only the inner interface database access and transaction processing operations *first, next, open, close,* and *insert* have to be individually coded, since they have so many idiosynchratic requirements.

Summary and Evaluation of the Outer Interface Building Approach

Outer interfaces have a clear advantage over inner ones in their construction; an outer interface requires no programming, and can be designed by a person knowledgeable about the resident DBMS to be interfaced but with no knowledge of the host DBMS. The designer can design the entire interface sitting at a personal computer.

There are two disadvantages to the outer approach: one is that it can only be used to design interfaces with DBMSs that have query languages. The other is that there is a great deal of overhead associated with the fact that the

data is captured to a file before it can be processed by the host system; in the case of very large databases, the storage and processing requirements to do this can be prohibitive.

A large portion of our research has been devoted to the design and construction of an expert system to aid the designers of outer interfaces in building the templates; this simplifies the process of interface building even more, because the builder has a great deal of help in his/her task. The expert system, which we have called TGS (the Template Generator Software) will be described in the next section.

There is also a set of Interface Guidelines (see the section of the same name below). This document is intended to serve as an aid to the builder of any interface, inner or outer. It "walks" the user through the process, from the determination of the type of resident interface is being built, through the design, construction and testing of the interface routines. The document can be used without the expert system, or as a supplement to it.

Front-end Processor

The first of the two major pieces of software involved in the DIDS project is the front-end processor which we have called Template Generator Software (TGS). When a user wishes to build an interface for a given DBMS, he/she must first run the front-end processor to tell the system the relevant information about that DBMS. This information is solicited through a dialog with the user. The TGS continually guides the user by asking a series of questions about the DBMS in question.

Detailed information concerning installation and running of the software as well as the relevant tables (files) generated by these programs can be found in the appendices.

The TGS is an interactive software package which infers templates from user input. The user describes how specific examples would be written in his/her own native database language, and the software infers the syntax of the language using techniques largely based on artificial intelligence.

The software is segmented to allow separate user sessions, each user session generating a specific type of template for the user's database language. It is only assumed that the user is knowledgeable about his/her own database language. Nothing is assumed about knowledge of **DAVID** templates, and the user is not asked to answer any questions which require knowledge about templates. For information's sake only, portions of the template being generated are displayed as the reasoning progresses.

The separate sessions are invoked by separate modules. The various sessions which can be invoked are:

1. A session to generate separate log-on and log-off templates for a session in which a user wants to define a database, using either the inner or the outer approach.

2. A session to generate a template for defining a database in the user's native database language. There are actually two modules built for this purpose:

2a. One which allows a user to describe how one defines a database in a relational database language.

2b. Another which allows a user to describe how one defines a database in a hierarchical or network database language.

3. A session to generate separate log-on and log-off templates for a session in which a user wants to access a database which already exists on the system (the Install operation).

4. A session to generate a query template for a session in which a user wants to enter a query against a database. (This session can be used only in the design of outer interfaces.)

A session of type 1 must be run before a session of type 2, because the template generated during a session of type 2 includes the log-on and log-off protocols elicited from the user in a session of type 1. Similarly a session of type 3 must be run before a session of type 4. Otherwise the sessions are completely independent.

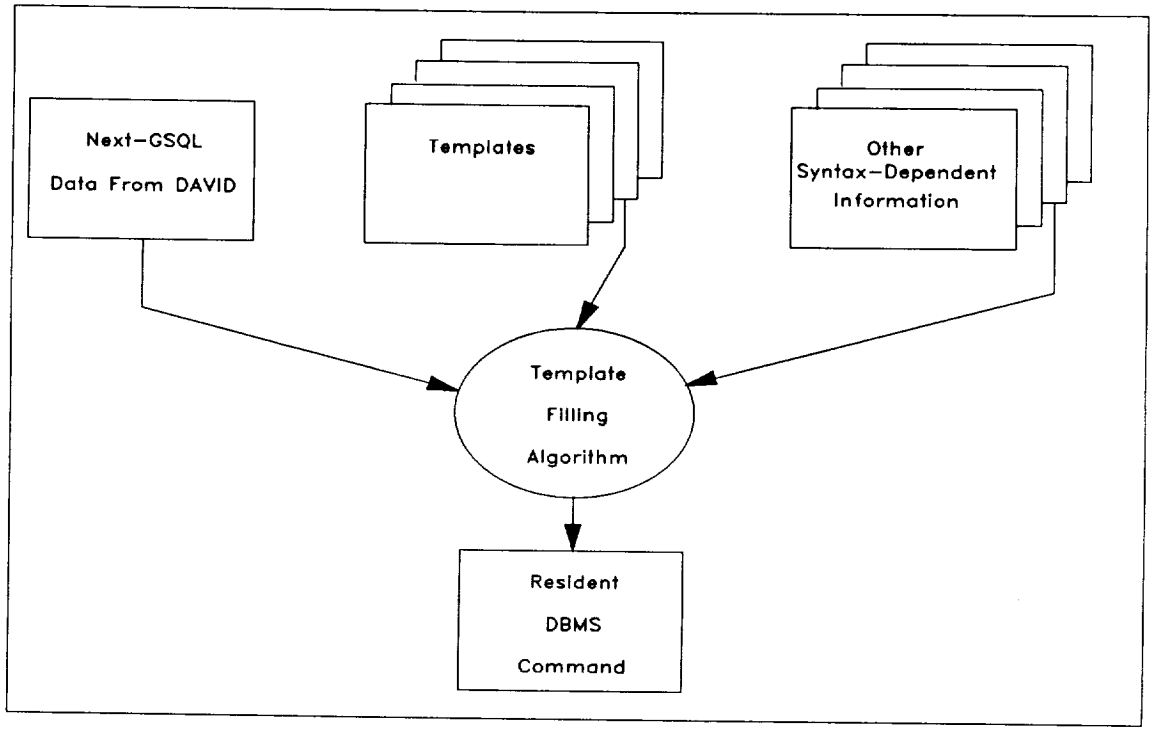
These modules generate tables along with templates when such tables are needed to supply additional information. The software runs on IBM PC compatible computers with 640K of main memory and a hard disk. At the completion of a session the templates and associated tables are stored on the hard disk so that they can be retrieved at the user's convenience.

This expert system tool can be used to design all components of outer interfaces; it can also be used to design the *Define, Delete, Install* and *Drop* components of inner interfaces.

Interface Driver program

The second software component of the DIDS system is the Interface Driver. Recall that a template is a block of text designed to capture all the syntax of a DBMS command without the actual database-dependent data. The TGS, described above, creates a template with DBMS independent items (names the items preceded with "@"). The Interface Driver is the software component that replaces all the "@" commands with DBMS specific language at run time. Thus the TGS is independent of all DBMSs, while the Interface Driver fills in the DBMS-dependent information into the output of the TGS. From the templates generated by the TGS and other syntax dependent tables generated by the TGS (see appendix 2), it creates a "filled-in" template which can be used directly by the **DAVID** system. See figure 1 for the over-all organization of this module.

Figure 1: Interface Module Architecture



The following 3 constructs are descriptions of items in Figure 1.

(1) NEXT-GSQL: This is a **DAVID** data structure that contains query-specific data to be translated into resident DBMS syntax. This structure is filled in by **DAVID** and passed to the Interface Drive.

(2) Templates: For each database operation (e.g., define, selection-projection, etc.) there is a template describing the syntax of that operation in that DBMS. The selected template will be filled by the template-filling algorithm which is the main element of the Interface Driver. The result is a module which performs the given operation for the given DBMS. These modules are executed by the **DAVID** system in a timely fashion.

(3) Other Syntax-Dependent Files: For each DBMS, there may be some DBMS dependent information required to complete the template generation. For example, in translating a DEFINE statement, on needs to know the corresponding type name of say INT, in the resident DBMS. The type name may be INTEGER or I or "numeric" etc. This information is found in the

type conversion table. This table and others is stored in files generated by the TGS. Contents and format of these files can be found in Appendix 4.

Template Design

A template is a block of text that captures all the syntax of a DBMS command without the actual data. Because a template does not contain any actual data, we need some special symbols to define the syntax of the template. In this section, we discuss the syntax of the templates.

General Syntax

[1] @-Sign: Any token that begins with an @-sign is a special command, to be processed by the software. Either it is replaced by a piece of data, or by a loop construct in the completed code. Symbols which are not prefix by @-signs are considered to be constants.

[2] Whitespace: Any @-command should be terminated by a whitespace (blank, newline, tab). This whitespace terminator is consumed during the scanning process and does not appear in the filled template. All other blanks will be echoed as regular characters.

[3] Back Slash: The back slash character (\) is a metacharacter that has special meanings. To have it in the (output) string, one must use \\ to override it. Thus "\\" is equivalent to "\". In general, "\c" is equivalent to "c" for any character c NOT equal to "n". See [4] for the meaning of "\n". This provides a way to generate the actual character "@" in the filled template ("\@").

[4] End-of-Line: Any "physical" end-of-line shall be ignored by the template-filling program. Instead, "\n" is used to break up the filled template into lines. In other words, the physical end-of-line in the template are there to make reading easier. It has nothing to do with how the filled template should look. (The filled template gets its line spacing from the back slash n character).

Substituting Commands

The syntax for the substituting commands is given below:

@<name>[-<seq.no.>]
where <name> is a string of UPPERCASE letters of length one or more, and the <seq.no.> is a positive integer. The sequence number is optional with a default value of one. A list of substituting commands and their definitions can be found in Appendix 7. The command names can be changed. To do so, the corresponding names in the template and the Build_hash_table function have to be changed to exactly the same name.

Repeating Commands

The syntax for the repeating commands is given below:

@BEGIN<name><delimiters>

....

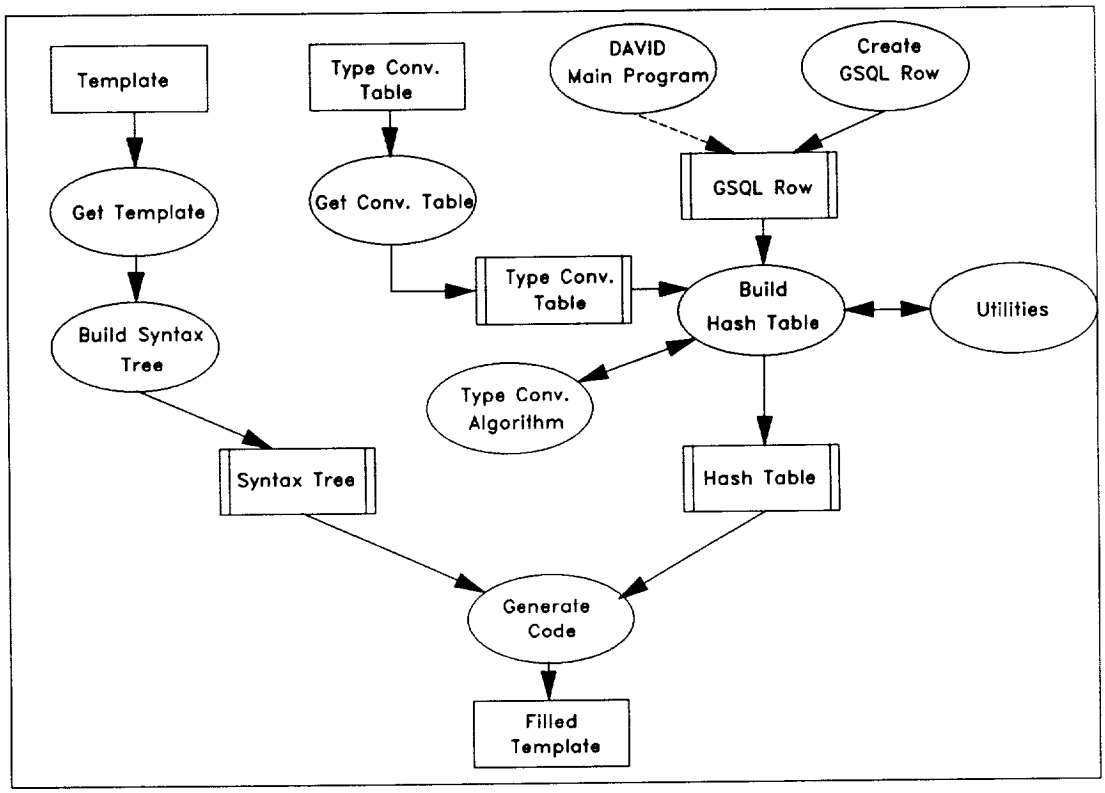
@END<name>

where the <delimiters> is defined as
["<" ,separator> [">" <L-delimiter> "><" <R-delimiter>] ">"]. A list of all repeating commands and their definitions can be found in Appendix 7. The command names can be changed. To do so, the corresponding names in the template and the Build_hash_table function have to be changed to exactly the same name.

Program Structure

A brief description of the structure of the program is given in this section. Figure 2 below shows a general organization of the interface module. Oval blocks indicate program modules. Rectangular blocks are data files/structures. Those rectangular blocks with double edges are internal structures. A description of each file included in this module is given below.

Figure 2: Interface Module Program Organization



david.c

This is the main driver of the module. In includes two header files gsqlrow.h and template.h. The main algorithm is stated below:

-Select and Open the Template

-Build a syntax tree using the Template

-Create a GSQL-ROW structure

-Select and open the Type-Conversion table

-Traverse the GSQL-ROW structure

-Select the information needed to fill in the template.

For type information call the type conversion function.

Build the hash table to store all selected information. All future references to the data is via the hash table.

-Open an output file

-Generate filled template

gettemp.c

This file contains a function to open a template file.

tree.c

The file contains two major functions dealing with syntax trees. The first, BUILDTREE, builds the syntax tree using the template. The second, (GENERATECODE) fills the template using the syntax tree and data from the hash table.

gettable.c

This file contains a function that reads the type conversion file and stores the information in an internal type conversion table for later use.

buildht.c

This file contains routines that build the hash table. The main body of the file walks through the GSQL-ROW to select the information neeeded to fill the template. For type information (name, length, precision, etc.), a call to the type conversion function is needed. Build the hash table to store all selected information. All future reference to the data is via the hash table.

typeconv.c

This file contains two functions involved in dealing with type conversion.

utility.c

This file contains several utility routines used throughout the module.

Guidelines for Interfacing Resident DBMSs to the DAVID System

Introduction

These guidelines are intended to serve as an aid to the interface developer. Using these guidelines, an interface developer should be able to design, build and test a complete resident database management system interface. The aid of the host DBMS database administrator is required for installation of the software generated with the help of these guidelines.

These guidelines can be used to design interfaces to any kind of DMBS. For designing outer interfaces (see explanation below of outer interface) the developer need not be a programmer; familiarity with the resident DBMS to be interfaced with the DAVID system is sufficient. To develop inner interfaces (see explanation below), programming ability is required.

The DAVID System

Here, we describe the **DAVID** database management system, and outline the process of interfacing your database management system (DBMS) to **DAVID.**

What is DAVID?

The *Distributed Access View Integrated Database (***DAVID***)* system is a heterogeneous distributed database management system currently under development at NASA's Goddard Space Flight Center. It is heterogeneous, meaning that its database structure, called a *cluster,* supports databases with relational, hierarchical or network structures. It is a distributed system, and can run simultaneously on a variety of computers communicating via a local area network and/or a variety of wide area networks. It has its own query language, GSQL, and also provides data access through a number of high-level programming languages such as C, PASCAL and FORTRAN. In addition to access via query language and high-level programming language, **DAVID** provides on-line interactive software for browsing through databases and performing transactions.

What is an Interface to the DAVID System and what is it used for?

An interface to the **DAVID** system consists of a set of software routines connected to the **DAVID** software, which allow the data from databases in your DBMS to be processed via **DAVID**. Once a DBMS has a set of interface routines linked into **DAVID**, any database in that DBMS can be accessed via **DAVID** (provided, of course, that its owner grants permission). Data from your DBMS can be stored into **DAVID**, and **DAVID** data can be stored in a database belonging to your DBMS.

Why would you want your DBMS to be interfaced with DAVID?

By building an interface between the **DAVID** system and your DBMS, we can provide your DBMS with the capabilities of the **DAVID** system: Queries can be issued on your data using the **DAVID** query language, **DAVID** access routines can access your data, and data from **DAVID** can be stored into a database on your DBMS and vice veers. This gives access capabilities to your database that you might not otherwise have. Furthermore, you are now "interfaced" to any other DBMS that is in turn interfaced to the **DAVID** system. So, for example, you can issue queries on an ORACLE database whose results can then be stored in a DBMS of your choice -- all through the **DAVID** system. (This assumes, of course, that interfaces exist between **DAVID** and your DBMS, and **DAVID** and ORACLE.)

What are the Components of an Interface?

Here, we discuss the component parts that make up a complete resident database interface. Depending upon the type of DBMS you wish to interface, different of these components will be used. A discussion of the types of DBMSs follows this section.

Components to Define, Install, Delete and Drop a Resident Interface.

These routines create new resident databases (DEFINE), connect existing ones to the DAVID system (INSTALL), disconnect resident databases from the host DBMS without

actually touching the data in any way (DROP), and delete resident databases from their own DBMS as well as the DAVID system (DELETE).

DEFINE allows the user to create a new database under his/her resident DBMS entirely through the DAVID system. The user creates a database definition, or schema, for a new database using DAVID syntax on the DAVID system. When the request is executed, it is translated into the syntax of the resident DBMS and executed; it then creates the new database. In addition, it puts an entry into the DAVID directory about the newly created database, and enough of its schema information for DAVID to be able to interpret the data from this database as if it were a DAVID database.

INSTALL performs the second part of the DEFINE operation; that is, it enters information about an existing resident database into the DAVID system so that it can be accessed via DAVID. It does not touch the database itself, which already exists.

The DEFINE operation, then, is used to create new resident databases and tell the DAVID system about them; the INSTALL operation is used only to tell DAVID about existing databases.

The DROP operation drops directory information about the resident database from the appropriate DAVID directory; the database is still intact in the resident DBMS after execution of a drop command, but DAVID can no longer access it.

The DELETE operation does all that the DROP operation, described above, does; in addition, it erases the actual database and all of its associated data.

All of the four operations described above exist on all resident databases, regardless of type. The components to be described next differ greatly according to the type of DBMS being discussed.

Components which depend on the type of DBMS to be interfaced

First, the necessary information about the resident must be installed in the directory of the host DBMS. Once this has been done, queries and transactions meant for the resident can be submitted through the host. These operations are outlined in the next group of routines.

The interaction between the resident and the host can take place on 3 possible levels: the query language level, the table level, and the path level.

The Query Language Level

Some resident DBMSs support their own query languages. For those DBMSs, a query or transaction involving a resident submitted through the host DBMS can be translated by the resident interface into a query or transaction in the language of the resident, and then executed by the resident in its own environment. The following primitive queries have been isolated as the components of any complex query between a resident database and a host database:

Generalized Selection-Projection. A selection-projection or selection-multiprojection is performed on a resident database. The results of the query are stored in a new database on the host.

Semijoin. A join is performed between a resident database and a host database. The result is a new database on the host, and a table of pointers to rows of data items in both the source host database and the new result database.

Store-to-Database. A selection-projection is performed on a database on the host DBMS; the result is stored in a new database in the resident DBMS.

Insert, Delete, Update. Transactions submitted through the host are performed on a resident database. Insert adds row(s) of data items, delete removes a row or more, update modifies a row or rows.

The Path Level

If the resident DBMS has a high-level language interface (such as C, PASCAL or FORTRAN), and supports a command to retrieve information from several tables of the database as a single access (as is often the case in a hierarchical database), then requests for information and transactions submitted via the host can be handled by the host at the Path Level. The host DBMS determines the proper access path through the resident database; then calls on the path access routines of the resident DBMS to navigate through the resident database. These routines are:

Path First Row. The first path "row" of the resident database (i.e., the first row of every table that makes up the specified path through the database) is read by the resident DBMS, and the data inserted into the host DBMS data buffers. There is one host DBMS data buffer for each corresponding table row of the resident. Any boolean evaluation is performed by the host DBMS on the data in the buffers.

Path Next Row, Path Previous Row, Path Last Row. The required path "row" of the database is read into the corresponding host DBMS buffers, where any boolean evaluation is done.

Path Insert, Update, Delete. A path "row" of the database, including all of the tables specified in the path, is inserted, updated or deleted.

Path Assign. This routine performs "housekeeping" functions necessary to allow the resident DBMS to communicate with the host at the path level: it determines which tables of the database must be used to make up the path, and allocates necessary data buffers required by the resident to contain path information.

Path Deassign. This routine performs any "housekeeping" functions necessary to terminate a resident interface path access by disassociating the resident with that path if necessary, and deallocating any special data areas set aside by Path Assign.

The Table-Row Level

In some cases, it is necessary to access the resident DBMS at the lowest level, i.e., the Table-Row Level. This table-at-a-time access capability is used for those resident DBMSs which support a high-level language interface, such as FORTRAN or C, but provide neither a query language nor access at the path level. This is common in network DBMSs. In these cases, the only way for the host DBMS to process the resident data is at the single table level. When resident DBMSs support user access at the table-row level, requests for information and transactions submitted via the host are processed by the host until the request is reduced to a call for a single row of a single table. At that level, the request can be handled by the resident DBMS. If the table-row request

includes a boolean, the boolean evaluation is done by the host on the data after it has reached the host buffer. The table-row routines are:

Table-Row First, Table-Row Last. The first (last) row of a table of the resident database is read into a buffer of the host DBMS. Any boolean evaluation is done in the host buffer.

Table-Row Next, Table-Row Previous. The next (previous) row of a table of the resident database is read into a buffer of the host. Any boolean evaluation is done in the host buffer.

Table-Row Insert, Table-Row Delete, Table-Row Update. A row of data of a table in the resident database is inserted, deleted or updated.

Table-Row Connect. A row of data which has just been inserted into a table in the resident database, is connected to the proper parent. This routine only has applicability in a database of network structure.

Table-Row Disconnect. A row of data is disconnected from the designated parent, and if that is the last parent, the row is deleted from the database. This routine only has applicability in a database of network structure.

Table-Row Parent. Given a row of a table, and the name of a parent table of that table, this routine returns a row of data from the parent table. This routine only has applicability in a database of network structure.

Assigning and Deassigning Resident Databases. Before either table-row routines or path routines can access a resident database, certain initialization functions must be performed. Similarly, after path or table-row access routines to a resident have been performed, some termination operations functions must take place before the resident DBMS is exited. These routines are as follows:

Assign Database. This routine performs "housekeeping" functions necessary to allow the resident to communicate with the host: it establishes the necessary data buffers and variables needed by the resident, logs on to the resident, provides the necessary security, performs any necessary language translation or execution of query language statements.

Deassign Database. This routine performs "housekeeping" functions necessary to terminate host interaction with a resident: closes the resident database, logs off of the resident DBMS, and deallocates any special data areas set aside by Assign Database.

Interface templates

The next section discusses the set of templates that makes up each of our interfaces.

What are they?

The templates are the "building blocks" of each resident interface package. There is one template for each interface component (described above) in every resident DBMS interface. The construction of these templates is the primary task of you, the interface builder. Once the templates have been constructed and installed, software in the **DAVID** system works with them to form the complete interface.

The templates can contain

- high-level programming language statements
- calls to **DAVID** data management routines
- calls to your own DBMS data management routines

Some templates contain a mixture of the three types of statements; some contain only one or two of the above, depending on your DBMSs requirements, and on the purpose of the routine. A sample template, that for a selection-projection (retrieval) routine for the ORACLE (a relational) DBMS, is shown below, just to give you a sense of what templates look like. No need to study it; we'll examine more templates in detail later in these guidelines.

How are the templates constructed?

You (the interface builder) construct a set of templates for any DBMS you want to use. This set of templates is used by all databases on that DBMS and for all queries and transactions issued under that DBMS. So if all of your databases run under the same DBMS, no matter how many databases you want to hook into the **DAVID** system, you need only build one set of interface templates. If you use several different DBMSs for your databases, say, three different DBMSs, then you will

need three different sets of templates for your three interfaces. Detailed instructions as to how to construct the templates (and what to do with them when you're finished) appear below.

Where do they fit into the DAVID system, and how do they interact with DAVID?

For this illustration, let's assume that you want to issue a selection-projection query on your employee database, such as

```
SELECT name, ss# FROM employee WHERE salary > 50000
```

--that is, Give me all names and ss#s of people in the database who have a salary greater than 50000. Let's also assume that your database runs on the ORACLE DBMS. Briefly, here's what happens when that query is entered into the **DAVID** system: As soon as **DAVID** determines that the query is intended for an ORACLE database, the ORACLE interface is "activated", that is, the template corresponding to the particular ORACLE transaction (in this case, selection-projection) is retrieved from the template "cluster". The associated software fills in the template with database-specific and query-specific information and, in most cases, writes it to a file. The result is a program containing a mixture of statements, commands to ORACLE and commands to **DAVID**. When this program is executed, your query is performed.

Some interface commands behave differently; for example, for "micro" DBMSs such as DBASE there is no programming language support, so the file built by the interface software simply captures your query results onto a file, which can then be read by **DAVID**. Some DBMSs don't have query language support. In these cases, retrieval routines must be built into **DAVID** in a different way. That's why, before building an interface, it's important to determine what type of DBMS you are using. The type classifications are explained below.

How to Build an Interface

In this section, we present detailed instructions for creating interface templates. If you read and follow the instructions carefully, hopefully you will be able to build an interface to **DAVID** for your DBMS.

Determining the interface type for your DBMS

The first thing you must do is determine into which type classification your DBMS falls, for **DAVID**'s purposes. Depending on the DBMS type, some of the interface templates differ. Here is a description of the types:

Type 1 DBMSs have the following characteristics:

-- they support databases which are composed of tables of data with no physical connections, i.e., no parents, children, or sibling pointers. These are commonly called relational databases.

-- they support a query language

-- they support an interface to a high-level programming language, such as C or FORTRAN.

Type 2 DBMSs have the following characteristics:

-- Their databases consist of tables (or records) connected, via pointers or physical proximity, to children, and to a single parent. There may also be sibling connections. A table can have multiple children, but only a single parent. These are commonly called hierarchical databases.

-- Their databases can be accessed via path routines, i.e., "macro" routines for the DBMS which, when given a path through the database consisting of parents, children, the children of those children, etc., will access the tables which make up that path.

-- They support an interface to a high-level programming language, such as C or COBOL.

-- Some support a query language, some do not.

Type 3 DBMSs have the following characteristics:

-- their databases consist of tables connected, via pointers, to children and parents. There may also be sibling pointers. A table can have multiple children, and multiple parents. These are commonly called network databases.

-- access of Type 3 database is via table-row routines, i.e., "macro" routines for the DBMS which, when given the name of a table and of the proper parent for that table, read/write/update/delete the table (or the parent) in that table-parent set

-- they support an interface to a high-level programming language, such as C or COBOL

-- most of them do not support a query language

Type 4 DBMSs have the following characteristics:

-- they support databases structurally composed of tables of data with no physical connections, i.e., no parents, children, or sibling pointers. These are commonly called relational databases.

-- they do not support an interface to a commonly-used high-level programming language, such as C or FORTRAN. Some support a limited language, with a very restricted set of operations, but they are not stand-alone languages, i.e., they are a part of their particular DBMS environment, and run only under that DBMS.

-- they have some sort of query capability; usually it is a primitive version of a query language.

These DBMSs commonly run on microcomputers.

Type 5 DBMSs have the following characteristics:

Type 5 DBMSs are not, strictly speaking, database management systems. The "databases" which will require a TYPE 5 interface consist of data stored on tape or other media as ASCII or EBCDIC files. Some of these files are part of a "system", i.e., they are described by headers of a standard format which contain parameters to aid in the interpretation of the data. Others have no header information, and it is up to the user to supply all routines to manipulate the data. In our research, we have treated all such files as another type of DBMS, with the missing database processing capabilities provided by the host. We will refer to them as "generic files".

An interface to generic files has proven to be an extremely useful tool. A great deal of data at NASA is on such files, and conversations with scientists at NASA has shown that the ability to operate on files with the power of a heterogeneous DBMS would be most welcome.

Residents classified as Type 5 exhibit the following properties:

-- their structure consists of a single file containing identically formatted records

-- they can be processed by any high-level programming language that has I/O consonant with their file organization

-- they do not support a query language

-- software used to process their data is user-supplied; i.e., they have no associated data-manipulation routines as do DBMSs

Using the information supplied above, you should be able to determine what type of DBMS your is. Here's a review of the points above:

-- if it has a query language, it's Type 1
-- if it has no query language, but a host language interface, and it's hierarchically structured, it's Type 2
-- if it's the same as Type 2 but has all the network features, it's type 3.
-- if it's a micro DBMS, with some limited query facility and no commercial language interface, it's Type 4
-- if it's not a DBMS at all, but just a collection of files, it's type 5.

The Generic Templates: Filling in the @@ signs

We have designed generic templates for you to work with in building your specific interface templates. For each DBMS type (types 1 through 5 are described above), there's a set of generic templates, one for each module of your interface. As soon as you have determined which type of DBMS you are working with (see the type descrip-

tions above), your job will be to fill in the generic symbols -- they are "@@" (double "at" signs) -- with information specific to your DBMS.

Building the Definition Generator Templates

There is a single generic definition generator template for all database types described. See figure 8 for the complete template. Some of the types use only a subset of the keywords in the generalized template; others use a different subset. Keywords which are not used are simply deleted, as we'll see in our example below.

In this section, we'll work through an example, building a template for definition generation. Then we'll outline a general strategy for filling in a Definition Generator template. Next, we'll discuss other considerations introduced by the different DBMS types.

An Example

The object of our efforts in this example is to produce a Definition Generator template for ORACLE from a real sample ORACLE database definition, using as a model the generic Definition Generator template.

Figure 8 is the generic template for the Definition Generator for a type 1 DBMS. We'll fill it in step by step for the ORACLE DBMS. You may wish to have figure 8 in front of you, as well as figure 9, a sample ORACLE database definition.

The ORACLE database definition in figure 9 is, along with every other database definition, made up of two types of objects: ORACLE components (keywords and symbols, such as the words "create table" and "char" and "INT", and the symbols "(" and ")" in our example); and non-ORACLE components, such as the variables "student" and "name" and the length "8". In our generalized template, the ORACLE components correspond to the "@@" items; the non-ORACLE items, to the "@" items in the template. To build an ORACLE template from a generalized template, we replace the symbols beginning with "@@" by the ORACLE keywords. Thus, the completed ORACLE template will contain only two kinds of items: ORACLE components, and template symbols beginning with "@".

Note that we do NOT replace the symbols beginning with "@" by non-ORACLE items; if we did that, the result would be, not a template, but a specific ORACLE database definition. By leaving the "@" symbols, we create a template into which a variety of variables can be filled in during definition processing.

We begin by scanning the template, line by line, comparing it with our ORACLE example. For every line, if there is an ORACLE keyword corresponding to the template "@@" symbol, we replace the template "@@" symbol with that ORACLE keyword. If there is no corresponding ORACLE symbol, we eliminate the template "@@" symbol. If there is an ORACLE constant (variable name, etc.) corresponding to the "@" (constant) symbol, we leave the template symbol alone; if there is no corresponding ORACLE constant, we eliminate the template "@" symbol.

The first line we encounter is one for schema name information. ORACLE has no schema name information, so we eliminate the entire line from the template. Similarly, we eliminate the line for file name, area name, and dbname (ORACLE has no database name; only the tables of ORACLE have names).

On the first line of our ORACLE example, we see the words "create table student". We know that "student" is the table name; in our template, it appears as the symbol @tablename. The words "create table", then, are our @@tablename keyword. So in the generalized template, we replace "@@tablename keyword with "create table". Since this is the only attribute describing a table, "@@table attribute separator" is eliminated. Similarly, the next four lines: parent information (ORACLE has no parents), length keyword, start position keyword, and comment keyword, can all be eliminated.

So the first symbol in our ORACLE template is the template keyword @beginrtable. Comparing the ORACLE template to the generalized template, we see that in the case of ORACLE, the information contained in the first six lines of the generalized template are not needed by ORACLE, and so can be eliminated.

Let's look at what we've done so far. We've scanned the generalized Definition Generator template, the first 12 lines. This much of the generalized template is represented in the corresponding ORACLE template as

CREATE TABLE @tablename

Before we begin our first ORACLE field (See figure 3.2) there is a "("; since it appears only before the first field, and the corresponding ")" after the last one, we conclude that these are table delimiters, not field delimiters. So the next line in the generalized template, "@@table left delim" is replaced by "(" in the ORACLE version. Later, we will see that "@@table right delim", towards the end of the template, will be replaced by ");" which are the symbols that appear at the end of each ORACLE table.

Our first ORACLE field is preceded by nothing after the "(", which we have already tagged as a table left delim. So we can eliminate the generalized template symbols "@@field header" and "@@field left delim" for our ORACLE template. Similarly, there is no "@@field name keyword". The symbol "@fieldname" represents the symbols "name" "id" and "grade" in our sample ORACLE database definition. The symbol "@datatype" represents the symbols "char" and "INT" in our sample ORACLE definition; there are no "@@datatype keyword" parameters or "@@field attributed separators". The "@@field length left delimiter" becomes "("; the "@@field length right delimiter" becomes ")".

There is no "@@fldstart" information; hence that line of the generalized template is not used. Neither is there a "@@field comment" section; hence, that line can be eliminated.

The "@@table right delimiter" symbol was discussed above; as we see from our sample ORACLE database definition, it turns out to be ");".

The rest of the generalized template symbols fall under the category of "network header" information; they are used only for Type 3 DBMSs, and so will be discussed in that context. We eliminate them for ORACLE

There is no "@@db right delim" symbol or "@db terminator" symbol; these items can be eliminated.

Figure 10 shows the completed ORACLE template.

Now we must fill in the data type table information (see Appendix for data conversion table); then our template package is finished. To do that, we consult an ORACLE user's manual and a **DAVID** user's manual, to match up the data types. We come up with the following:

ORACLE DATA TYPE	TEMPLATE TYPE #	**DAVID** DATA TYPE
char	2	char
INT	2	num
INT	4	float

A final table to be completed is called the Length Parameter table. For all datatypes in your DBMS, the following questions are answered:

-- is there a length field?
-- if so, how many length parameters are there?
-- if there are two parameters, what do they mean?

The complete table is in Appendix 4.

With the completed template, and the tables above filled in, the interface for your DBMS is now complete. Section 3 explains what to do next.

General Strategy

In the previous section, we filled in the generalized Definition Generator template using our ORACLE example as a model, creating an ORACLE Definition Generator template. Here, we outline a general strategy for filling in a Definition Generator template.

The first thing you must do is determine which type of DBMS (types 1 through 5 are described above). Then skip to the section in this document which describes the Definition Generator Templates for your type of DBMS.

Type 1 Definition Generator Templates

In the example of ORACLE we filled in a type 1 Definition Generator template. Our type 1 sample DBMS, ORACLE, did not contain such fields as database name. Your type 1 DBMS may require more of these items.

To fill in the template, employ the same method as we did in our example above: use the generalized template as a guideline, and for each symbol prefixed by "@@"

-- if that construct exists in your DBMS, replace the @@ symbol by the one in your DBMS
-- if that construct doesn't exist in your DBMS, remove the @@ symbol.

Type 2 Definition Generator Templates

Type 2 database management systems support tree-structured databases, where the tables can have parents and children. So in addition to the information about individual fields such as that supplied in the Type 1 templates, parent and child information must be supplied in the definition of a table.

Figure 3.3 contains the general template for a type 2 DBMS; figure 3.4 contains a typical type 2 schema: one for the IMS DBMS. The following additional keywords appear:

```
   @@parent_keywd -- to indicate that this is a parent
table
   @@table_length_keywd  -- to indicate the length of
the table
   @@table_start_pos  -- to indicate the starting
                         position of the table
   @@table_comment_keywd -- to indicate a comment
```

Associated with the above keywords, there are left and right delimiters and separators that must be supplied in the proper places. See the example above, in which these delimiters are filled in, to see how the process is done.

As an example, the generalized template portion which pertains to tables and their hierarchical relationships appears as follows (this is a portion of the generalized template):

```
@@table_name_keywd @@tbname_prefix @TABLE_NAME @@tbna-
me_suffix &
@BEGINPAR @@table_attr_Sep @@parent_keywd @PARENT_NAME
@ENDPAR
@BEGINTLEN @@table_attr_sep   &
@@table_length_keywd @TABLE_LENGTH @ENDTLEN
```

```
@BEGINTPOS @@table_attribute_sep  &
@@table_start_pos_keywd @TABLE_START_POS @ENDTPOS
@BEGINTCOM  @@table_attr_sep &
@@table_comment_keywd @TABLE_COMMENT @ENDTCOM &
```

When the @@ information on the template is filled in by the interface builder with information germane to a type 2 database, the resulting text is a piece of the template for a database definition of that type 2 database. Here is an example (this example is a part of the database definition for an IMS schema):

```
@BEGINTABLE<\n>
SEGM NAME=@TABLENAME,PARENT=@PARENT_NAME,
     BYTES=@TABLE_LENGTH,START=@TABLE_START_POS
```

so, we can see, for this example,

@@table_name_keywd is "SEGM NAME="

@@table_attr_sep (separators between table attributes) is ",",

there are no comments, so @@table_comment_keywd is eliminated, along with @BEGINTCOM, @TABLE_COMMENT and @ENDTCOM

Proceeding along these lines, and using the techniques illustrated in the ORACLE example, above, we either fill in the rest of the @@ symbols with their DBMS-dependent counterparts, or delete them.

Below is an example of a piece of a type 2 database definition, the result of the interface driver's operation on the template subset shown above:

```
SEGM NAME=employee, PARENT=deptment, BYTES=24
```

This is an actual portion of the database-specific command that would get executed by the resident database system to create a new database. The rest of the template, and hence the code generated to define the new database, is the same as for Type 1 databases, and so will not be discussed here.

Type 3 Definition Generator Templates

Type 3 definition generator templates differ from those of Type 2 only in that the data organization of the

databases is different. Since Type 3 databases are network in structure, one child can have several parents, and so the structure described above will not suffice. Instead, the generalized template contains information about SETS, which is what child-parent tuples are usually called. For each set, the requisite information to be supplied is the name of the parent, or OWNER, as it's often called, and the name of the child, or MEMBER, as it is sometimes known, plus appropriate delimiters.

Here is an example of a subset of the generalized template used to connote these relationships. The rest of the template is the same as for Type 1 DBMSs.

```
@@network_header
@BEGINSET<@@set_sep><@@set_left_delim>&
   <@@set_right_delim>
@@set_name_keyword @SET_OWNER_NAME @@set_name_sep
@SET_MEMBER_NAME &
@@set_attribute_left_delim
@@set_owner_keyword @SET_OWNER_NAME @@set_attribute_se-
parator   &
@@set_member_keyword @SET_MEMBER_NAME  &
@@set_attribute_right_delim   &
@ENDSET
```

We now fill the above template subset in for a sample Type 3 DBMS; namely, IDMS. We have no @@network_header, so we eliminate the line. Our set_name_keyword is SET NAME IS; our set_owner_keyword is OWNER IS; our set_member_keyword is MEMBER IS. The completed template subset for this DBMS, then, is:

```
@BEGINSET
SET NAME IS @SETMEMBERNAME-@SETOWNERNAME \n
OWNER IS @SETOWNERNAME    \n
MEMBER IS @SETMEMBERNAME  \n
@ENDSET  \n
```

to be repeated for as many sets as there are in the specific database. A completed example of this subset of a database definition, completed for a department-employee database, is as follows:

```
SET NAME IS deptment-employee
OWNER IS deptment
MEMBER IS employee
```

The rest of the database description is generated as it is in the other database types described above.

Type 4 Definition Generator Templates

The database definitions for Type 4 databases do not differ in structure of those of type 1. Since these databases are tabular in structure (in fact, they are usually limited to a single table), and since the tables are never connected, they have no parent or set pointers, and so the subset of the generalized template used in type 1 databases also applies to them.

For a complete explanation of how to fill in a type 4 definition generator template, see the ORACLE example above, and the explanation of type 1 definition generator templates.

Type 5 Definition Generator Templates

There are no Definition Generator Templates for objects of Type 5. Since these are not databases, but rather general files, there is no "schema" or database definition to be generated here. Type 5 databases rely only on their counterpart, the **DAVID** database definition, to describe them. To define a Type 5 database to the **DAVID** system, you simply use the **DAVID** DEFINE command as if it were a **DAVID** database (see the **DAVID** user's manual for the syntax of the DEFINE command); in the STORE AS clause at the end of the DEFINE, you fill in the DBMS type as FILE and the name as the name of your file. For example, to create a definition for a file named FILE1, your **DAVID** DEFINE command ends with the clause

```
STORE AS FILE(FILE1)
```

See the section *Testing Your Interface Components*, for more information on how to generate a database definition.

Building the Define Templates

Your Define Template emerges from the Definition Generator Template (see section 2.3, above) you constructed. The Define Template surrounds the Definition Generator with the commands needed to bring up, or activate, your DBMS and provide it with the necessary security, and the commands needed to exit from your DBMS. Usually, this is only a command or two.

The commands for the DEFINE appear in the Definition Generator template, at the top and the bottom. When you have completed the process of filling in (or deleting) the "@@"-prefixed symbols in the generalized template, these commands will be filled in also; there is nothing additional for you to do.

Building the Install Templates

The Install templates are constructed in the exact same manner as the Definition Generator templates; remember, the only difference between the Define operations and the Install operations is that the Define creates an entirely new database from scratch, while the Install connects an existing database to the **DAVID** system.

Building the Drop and Delete Templates

The Drop operation is entirely a **DAVID** matter, since the Drop operation simply disconnects your database from the **DAVID** system. No resident database commands are needed for a Drop, and so there is no template to construct.

The Delete operation, on the other hand, both drops the database definition from the **DAVID** system *and* deletes the database itself. So the template for this operation contains the command to actually delete, or erase, the database definition and all its associated data from your DBMS. The template is a simple one; just replace the @@delete_database symbol with the language your DBMS uses to delete a database.

Building the database access templates

There are several different methods for building the database access templates, depending on the database Type of your DBMS. For those DBMSs which support a query language, (i.e., Type 1 and type 4 DBMSs), the purpose of the template is to tell the **DAVID** system how to interpret the query. For those DBMSs which have no query languages and which have interfaces at the table-row level (i.e., Types 2, 3 and 5), the template helps to construct routines to walk through your DBMS and feed information to **DAVID**, or insert information from **DAVID**.

The access templates for Types 1 and 4 DBMSs, i.e., those which serve as aids for query translation, reside as data in the host DBMS, and are activated as an aid to building specific queries at execution time. The access templates for the other DBMS types serve a completely different function: they are aids to interface builders for writing procedures that will then be linked into the host system.

So filling in the generalized access templates for Type 1 and Type 4 DBMSs yields another set of templates, containing symbols prefixed by "@", to be filled in at execution time with query-dependent information; while the access templates for the other types of DBMS are filled in with actual code that gets compiled and linked into the host system. There are no symbols in these completed templates; there is only executable code.

NOTE: Writing the code for Type 2, 3 and 5 interfaces sometimes presents a "language" problem. For example, we wrote an interfaces to IDMS in C, since the code was embedded inside a large system which was all written in C; IDMS has a protocol with COBOL, PL/I and IBM Assembler -- but not C. So the register conventions are all different, parameters are passed differently, and many potential problems can result. Let the coder beware!

In either case, the filled-in database access templates form the heart of the interface; they provide the mechanism for accessing the data.

Type 1 Database Access Templates

Below is the generalized template for database access for Type 1 DBMSs. It will also be used for Type 4 DBMSs, as we will see below. As we did with the Definition Generator templates, filling in the symbols beginning with "@@" for our DBMS will yield a DBMS-specific template to be used for queries. We will proceed below, with an example.

There are also two tables which accompany these templates: the Linguistic Convention Table (LCT) and the Boolean operator Table (BOT). These must be filled in with information which tells the host system how to interpret your query. after filling in the templates, completing the LCT and the BOT will provide a complete interface.

```
   @@invoke_DBMS
   @@give_password  @RUID @@pwd_sep @@give_userid @RPWD
   @@open_db @dbname
   @@establish_file_device @FILENAME

@@select_command
@BEGINSELECT
@result_field_sep><@@result_field_lft_delim>&
   <@@result_field_rt_delim>
   @RESULTFLD
              /* this may also contain table prefix.
                 see table 1 for list of options */
@ENDSELECT
   @@from_command
@BEGINSOURCE
  /*note: doing this as a loop allows for JOINS */
<@@source_db_sep><@@source_db_lft_delim>&
       <@@source_db_rt_delim>
@SOURCEDB
     /* this may contain db name and/or table name */
    /* and will appear as @@SOURCEDB or @@SOURCETABLE
*/
 @ENDSOURCE
   @@bool_command
 @@bool_lft_delim @bool_string @@bool_rt_delim

              /* @@bool_string may also contain table
prefix.
                 see BOT for list of options */

   @@close_db
   @@exit
   @@terminator
```

When the @@ symbols have been filled in with keywords from our DBMS language (or NULLed out, if that language construct doesn't exist in our DBMS), the result will be a query-dependent template to be filled in at run time.

Let us fill in the template for the ORACLE DBMS, following through from the database definition example. ORACLE has an on-line query language, OLQ, through which our query will be generated. Below is an example of an ORACLE query, to retrieve information from a student record file.

```
OLQ
USER = marsha, PWD = hi
OPEN students
SELECT name, id, grade FROM students
    WHERE semester = 'fall' AND year = 1989 AND course
= 'csc126'
                OUTPUT TO file1
EXIT
```

In the above template,

```
   @@open_DBMS is replaced by OLQ
   @@user_id is USER =
   @@pwd_sep is ,
   @@pwd is PWD =
   @@open_db  becomes OPEN
   @@establish_file_device is eliminated
```

Continuing in this vein, using our example and the generalized template, we come up with the following template for ORACLE. Notice that it contains only items with a single "@"; those are query-dependent and get filled in at run-time. All the symbols beginning with "@@" have either been replaced by DBMS-specific constructs, or have been eliminated.

ORACLE QUERY TEMPLATE

```
OLQ
OPEN

@BEGINSELECT
       <,><><>
  @RESULTFLD
@ENDSELECT
FROM
@BEGINSOURCE  /*note: doing this as a loop allows for
JOINS */
    <,><><>
  @SOURCEDB
@ENDSOURCE
WHERE
    @bool_string   @BEGINRESULT
OUTPUT TO @RESULTFILE
CLOSE @dbname
EXIT
```

Now we need to fill in the two accompanying tables; then we are finished. First, the LCT.

The Linguistic Convention Table provides a "picture" of the query for it to be properly interpreted by the system: such items as

- whether spaces are required around operators
- whether field names have to be qualified by tables

The format of the table is just a set of answers to simple questions; the complete table appears in the appendix to this document.

The Boolean Operator table aids the system in constructing a Boolean condition for your query in your DBMS. For each Boolean operator that the host has, you are requested to represent it in the language of your DBMS. Also, the precedence rules are established. This table also appears in the appendix to this document.

These tables, along with the completed template, form a complete picture of your DBMS; the interface software of the host DBMS system can now construct queries in your query language to be submitted on your databases.

Type 2 Database Access Templates

Type 2 and Type 3 DBMSs are both connected to the **DAVID** system via inner interfaces. In that case, as you may recall, we need to provide five database operations for the interface, namely, *open, close, first, next,* and *insert.* Since database systems differ greatly in the way these operations are handled, a detailed case of both a Type 2 and a Type 3 interface will be discussed below.

If there were a database access template for type 2 and type 3 DBMSs, it would simply consist of the commands:

```
@@open_database
@@read_first_record
@@read_next_record
@@close_database
```

These commands would be used quite differently than the conventional template commands. Each would yield not a string of text, but a function or collection of functions, written in C or some other high-level language. The functions themselves would not appear sequentially, but would be imbedded in other functions, scattered throughout the DAVID DBMS code. They would be compiled and linked into DAVID, and executed via DAVID function calls. Their net effect would be to make the resident database function, not like a resident database with its own associated DBMS, but like a DAVID database. DAVID would call on the resident DBMS only to implement the calls named above.

Below we described the 4 template commands described above, plus a fifth one: *open, first, next, close,* and *insert.*

The *open* command must "bring up" the DBMS, open the specific database to be processed, and perform some "housekeeping chores". Some of these might be (the actual tasks to be done depend on the requirements of the particular DBMS to be interfaced): allocate a work area; allocate a file control block for the database and initialize it; set up an error reporting mechanism; open any files associated with the database.

The *close* command must close the database, perform any DBMS-specific functions required, such as freeing any storage areas allocated by the *open* routine, and exit from the DBMS.

Type 2 databases generally are hierarchal in nature, so that when we are asking for the first record or the next record, we must set those requests in the context of the parents of the record. Since we are supplying the parent (which the host DBMS keeps track of) for each call, these calls are in actuality for the first or next record within a particular parent.

Since *first* and *next* routines are usually implemented via subroutine calls in Type 2 databases, the two operations differ from each other only in that they are different parameters to the same subroutine call, or different subroutines. For example, in the FOCUS DBMS, the call to retrieve the next record within the same parent is a call to routine **focnxp**; the call to retrieve the first record is a call to routine **focfst**. All parameters are the same.

Similarly, *insert* is implemented by a similar subroutine call, requesting that the record be inserted into the proper place in the database.

Type 3 Database Access Templates

As in Type 2 interfaces, these DBMSs are connected to the host via inner interfaces. An explanation of the *open, close, first, next,* and *insert* follows below.

In the *open* command, the DBMS is invoked, usually via system commands, and the database is opened. Also, some of the following "housekeeping" functions are performed: storage is allocated for schemas and subschemas ("bindings" are created), data control areas are set up, an error-checking mechanism is established. Note: in type 3 DBMSs, often a new declaration for the subschema control area must be set up for every different kind of database operation to be performed on that schema; in particular, get (first or next) and insert must have different subschema control areas. The easiest way to handle this is to automatically set up two -- one for reading, one for updating -- in every open.

The *close* operation closes the database, deallocates any storage allocated by the open operation, and exits from the DBMS.

The *first, next,* and *insert* operations are usually all done via a single procedure call. The procedure has parameters for the subschema name, the name of the record, the name of the physical area in which the record sits, the control area in which currency is maintained so the request for a "next" record has some meaning, and, of course, the database operation. In IDMS, for example, the single subroutine is called IDMS; a call to obtain the next record would be

```
CALL IDMS (SUBSCHEMA_CTL, IDBMSCOM(11),
     RECNAME, AREANAME, ERROR)
```

where the IDBMSCOM number signifies the database operation.

Type 4 Database Access Templates

Type 4 DBMSs have a language that resembles a query language, although in some cases it is rather primitive. As a result, they use the same kind of template

as is used for Type 1 DBMSs: the query template. The process used to fill in the query template is described in detail in section on type 1 database access templates, and an example appears there.

Type 5 Database Access Templates

Type 5 databases are not, strictly speaking, databases at all. They have no query languages associated with them, and so the only approach to take is the inner interface approach. A brief discussion of *open, close, first, next* and *insert* follows. We will show that the only operations to be constructed by the interface builder for processing Type 5 databases are *first, next,* and *insert.*

The *open* routine simply opens the file upon which the data is residing. At execution, the name of the file is passed to the routine as a parameter. Any work data areas to be set up or error-reporting mechanisms to be established, are already built into the **DAVID** system for our generic file interface. So there is nothing for the template builder to do here.

The same is true for the *close* operation: the **DAVID** system already contains a generic file routine for closing the data file and deallocating any storage areas for that processing. So nothing needs to be done.

The entire template for accessing a database, then would consist of the commands:

```
@@read_first_record
@@read_next_record
@@insert_record
```

The *first* and *next* operations differ only in that the *first* routine must read past any header on the front of the file. So the template routines to be filled in for the *first* routine are:

```
@@read_past_header
```

and

```
@@read_next_record
```

Routine read_past_header simply bypasses any heading information on the front of the file, and positions us for the next read at the first piece of data on the file. This routine will be the same for every file on the same file system (our FITS interface, for example, contains a routine to read past any FITS header). If the file has no header, this routine will be null.

The object of the read_next_record routine is to place the data from one record of the file into a host data buffer. Once the data is in this buffer, the host system can then process it. The routine must go through the data items one at a time, performing pre-processing and data conversion where necessary and moving the data into the corresponding position in the proper **DAVID** data buffer. Preprocessing might include stripping leading characters or trailing characters from the data. Conversion might include translation from ASCII to binary data representation.

The *insert* operation moves the data from the host data buffer into a record of the file; therefore it is the opposite of the *next* operation. The components of the insert operation parallel those of next, but the data is moved in the other direction. Once one of the routines is written, it will be very clear what is needed to construct the other.

What to do with the templates you just constructed

Depending upon the Type of your DBMS, the database access templates you constructed are used in different ways. However, the Define, Delete and Install templates are used in the same way, regardless of what kind of database system you have. We explain their use first.

The set of templates you just constructed will become part of the **DAVID** system, as a "system cluster" of data which **DAVID** calls upon to do its work. Each time a request for a database operation on one of your databases comes to **DAVID**, the proper template -- the one for the correct operation on your DBMS -- must be retrieved and filled in by **DAVID**. The resulting text string contains a set of instructions to your DBMS. When that text string is executed, it will "bring up" your DBMS and perform the operation on it.

For all DBMS types, the DEFINE, DROP, INSTALL and DELETE templates are processed as described above. For database systems of Types 1 and 4, the database access (query processing) templates are also treated in this manner; however, for database systems of types 2, 3, and 5, they are used quite differently. Here, we outline that process.

The database access templates for database systems of types 2, 3, and 5, you may recall, generate not query-dependent templates to be filled in with query-specific information, but procedures. These procedures are general, i.e., they contain no query-specific information, and they have to be compiled and linked into the host DBMS -- once. So when the access routines are finished, they are delivered to the DAVID database administrator, who links them into the DAVID system, and makes some system changes to accommodate the routines. For DBMS Types 2, 3, and 5 you should skip the next section, since there will be no templates to install, and proceed with the section which discusses the testing of your new interface.

Installation into the Template Cluster

Each template you have generated should be in a separate text file. On the front of the file, 2 pieces of information must be added: the DBMS number, and the type of operation. The DBMS number is a distinct 3-digit number for each non-**DAVID** DBMS on the system; ORACLE, for example, is DBMS number 101, while INGRES is DBMS number 102. The **DAVID** database administrator must assign a number to the DBMS before the templates can be installed. The type operations are as follows (they are 8-character codes):

```
DEFINE
INSTALL
DROP
DELETE
SELPROJ
STORE
```

Note: the first four operations are for any DBMSs; the last two are for Type 1 and Type 4 DBMSs only.

There is a stand-alone program called load_template_cluster which asks for the name of the file in which the template sits; then reads that file and adds the data to the **DAVID** template cluster.

A few things must be done to the **DAVID** system before you can start testing your interface components. These operations will be performed by the database administrator of **DAVID**. They are:

Your DBMS must be assigned a 3-digit DBMS number, which is added to the resident database structure numbers [in clstruct.h]. It is described above.

The logical statements which determine which resident interface belongs to which query must be expanded to include your DBMS name and number.

Once these actions have been performed, and **DAVID** has been relinked by the database administrator, you are ready to begin testing (and using!) your interface.

Testing your interface components

You must test your interface components on a "real" database on your DBMS. To do this, you must first Define or Install a database of that type. Let's try Define first; you can do it through the **DAVID** system. You simply issue a **DAVID** DEFINE command; the end of the command says STORE AS XX(YYY); where XX is the name of the DBMS, and YYY is the name of the new database you wish to create. So if, for example, you wish to create an ORACLE database called DB1, your DEFINE command would end .... STORE AS ORACLE(DB1); See the **DAVID** users manual for the syntax of the complete DEFINE command.

You can also create a new database in your DBMS via a **DAVID** query. This way, it immediately gets loaded with data, also. Simply query an existing **DAVID** database and specify that the results be stored in a database in your DBMS. For example, to create database DB1 as an ORACLE database containing the same information as a **DAVID** database called DB0, issue this query:

```
CREATE ACTUAL CLUSTER n1.usr1.file1.db1
  SELECT * FROM n1.usr1.file1.db0
  WHERE all
  STORE AS ORACLE (DB1);
```

NB: n1 is the physical node on which your data resides
usr1 is your user id
file1 is the physical file on which the data resides

See your **DAVID** database administrator for help in deter-

mining these values.

Once your database has been created, and is connected to the **DAVID** system, you can test the following operations on it:

a) You can issue queries. (See the **DAVID** user's manual for the syntax of query commands). Your queries can transfer data from one database to another in your DBMS, from yours to **DAVID**, from **DAVID** to yours, or to and from yours and any other DBMS for which a **DAVID** interface exists.

b) You can browse through your database, looking at records one at a time, setting "windows" (selected fields) to restrict the number of fields and booleans to restrict the number of records. You can even insert records into your database using this browsing tool. See the Reading Room user's manual for how to do this. An important caveat: *You can use this feature only if your database interface is an inner interface, i.e., it was built at the table-row level.* DBMS interfaces that have query-level interfaces can only support queries; no browsing is permitted.

Summary - The Capabilities of your interface

Now that your database management system has been interfaced to the **DAVID** system, all your databases can be accessed, queried and updated via the **DAVID** system, and data can be transferred between the **DAVID** system and your database system, and between any other DBMS connected to **DAVID** and your database system. In short, your DBMS becomes just another **DAVID** database organization type, and your databases become just more **DAVID** databases of different types.

For every database you wish to process via the **DAVID** system, you must install it on **DAVID**, i.e., you must tell **DAVID** about its database schema. Once the installation has been done, the full spectrum of the DAVID query language is at your command.

Permissible operations other than queries on your databases are governed by the type of interface you have to the **DAVID** system, which in turn is determined by the TYPE classification you selected for your interface, based on the characteristics of your DBMS. The inner interfaces (for databases of types 3 and 4) have the most flexibility; you can issue

queries on them, as well as browse through them using the **DAVID** browsing tool. The outer interfaces are not endowed with browsing capability; however, GSQL is a very powerful query language, and provides a full range of database access capabilities.

Outlook for Phase III

Ken Wanderman & Associates is vigorously pursuing follow-on contracts to this project in two areas. The first area is using the results described in this report to enable us to act as consultants for parties who have data in various DBMSs and files, to interface that data to **DAVID**. In particular, in the astrophysics community within NASA there is great interest in using the **FITS** and **FILE** interfaces to translate data. The arrangements we are pursing would be to have our company perform the translations in house. Such translations might possibly involve writing additional interfaces depending on the nature of the data. We expect contracts to be signed within the next few months.

A second area we are pursuing is in the direct sale of our technology to commercial database management companies who would like to have interfaces of their product to others. The concept of interface between commercial relational databases has come alive within the time span of this project; and, several major companies such as FOCUS and ORACLE are now putting emphasis on interfaces between their products and other commercial DBMSs. Unfortunately, there are no commercial heterogeous distributed DBMSs, so our concept remains slightly ahead of its time. Our plan is to pursue consulting until the market catches up.

Suggestions for Further Research

In our strategies to build interfaces we only considered what one could call traditional type database systems; i.e., ones based on tables. These were the relational, the hierarchal and the network type DBMS. At this point, these traditional systems make up the overwhelming majority of systems in use by scientists and business and industry. However, the market for commercial databases is an active one. New products and technologies are constantly emerging as the demand for more sophisticated database products grows. Our approach accounts for new products in the traditional area, but DBMS based on entirely new techniques may call for interfaces inconsistent with our approach. Two types of database systems still in the development stage come to mind. One is Hypertext which allows a more sophisticated linking mechanism than traditional DBMSs. Interface to Hypertext systems should be considered for the future, particularly when they become available on machines larger than micro computers. A second type of DBMS is the object-oriented database. A leading vendor in this area is Servio Logic, Inc., whose product, Gemstone, has become the pioneering product. While time and resources of this contract did not allow us to pursue an interface with Gemstone, our company plans to make this effort in the near future.

Illustrations

Figure 3 - Typical FITS Header

```
SIMPLE  =                    T / File is standard FITS format
BITPIX  =                    8 / Character information
NAXIS   =                    0 / No image data array present
EXTEND  =                    T / There are standard extensions
COMMENT  AGK3 Astrometric catalog, in FITS Tables Format
COMMENT   extension header follows in a new block
END

XTENSION= 'TABLE   '           / Table extension
BITPIX  =                    8 / 8-bits per "pixel"
NAXIS   =                    2 / simple 2-D matrix
NAXIS1  =                   74 / no of characters per row
NAXIS2  =                    3 / no of rows
PCOUNT  =                    0 / no "random" parameters
GCOUNT  =                    1 / 1 group
TFIELDS =                   16 / 16 fields per row
EXTNAME = 'AKG3    '           / name of the catalog

TTYPE1  = 'NO      '           / the star number
TBCOL1  =                    1 / start in column 1
TBFORM  = 'A7      '           / 7 character field

TTYPE2  = 'MG      '           / stellar magnitudes
TBCOL2  =                    8 / start in column 8
TFORM2  = 'E4.1    '           / xx.x standard precision floating
point
TUNIT2  = 'MAG     '           / units are magnitudes

TTYPE3  = 'SP      '           / spectral type
TBCOL3  =                   13 / start in column 13
TFORM3  = 'A2      '           / 2 character field
TNULL3  = '        '           / blank is indefinite value

TTYPE4  = 'RAH     '           / right ascension hours
TBCOL4  =                   16 / start in column 16
TFORM4  = 'I2      '           / 2-digit integer
TUNIT4  = 'HR      '           / units are hours
TNULL4  = '99      '           / null value
```

```
TTYPE5  = 'RAM      '           / right ascension minutes
TBCOL5                       19 / start in column 19
TFORM5  = 'I2       '          / 2-digit integer
TUNIT5  = 'MIN      '          / minutes of time
TNULL5  = '99       '          / null value

TTYPE6  = 'RAS      '          / right ascension seconds
TBCOL6  =                    22 / starting in column 22
TFORM6  = 'E6.3     '          / xx.xxx standard precision float
TUNIT6  = 'S        '          / seconds of time

TTYPE7  = 'DECDSIGN'           / declination sign
TBCOL7  =                    29 / start in column 29
TBFORM  = 'A1       '          / character field

TTYPE8  = 'DECD     '          / declination degrees
TBCOL8  =                    30 / start in column 30
TFORM8  = 'I2       '          / 2 digit integer
TUNIT8  = 'DEG      '          / degrees

TTYPE9  = 'DECM     '          / declination minutes
TBCOL9  =                    33 / start in column 33
TFORM9  = 'I2       '          / 2 digit integer
TUNIT9  = 'ARCMIN   '          / minutes (angle)
TNULL9  = '99'                 / null value

TTYPE10 = 'DECS     '          / declination seconds
TBCOL10 =                    36 / start in column 36
TFORM10 = 'E5.2     '          / xx.xx standard precision float
TUNIT10 = 'ARCSEC   '          / seconds (angle)
TNULL10 = '99.99    '          / null value

TTYPE11 = 'EPOCH    '          / epoch of positions
TBCOL11 =                    42 / start in column 42
TFORM10 = 'E7.2     '          / xxxx.xx standard precision float
TUNIT11 = 'YR       '          / units are years

TTYPE12 = 'N        '          / no. photo obs.
TBCOL12 =                    50 / start in column 50
TFORM12 = 'i1       '          / one digit integer

TTYPE13 = 'RAPM     '          / proper motion in r.a.
TBCOL13 =                    52 / start in column 52
TFORM13 = 'E4.3     '          / .xxx standard precision float
TUNIT13 = 'ARCSEC.YR-1'        / units are arc-seconds/yr
TNULL13 = '9999     '          / NULL VALUE
```

```
TTYPE14 = 'DECPM    '           / proper motion in dec.
TBCOL14 =                    57 / start in column 57
TFORM14 = 'E4.0     '           / xxx. standard precision float
TUNIT14 = 'ARCSEC.YR-1'         / units are arc-seconds/yr
TSCAL14 =                 0.001 / scale factor = 0.001
TNULL14 = '999      '           / null value

TTYPE15 = 'DEPOCH   '           / difference in epoch AGK3-AGK2
TBCOL15 =                    62 / start in column 62
TFORM15 = 'E5.2     '           / xx.xx single precision float
TUNIT15 = 'YR       '           / unit is years

TTYPE16 = 'BD       '           / Bonner Durch. star number
TBCOL16 =                    68 / start in column 68
TFORM16 = 'A7       '           / 7-character field
TNULL16 = '         '           / blanks indicate null

AUTHOR  = 'W. Dieckvoss'
REFERENC= 'Hamburg-Bergsdorf 1975'
END
```

Figure 4 Assign File

```
/*************************************************************
*                                                            *
*   File: 6file_asgn.c                                       *
*   Module: scracc                                           *
*  Does extra things associated with assigning a general     *
*  file or FITS   cluster (like reading the auxiliary cluster *
*    into the CCA, and opening the file).                    *
*************************************************************
*************************************************************

     Function: file_asgncluster
     File:     6file_asgn.c
     Author:   M. Moroh
     Last Update: 8/88

  For general files & FITS, opens files and reads header
   from auxiliary cluster into CCA (new place: res_area)
  Called by routine arbi_asgncluster /scracc/6212.c

*************************************************************

 NOTE: This routine is called by the asg cluster process.  It
does NOT assign the main cluster, which is already done (c1 is
its ptr). It simply assigns the auxiliary cluster associated with
it,loads the header infor into main cca, and deassigns the aux
c1.

 This routine is also called by define cluster (since the define
operation for file-type clusters also has to assign it). To avoid
a big mess, this routine exits if it discovers the header to be
empty.  This means:  header isn't constructed yet!  So don't try
to read it!!! */

#include "chap4std.h"
/*  and utility.h
   JWF - was 'marsha.h' in original fits but
   that file has been merged with 'chap4sdt.h'
```

```
#define pmode 0777      /* file access mode for new cluster file
*/
#define VCA_EOT 60002   /* EOF on subcluster read */

file_asgncluster(v1,c1,c_name,query,access)

VCA *v1;            /* same VCA as usual */
CCA *c1;            /* CCA ptr for main cluster */
char *c_name;       /* cluster name of main cluster */
char *query;        /* if this is to be an "outie", i.e., a fcn
executed */
char *access;       /* whether aux cluster is to be read/write/up-
date */

/* note:  the parameter "query" will be null for most general
files.*/
/*  only for retrieval systems w. built-in functions will it be
nonnull */

{
 TCA *t1;            /* TCA ptr for table containing data */
 CCA *c2;            /* CCA ptr for the auxiliary cluster */
 SCA *sca;
 char *uid;
 char *pwd;
 FILE *fopen(), *fp;
 clust_struct *name_ds;     /* cluster name */
 FF_AREA *file_area;      /* pointer to structure for data areas
*/
 char *calloc();
 char the_name[CLUSTER_HDG_LEN + 2];
 int result = SUCCESS;
 int bind_flag = FALSE;

 AFIELD *eachfield;      /* for allocating fieldinfo  */
 AFIELD *last;           /* to find length of record */
 char openmode[3];       /* mode for opening file (r,w)  */
 int int_length;         /* int version of data length. To get
around */
                         /* USHORT problem (cast it; assign it to
*/
                         /* eachfield->length)  */

/*  Allocate data areas for file header info, etc.   */
   file_area = (FF_AREA *) calloc(1,sizeof(FF_AREA));

/*  Insert a pointer to the resident_areas into the CCA   */
    c1->res_area = file_area;
```

```
/*  Set TCA pointer (t1) to first table of CCA (we need it later)
*/
    t1 = c1->tca_ptr;
/*  Provide the userid and/or password  */
/*  (Is this ever necessary?)    */
   /*  uid = (char * ) calloc(1,NAME_LEN); */
   /*  pwd = (char * ) calloc(1,NAME_LEN); */
/*  Open up the general file  & store its ptr in the CCA */
/*  Open it for read, write or update, depending on access. */
/*    On the VAX it would be:  */

/*  JWF -  DAVID ops other than delete put a 'D' in the access
*/
/*  string - this will have to be resolved eventually but for now
*/
/*  checking will be disabled.
*/
/*{
    result = E_ILLEGAL_ACCESS_MODE;
    set_status(v1, E_ILLEGAL_ACCESS_MODE);
}      JWF    */

/*  NOTE: code below will allow for "RW" (though it'll actually
open "a" */
/*    in that case).  It won't alllow for "D"          */
/*  Allowing for RW provides multiple user access to same file (I
think) */
eachfield = NULL;
last = NULL;
if (result == SUCCESS)
{
     if (foundit(access,"RW") != NULL) strcpy(openmode,"a+");
     else if (strchr(access,'W') != NULL) strcpy(openmode,"a");
     else  /* cluster is for reading only */
           strcpy(openmode,"r");

     if ((fp = fopen(c1->res_name,openmode)) == NULL)
     {
        printf("\n unable to open file %s with access mode
%c%c\n",
              c1->res_name, openmode[0], openmode[1]);
        set_status(v1,E_ARBI_FILE_CREATE_ERROR);
```

```
        result = E_ARBI_FILE_CREATE_ERROR;
      }
}
/*  Now get the header information from the "auxiliary CCA" */
if (result == SUCCESS)
{
    c1->res_file_ptr = fp;
    /*  first, allocate a field description structure  */
    eachfield = (struct afield *) calloc(1,sizeof(struct
afield));
    eachfield->next = NULL;
    file_area->firstfield = eachfield;   /* hook structure to cca
*/
    /*  Now assign and read the secondary cluster for the file,
*/
    /*  which contains the header information.    */
    /*   create the name of the secondary cluster (routine
get_aux_name)
         it was created in the define routine (in file
arbi_hdr.c)   */
    get_aux_name(c_name,the_name);
    /*  for testing, a single-level table (omit device info) */
    /*  if ((result =
          bindcolumn(v1,c2,"device","device_info",
                     file_area->device_info, DCHAR,20 ))
    */
    if ((result = asgcluster(v1,&c2,the_name,"*","RWD")) != SUC-
CESS  ||
        (result =
          bindcolumn(v1,c2,"xheader","xname",eachfield->name,
                     DCHAR,NAME_LEN )) != SUCCESS ||
        (result =
         bindcolumn(v1,c2,"xheader","xtype",eachfield->type,
DCHAR, 8))
          != SUCCESS  ||
        (result =

bindcolumn(v1,c2,"xheader","xlength",&(eachfield->length), DINT,
0))
            != SUCCESS  ||
        (result =
```

```
bindcolumn(v1,c2,"xheader","xnodecs",&(eachfield->nodecs), DINT,
0))
             != SUCCESS ||
         (result =

bindcolumn(v1,c2,"xheader","xcolumn",&(eachfield->column), DINT,
0))
             != SUCCESS ||
         (result =
           bindcolumn(v1,c2,"xheader","xcomments",
                      eachfield->comments,DCHAR,200)) != SUCCESS)
           ;   /* Don't do anything */
}
/* Read information from header into auxiliary cluster */
if (result == SUCCESS)
{
        bind_flag = TRUE;
    if ((result = asgsubcluster(v1, c2, &sca, "*")) != SUCCESS)
    {
             dasgcluster(v1,&c2);
             result = CANT_ASGN_AUXSUBCL;
    }
}
if (result == SUCCESS)
{
    scrbfirst(v1,c2,sca,NULL);
    if (v1->status == VCA_EOT)
    { /* printf("\n in file_asgn, xheader cluster showed up
empty."); */
        /*  JWF - Clean up bindings.  */
    if ((result =
          unbindcolumn(v1,c2,"xheader","xlength",&(each-
field->length)))
         != SUCCESS ||
        (result =
          unbindcolumn(v1,c2,"xheader","xname",eachfield->name))
         != SUCCESS ||
        (result =
          unbindcolumn(v1,c2,"xheader","xtype",eachfield->type))
         != SUCCESS ||
        (result =
          unbindcolumn(v1,c2,"xheader","xnodecs",&(each-
field->nodecs)))
         != SUCCESS ||
        (result =
```

```
         unbindcolumn(v1,c2,"xheader","xcolumn",&(each-
field->column)))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xcomments",each-
field->comments))
        != SUCCESS)
        ; /* dont do anything */
          dasgcluster(v1,&c2);
          return(SUCCESS);
    }
    else if (v1->status != SUCCESS)
    {
          printf("\n in file_asgn, cant read 1st header record
properly");
          dasgcluster(v1,&c2);
          result = E_CANT_READ_ARBI_HDR;
     }
}
/* now unbind all cluster variables; then rebind. */
while (result == SUCCESS)
{
    if ((result =
          unbindcolumn(v1,c2,"xheader","xlength",&(each-
field->length)))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xname",eachfield->name))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xtype",eachfield->type))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xnodecs",&(each-
field->nodecs)))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xcolumn",&(each-
field->column)))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xcomments",each-
field->comments))
        != SUCCESS)
        ; /* dont do anything */
```

```
    else
    {
        /* set up another memory location for next field info  */
        last = eachfield;   /* keep track of last field */
        eachfield->next = (AFIELD *) calloc(1,sizeof(AFIELD));
        eachfield = eachfield->next;
        eachfield->next = NULL;

        if ((result =
          bindcolumn(v1,c2,"xheader","xname",eachfield->name,
DCHAR,NAME_LEN ))
            != SUCCESS ||
        (result =
         bindcolumn(v1,c2,"xheader","xtype",eachfield->type,
DCHAR, 8))
            != SUCCESS ||
        (result =

bindcolumn(v1,c2,"xheader","xlength",&(eachfield->length), DINT,
0))
            != SUCCESS ||
        (result =

bindcolumn(v1,c2,"xheader","xnodecs",&(eachfield->nodecs), DINT,
0))
            != SUCCESS ||
        (result =

bindcolumn(v1,c2,"xheader","xcolumn",&(eachfield->column), DINT,
0))
            != SUCCESS ||
        (result =

bindcolumn(v1,c2,"xheader","xcomments",eachfield->comments,
                        DCHAR,200))
            != SUCCESS)
        ; /* Dont do anything */
        else
            /*  now read the next row of the auxiliary cluster */
           result = scrbnext(v1,c2,sca,NULL);
    }
) /* end of while loop */

/* Check for normal ending of while loop */
if (v1->status == VCA_EOT)
    result = SUCCESS;
```

```
/* now unbind all cluster variables */
if (result == SUCCESS || bind_flag)
{
    if ((result =
          unbindcolumn(v1,c2,"xheader","xlength",&(each-
field->length)))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xname",eachfield->name))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xtype",eachfield->type))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xnodecs",&(each-
field->nodecs)))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xcolumn",&(each-
field->column)))
        != SUCCESS ||
       (result =
         unbindcolumn(v1,c2,"xheader","xcomments",each-
field->comments))
        != SUCCESS)
        ;
}
/* if there's one extra instance of eachfield.   Free it.  */
if (last != NULL) last->next = NULL;
if (eachfield != NULL)
    cfree(eachfield);       /* JWF */

/* Figure out the max record length (not necessary to rd NAXIS
then) */
/* --it's the column of the last record + the length of the last
rec */
/*  NOTE:  statement below seems to point to FIRST field, not
last.  */
/*  That's because DAVID stores records BACKWARDS, so first IS
last.  */
/*   Note from KW, however:  should really sort on columns,
because */
/*    it's not necessariliy last; could sometimes be a different
order. */
```

```
if (result == SUCCESS)
{
     file_area->maxlen = (file_area->firstfield->column) +
                         (file_area->firstfield->length) - 1;
     dasgcluster(v1,&c2);

/*  very last thing to do before leaving:  calculate offsets for
*/
/*  buffer fields.  Routine sort_offsets, below,  does this.  */
    result = sort_offsets(file_area->firstfield, t1);
}
return(result);
}   /* end file_asgncluster */
```

Figure 5 File First and Next

```
#include "chap4std.h"
#include <ctype.h>
/*****************************************************************/
/*                                                               */
/*    File: 6file_ops.c                                          */
/*    Module: scracc                                             */
/*                                                               */
/*    contains functions read, write & insert (plus utilis)      */
/*    for FITS and general file clusters                         */
/*     Routines:                                                 */
/*                        file_insert                            */
/*                        file_first                             */
/*                        file_next                              */
/*****************************************************************/
/*****************************************************************/
/*                                                               */
/*     Function: file_first                                      */
/*     File:     6file_ops.c                                     */
/*     Author:   M. Moroh                                        */
/*     Last Update: 8/88                                         */
/*                                                               */
/*  Read first record of general files and files of FITS         */
/*  Called by  /scracc/62i.c routine generic_function            */
/*                                                               */
/*****************************************************************/
char *calloc();
void free();
#define NEW_LINE '\n'
/*   read first record of a generic file into a buffer   */
file_first(v1,c1,t1)
VCA *v1;                              /* vca pointer */
CCA *c1;                              /* pointer to CCA */
                        /*for host version of resident cluster */
TCA *t1;                /* tca for table corresponding to file */
{
int result;
```

```
/*   first, rewind the tape (or reposition the disk, or whatever)
*/
/*  NOTE:  This routine is probably system dependent. */
result = file_rewind(c1->res_file_ptr);
if (result == SUCCESS)
{
        result = read_past_hdr(c1);
        if (result == E_EOF_ON_FILE_HDR) result = file_rewind(c1-
>res_file_ptr);
         if (result == SUCCESS)
                result = file_next(v1,c1,t1);
}
return(result);
} /* end file_first */
/*************************************************************/
/*                                                           */
/*     Function: file_next                                   */
/*     File:     6file_ops.c                                 */
/*     Author:   M. Moroh                                    */
/*     Last Update: 4/90 - KW add binary types               */
/*                                                           */
/*                                                           */
/*  Read next record of general files and files of FITS      */
/*  Called by  /scracc/62i.c routine generic_function        */
/*                                                           */
/*************************************************************/
/*  Routine to  read the next record of a generic file into
buffer */
/*  remember - assign file already read file header cluster */
/*   and puts header info into c1->res_areas     */
/*************************************************************/
file_next(v1,c1,t1)
VCA *v1;              /* vca pointer */
CCA *c1;              /* pointer to CCA for host version of
resident cluster */
TCA *t1;              /* tca for table corresponding to file */

{
AFIELD *eachfield;     /* for allocating fieldinfo  */
AFIELD *last;          /* to find length of record */
AFIELD *firstfield;    /* beginning of record (1st field) */
int length;                   /* for calculating field lengths */
```

```
 char *datarec;                    /* buffer for a record of data */
 char *item;                       /* buffer for an item of data  */
 FIELD *fpointer;
 FIELD *find_data_item();          /* function find... returns ptr to
 FIELD */
 int itemlen;                      /* length of a field */
 int input_length;
 char *place;
 int k;
 char *file_format;
 int i;
 int c;
 int anint;
 float afloat;
 int errorcode = OK;
 union {
         char c[4];
         short s;
         int   i;
 } utype;

 /*  allocate storage for the file record */
 /*  length was put into cca special area c1->res_area by table-
row assign */
 /*  also allocate storage for a data item (max size: whole
record) */
 /* printf("\n With a resident file pointer of %d", c1-
>res_file_ptr); */

 /*  printf("\n logical pointers in buffer: t1->nlptrs is %d, t1-
>npptrs is %d",
     t1->nlptrs, t1->npptrs);   */

 input_length = c1->res_area->maxlen + 1;
 datarec = (char * ) calloc(1,input_length);
 item  = (char * ) calloc(1, input_length);

 /*************************************************************/
 /*  read next record of the file & get it into the TCA buffer  */
 /*************************************************************/
 i=0;
 c=EOS;
 while (( i < input_length-1 ) && (c != EOF))
 {
      c =  getc(c1->res_file_ptr);
      if ((c != '\n') && (c!= EOF)) datarec[i++] = c;
     /* maybe in future versions more ending characters should be
```

```
included */
}
datarec[i++] = EOS;
/* if (c != EOF) printf("\n the data record image: %s", datarec);
*/
if (c == EOF)   /* end of file */
        errorcode = EOT;

/* now go through the data, field by field. Convert it to the */
/* corresponding host format and move it to the host buffer */
else
{
        firstfield = c1->res_area->firstfield;
        eachfield = firstfield;   /* start at beginning of field
list */
}
/*  REMEMBER:  FIELDS COME OUT BACKWARDS */
while (eachfield != NULL && errorcode == OK)
{
     /* convert the data to the corresponding host data type
*/
     /* the file data item is called item; the host item, */

     place = (datarec - 1) + eachfield->column; /* ptr to place
in record */
     bfwdcopy(item,place, eachfield->length);   /* copy from file
record  */
     *(item+eachfield->length) = EOS;           /* add end of
string mark */
    /* printf("\n item to decode is %s; its len is %d",i-
tem,strlen(item)); */
    /* for testing:  skip conversion table.  Use only DAVID data
types. */
    /*  Do this by giving the ce pointer (to conversion table) a
null value. */
    c1->res_area->ce = NULL;     /* for testing -- see above
lines */
    /* find the name of the data item in the TCA of the main
cluster */
        /* fpointer is a ptr to the FIELD in the TCA of the main
cluster */
    /* printf("\n in file_next, about to look for field
%s",eachfield->name);*/
    fpointer = find_data_item(t1,eachfield->name);
```

```
    if (fpointer == NULL)
    {
       /* printf("\n CANT FIND FIELD IN ARBI FILE: %s",eachfield-
>name); */
       errorcode = E_CANT_FIND_ARBI_FIELD;
    }

    /* it may be necessary to convert the FILE's format types to
lower case */
    file_format = calloc(1,strlen(eachfield->type) +  1);
        strcpy(file_format,eachfield->type);
    lower_case(file_format);

    /* Transfer the data to the DAVID buffer */

    /*********File and DAVID are both char or or both num
***************/
    if (((strncmp(file_format, "char",4) == 0) && (fpointer->type
== TCHAR)) ||
       ((strncmp(file_format, "num", 3) == 0) && (fpointer->type
== TNUM)))
       /* stick the char string item right in the buffer */
    {
        bfwdcopy((t1->buf_ptr + *OFFSET(t1,fpointer->id)),item,
strlen(item));
    }

    /**** File is ASCII integer; DAVID is integer data ********/
    else if (((strncmp(file_format,"int",3)==0) && (fpointer-
>type == TINT)) ||
             ((strncmp(file_format,"num",3)==0) &&
(fpointer->type == TINT)))
    {
        sscanf(item,"%d",&anint);
        /* put the host version of the data item into the host
buffer  */
        bfwdcopy ((t1->buf_ptr + *OFFSET(t1,fpointer->id)),
                  ((char *)&anint),fpointer->length);
        }

        /**** File is Binary integer I4; DAVID is integer ****/
        else if(((strncmp(file_format,"i4",2)==0) && (fpointer-
>type==TINT)) ||
                          ((strncmp(file_format,"i4",2)==0) &&
(fpointer->type==TNUM)))
        {
         bfwdcopy((t1->buf_ptr + *OFFSET(t1,fpointer->id)),
                 item,fpointer->length);
```

```
        }
        /**** File is binary integer I2; DAVID is integer INT
***/
        else if ((strncmp(file_format,"i2",2)==0) && (fpointer-
>type==TINT))
        {
                /* First convert to short integer */
                bfwdcopy(utype.c,item,2);
                /* convert to regular integer */
                anint = utype.s;
                /* Copy to DAVID buffer */
                bfwdcopy((t1->buf_ptr + *OFFSET(t1,fpointer-
>id)),(char *) &anint,sizeof(int)); }
    /**** FILE is ASCII floating point and DAVID is floating
point */
        else if ((strncmp(file_format,"float",5)==0) &&
(fpointer->type == TFLOAT))
        {
         sscanf(item,"%f", &afloat);
         bfwdcopy ((t1->buf_ptr + *OFFSET(t1,fpointer->id)),
                  ((char *)&afloat),fpointer->length);
    }
        /**** FILE is binary 4 byte floating and DAVID is
floating point ****/
        else if ((strncmp(file_format,"r4",2)==0) && (fpointer-
>type ==TFLOAT))
        {
         bfwdcopy((t1->buf_ptr + *OFFSET(t1,fpointer->id)),item,
                                  fpointer->length);
        }  /* end of floating point handling */
    else
    {  /* An unexpected condition has occurred */
        printf("Unexpected condition-Data mismatch in main + aux
clusters:\n");
        printf("Item = %s, type in aux = %s, type in DAVID is
%d",
                item,file_format,fpointer->type);
    }
     /* now do the same for the next field in the file record */
    cfree(file_format);          /* JWF */
    eachfield = eachfield->next;

}  /*  end of WHILE loop */
```

```
cfree(item);                          /* JWF */
cfree(datarec);                       /* JWF */
return(errorcode);
}       /* end of file_next   */
```

Figure 6 - File Insert

```
/*************************************************************/
/*                                                           */
/*     Function: file_insert                                 */
/*     File:     6file_ops.c                                 */
/*     Author:   M. Moroh                                    */
/*     Last Update: 4/90                                     */
/*                  Add binary types                         */
/*                                                           */
/*  Routine to insert records into a generic file from       */
/*    DAVID buffer                                           */
/*  Called by  /scracc/62i.c routine generic_function        */
/*                                                           */
/*************************************************************/

file_insert(v1,c1,t1)
VCA *v1;              /* vca pointer */
CCA *c1;              /* pointer to CCA */
TCA *t1;              /* tca for table corresponding to file */
{
  AFIELD *eachfield;     /* for allocating fieldinfo  */
  AFIELD *last;          /* to find length of record */
  AFIELD *firstfield;    /* beginning of record (1st field) */
  int length;            /* for calculating field lengths */
  char *datarec;         /* buffer for a record of data */
  char *item;            /* buffer for an item of data  */
  FIELD *fpointer;
  FIELD *find_data_item();
  int itemlen;               /* length of a field */
  int input_length;
  char *place;
  int k;
  char *file_format;
  int i;
  int c;
  int anint;
  float afloat;
  char *s;
  char *ptr;
  char *temp;
  int *int_ptr;
  int errorcode = OK;
  short *short_ptr, ashort;
union {
```

```
        char c[4];
        short s;
        int   i;
} utype;

/*  allocate a data record (datarec) for file stuff, */
/*   and a variable (item) for each field to get from DAVID.  */

input_length = c1->res_area->maxlen + 1;
datarec =  calloc(1,input_length);
temp = calloc(1,input_length);

/* Initialize the buffer to blanks */
for (i = 0; i < input_length; i++)
      *(datarec + i) = ' ';


/* go through the data, field by field in the aux cluster. Con-
vert it to the */
/* corresponding resident format (if nec) and move it to the host
buffer */

firstfield = c1->res_area->firstfield;
eachfield = firstfield;   /* start at beginning of field list */
                          /*  REMEMBER:  FIELDS COME OUT
BACKWARDS */

while (eachfield != NULL &&
           (fpointer = find_data_item(t1,eachfield->name)) !=
NULL)
{
     /* printf("\n the field we're working on is %s", eachfield-
>name); */

     /* place is a pointer to the position of this field in the
data record */
     place = (datarec - 1) + eachfield->column;

     /* for testing:  skip conversion table.  Use only DAVID data
types. */
     /*  Do this by giving the ce pointer (to conversion table) a
null value. */
     c1->res_area->ce = NULL;     /* for testing -- see above
lines */

        /* it may be necessary to convert the FILE's format types
to lower case */
        file_format = calloc(1,strlen(eachfield->type) +  1);
        strcpy(file_format,eachfield->type);
        lower_case(file_format);
```

```
        /********** DAVID data was CHAR or NUM. File is ASCII
**************/
        if (((strncmp(file_format, "char", 4) == 0)  &&
(fpointer->type == TCHAR)) ||
        ((strncmp(file_format, "num", 3) == 0) && (fpointer->type
== TNUM)))
        {
                /* Copy the data from the DAVID buffer to
datarec.  Note that the
          DAVID buffer item will contain an EOS if it less than
the maximum
          field length; hence the following routine:
                */
                ptr = (char *) (t1->buf_ptr +
*OFFSET(t1,fpointer->id));
                copy_no_eof(place,ptr,fpointer->length);
        }     /* end of if item is char or num */
    /**** DAVID data type is INT. File is ASCII (called INT OR
NUM)****/
        else if (((strncmp(file_format,"int",3) == 0) &&
(fpointer->type == TINT)) ||
          ((strncmp(file_format,"num",3) == 0) && (fpointer->type
== TINT)))
        {
        bfwdcopy((char *)&anint, (t1->buf_ptr +
*OFFSET(t1,fpointer->id)),
                sizeof(int));
        /* printf("\n INTEGER value about to be put on file is
%d\n", anint); */
        sprintf(temp,"%*d", eachfield->length, anint);
        copy_no_eof(place,temp,strlen(temp));

                /*      for (k = 0; k < eachfield->length; k++)
            printf(place + k);
                */
        }  /* end of if item is int */
        /****DAVID type is INT; FILE type is I4 binary integ-
er****/
        else if ((strncmp(file_format,"i4",2)==0) &&
(fpointer->type == TINT))
        {
                bfwdcopy ((char *)&anint,(t1->buf_ptr +
*OFFSET(t1,fpointer->id)),
                                        sizeof(int));
```

```
                ptr = (char *) (&anint);
                bfwdcopy(place,ptr,sizeof(int));
        }
        /****DAVID type is int; FILE type is I2 binary integer
****/
        else if ((strncmp(file_format,"i2",2)==0) && (fpointer-
>type == TINT))
        {
                /* copy to an integer for alignment */
                bfwdcopy ((char *)&anint,(t1->buf_ptr +
*OFFSET(t1,fpointer->id)),
                                        sizeof(int));
                /* convert to short */
                ashort = anint;
                ptr = (char *) (&ashort);
                bfwdcopy(place, ptr,2);
        }
        /******* DAVID data type is float; file type is float but
data's ascii ***/
        else if ((strncmp(file_format,"float",5) == 0) &&
(fpointer->type == TFLOAT))
        {
                sprintf(temp,"%*.*f",13,6,*(t1->buf_ptr +
*OFFSET(t1,fpointer->id)));
                bfwdcopy((char *)&afloat, (t1->buf_ptr +
*OFFSET(t1,fpointer->id)),
                sizeof(float));
                sprintf(temp,"%*.*f", eachfield-
>length,eachfield->nodecs, afloat);
                copy_no_eof(place,temp,strlen(temp));
        }
        /****DAVID type is FLOAT and FILE type is binary float
(R4) ****/
        else if ((strncmp(file_format,"r4",2)==0) && (fpointer-
>type==TFLOAT))
        {
                /* Both are binary so just byte copy one to other
*/
                ptr = (char *) (t1->buf_ptr +
*OFFSET(t1,fpointer->id));
                bfwdcopy(place,ptr,fpointer->length);
        } /* end if item is float
```

```
        /*  here add code to decode other data types.  */
        else printf("\n item %s didn't match appropriate DAVID
data type",
                                eachfield->name);
        /* Prepare for next iteration of field loop */
        eachfield = eachfield->next;
        cfree(file_format);                /* JWF */
}   /*  now do next item    */
if (fpointer == NULL)
        errorcode = E_CANT_FIND_ARBI_FIELD;
else
{
        /* now write the record.   The contents are in datarec.
*/
        for (i = 0; i < input_length - 1; i++)
        {
                c = *(datarec + i);
            putc(c,c1->res_file_ptr);
        }
        c = NEW_LINE;
        /* Put a newline on the end of the record for readability
*/
        /* NOTE: change DEFINE for NEW_LINE if don't want one.
Some files dont?*/
        putc(c,c1->res_file_ptr);
}
/* Print the record for debugging
*(datarec+input_length-1) = EOS;
printf("\n the record just written is %s", datarec); */
/* Cleanup and return */
cfree(datarec);                    /* JWF */
cfree(temp);                       /* JWF */
return(errorcode);
} /* end file_insert */
```

Figure 7 - File Deassign

```
/*****************************************************/
/* File is 62E.C  **   GENERIC DEASSIGN CLUSTER    */
/*****************************************************/
#include "chpt6std.h"

gdeasgncluster(vca,cca)

VCA *vca;
CCA *cca;

{

int errorcode;
CCA *cluster_ptr,*next_cluster;

/*********************/
/* Beginning of code */
/*********************/
errorcode = OK;

/*if (cca->structure >= FIRST_ARBI)    JWF - will do for
ARBI_FITS only
                                       for now.    */
if (cca->structure == ARBI_FITS)
    errorcode = arbi_deasgncluster(vca,cca);

else
if ((errorcode == OK) && ((errorcode=david_deasgnclus-
ter(vca,cca))==OK))
    errorcode=deassign_cluster(vca,cca);  /* chpt 7 deassign */


if (errorcode == OK)
{
    /* Delete cluster pointer from chain in VCA */
    next_cluster=cluster_ptr=vca->clusters;
    /* Find the cluster's position in the chain */
    while (next_cluster != NULL)
    {   if (next_cluster == cca) break;
        else
           {
           cluster_ptr=next_cluster;
           next_cluster=cluster_ptr->next;
           }
    }/* end of while */
    if (cluster_ptr == next_cluster)
    {
      if (cluster_ptr == NULL) vca->clusters = NULL;
```

```
      else vca->clusters = cluster_ptr->next;
    }
    else if (next_cluster == cca) cluster_ptr->next = cca->next;
}
else set_status(vca,errorcode);

return(errorcode);
} /* end of gdasgncluster */
/*****************************************************/
/* File is 62E1.C  **   DAVID DEASSIGN CLUSTER    */
/*****************************************************/
david_deasgncluster(vca,cca)
VCA *vca;
CCA *cca;
{
CCA *next_cluster;
int errorcode;
/**********************/
/* Beginning of code */
/**********************/
/* Close the file, if this is the only assigned cca from file. */
next_cluster=vca->clusters;
while (next_cluster != NULL)
  if ((next_cluster != cca) && (next_cluster->file == cca->file))
break;
  else next_cluster = next_cluster->next;
if (next_cluster == NULL) close_file(vca,cca->file);

/* Deallocate all the memory which was allocated by assign clus-
ter */
if (strncmp(cca->name,"DIRECTORY",9) == 0)
  { free_cca(cca); errorcode = OK; }
else
  if (strncmp(cca->verify,CCA_VERIFY,sizeof(cca->verify))==0)
    {
    alloc_deassign(cca);
    errorcode=OK;
    }
  else
    errorcode=E_NOT_CCA;
```

```
return(errorcode);
} /* end of david_deasgncluster */
/*************************************************************/
/* File is 62E2.C  **  ARBI DEASSIGN CLUSTER                 */
/* Code added 8/88 by MM to do the ARBI part of assign cluste*/
/*************************************************************/
arbi_deasgncluster(vca,cca)
VCA  *vca;
CCA  *cca;
{  /* begin function */
int errorcode = OK;
/*********************/
/* Code begins here */
/*********************/
FF_AREA *file_area;
AFIELD  *eachfield, *next_one;
/*************************************************/
/* FILE *fopen();
int k;
char item[4];
*/
/* If a resident with a file, close the file (its ptr is in the
CCA) */
  if (fclose(cca->res_file_ptr) != 0) errorcode = E_RES_FI-
LE_CLOSE;

/* Cleanup -- Free allocated storage */

eachfield = cca->res_area->firstfield;
next_one = eachfield->next;
while (eachfield != NULL)
{
    cfree(eachfield);                    /* JWF */
    eachfield = next_one;
    if (eachfield != NULL)               /* JWF  02/01/90 */
      next_one = eachfield->next;
}

cfree(cca->res_area);                     /* JWF */
```

```
if (strncmp(cca->verify,CCA_VERIFY,sizeof(cca->verify))==0)
    alloc_deassign(cca);
else
    errorcode=E_NOT_CCA;
return(errorcode);
} /* end arbi_deasgncluster */
```

Figure 8 - Generic Template for Definition Generator

```
Note: Blank lines are for readability only!!


/*********  This section defines the database  ********/
@@invoke_dbms                &    /* these 3 commands bring up the
DBMS */
@@give_password  @RUID @@separator  @RPWD /* and give it the
definition */
/**********  Above section defines the database  *********/

/*************************************************************/
/****  Below section provides the database definition  ***/
@@dbname_keyword  @@dbname_prefix @DBNAME  @@dbname_suffix  &
@@db_left_delim                              &
/*  below is for relational DBMSS */
@BEGINTABLE<@@table_separator><@@table_left_ delim>          &
 @@table_name_keywd @@tbname_prefix  @TABLE_NAME   @@tbname_suf-
fix
/* above fields are for relational DBMSs  */

@BEGINFIELD<@@field_separator><@@field_left_delim>&
        <@@field_rt_delim>
    @@field_attribute_left_delim                  &
    @@field_name_keywd @RFNAM @@field_attribute_separator &
    @@field_type_keywd @RFTYP  &
    @BEGINLENGTH<@@field_length_sep><@@field_length_left_delim>
&
         <@@field_length_rt_delim> @RFLEN  @ENDLENGTH   &
     @@field_attribute_separator                      &
    @@field_size_keywd @RFLEN
     @BEGINFPOS @@field_attribute_separator &
    @@field_start_pos_keyword @RFPOS @ENDFPOS
     @BEGINFCOM @@field_attrib_sep  &
    @@field_comment_keywd @RFCOM @ENDFCOM  &
    @@field_attribute_right_delimiter  &
     @@field_suffix
```

```
@ENDFIELD                                                          &
@@field_terminator
@ENDTABLE                                                          &

@@db_right_delim
@@db_terminator
@@exit              /* this command exits the database */
```

COMMENTS:

Above is a generalized template. It was made to include all possilities. Items may be absent for a specific DBMS, and order of items may be different from the way they're specified above. Keywords may either precede or follow the associated value.

Note: there are no longer default assumptions about carriage returns; they will be specifically inserted into the DBMS-dependent template as "\n"s.

Some of these fields are database type specific. For example, PARENT only occurs in type 2 (hierarchical) databases. Network stuff (owner, member) only appears in type 3 (network) databases. Schema, file and area will not appear in type 1 (relational), type 4 (micro) or type 5 (generic file) DBMSs.

The item "header" is for a description of an item that appears before the loop for that item: e.g., FIELD DESCRIPTION. It'll appear exactly once, no matter how many fields there are.

The field length information is filled in from the datatype table. That table will accompany the template.

NOTE: The separators and delimiters appear in the template exactly as they will in the final template, i.e., the @@ symbols will be filled in with real items like "," rather than symbols for those items.

Figure 9 Defining an ORACLE database

```
$UFI  user1/mypwd
CREATE TABLE courses
  (dept char(4),
   course char(9),
   sec# number(2));
CREATE TABLE teachers
  (id# number(4),
   instruct char(10),
   dept char(4));
CREATE TABLE students
  (studid number(4),
   studname char(30),
   major char(4));
CREATE TABLE transcripts
  (semester char(6),
   year number(4),
   studid number(4),
   course char(9),
   sec# number(2),
   grade char (1));
EXIT
```

Figure 10 - Generalized ORACLE Template

```
@BEGINTABLE<\n>
create @TABLENAME (\n
@BEGINFIELD<,\n>
@RFNAM = @RFTYP
@BEGINLENGTH<><><> @RFLEN @ENDLENGTH
@ENDFIELD )
@ENDTABLE
\n
```

Figure 11 GENERALIZED TEMPLATE FOR Type 2 Database Definition

Note: Blank lines are for readability only!!

```
/*********  This section defines the database  ********/
@@invoke_dbms                &     /* these 3 commands bring up the
DBMS */
@@give_password  @RUID @@separator  @RPWD /* and give it the
definition */
/**********  Above section defines the database  *********/

/*************************************************************/
/*******  Below section provides the database definition  ******/
@@dbname_keyword  @@dbname_prefix @DBNAME  @@dbname_suffix  &
@@db_left_delim                              &
/*******  Below 2 statements are for hierarchical/network DBMSs
***/
@@file_name keyword   @file_name      @@file_name_terminator
&
@@area_name keyword   @area_name      @@area_name_terminator
&
/*******  Above statements are for hierarchical/network DBMSs
***/

/* below fields are for hierarchical/network DBMSs */
@@table_header   /* mt takes care of this as text */     &
@@BEGINTABLE<@@table_separator><@@table_left_delim>&
      <@@table_rt_delim
@@table_attr_left_delim          &
  @@table_name_keywd @@tbname_prefix @TABLE_NAME @@tbname_suffix
&
  @BEGINPAR  @@table_attr_sep @@parent_keywd @PARENT_NAME  @END-
PAR
  @BEGINTLEN @@table_attr_sep    &
  @@table_length_keywd @TABLE_LENGTH @ENDTLEN
  @BEGINTPOS @@table_attribute_sep &
  @@table_start_pos_keywd @TABLE_START_POS @ENDTPOS
  @BEGINTCOM @@table_attr_sep  &
```

```
  @@table_comment_keywd   @TABLE_COMMENT  @ENDTCOM    &
 @@table_attribute_right_delim
/* above fields are for hierarchical DBMSs */

/*  below is for relational DBMSS */
@BEGINTABLE<@@table_separator><@@table_left_ delim>          &
 @@table_name_keywd @@tbname_prefix  @TABLE_NAME   @@tbname_suf-
fix
/* above fields are for relational DBMSs  */

@BEGINFIELD<@@field_separator><@@field_left_delim>&
    <@@field_rt_delim>
    @@field_attribute_left_delim
     @@field_name_keywd @RFNAM @@field_attribute_separator &
    @@field_type_keywd @RFTYP  &
    @BEGINLENGTH<@@field_length_sep><@@field_length_left_delim>
&
        <@@field_length_rt_delim> @RFLEN  @ENDLENGTH   &
     @@field_attribute_separator                      &
    @@field_size_keywd @RFLEN
     @BEGINFPOS @@field_attribute_separator &
    @@field_start_pos_keyword @RFPOS @ENDFPOS
     @BEGINFCOM @@field_attrib_sep  &
    @@field_comment_keywd @RFCOM @ENDFCOM  &
    @@field_attribute_right_delimiter  &
     @@field_suffix
@ENDFIELD                                          &
@@field_terminator
@ENDTABLE                                          &

/*************  below-only for NETWORK/HIERARCHICAL DBMSs
*************/
@@network_header     /* mt takes care of this as text to copy*/
&
@BEGINSET<@@set_sep><@@set_left_delim><@@set_right_delim> /*mt:
no*/ &
@@set_name_keyword  @SET_OWNER_NAME @@set_name_sep @SET_MEM-
BER_NAME  &
   /* note: in previous line, order can be reversed */
@@set_attribute_left_delim   /* mt: no */  &
@@set_owner_keyword @SET_OWNER_NAME @@set_attribute_separator  &
@@set_member_keyword @SET_MEMBER_NAME      &
@@set_attribute_right_delim  /* mt: no */  &
@ENDSET                                    &
```

```
/**********  above - only for NETWORK/HIERARCHICAL DBMSs
**********/

@@db_right_delim
@@db_terminator
@@exit                  /* this command exits the database */
```

COMMENTS:

Above is a generalized template. It was made to include all possilities. Items may be absent for a specific DBMS, and order of items may be different from the way they're specified above. Keywords may either precede or follow the associated value.

Note: there are no longer default assumptions about carriage returns; they will be specifically inserted into the DBMS-dependent template as "\n"s.

Some of these fields are database type specific. For example, PARENT only occurs in type II (hierarchical) databases. Network stuff (owner, member) only appears in type III (network) databases. Schema, file and area will not appear in type I (relational), type IV (micro) or type V (generic file) DBMSs.

The item "header" is for a description of an item that appears before the loop for that item: e.g., FIELD DESCRIPTION. It'll appear exactly once, no matter how many fields there are.

The field length information is filled in from the datatype table. That table will accompany the template.

NOTE: 3/20/89
The separators and delimiters appear in the template exactly as they will in the final template, i.e., the @@ symbols will be filled in with real items like "," rather than symbols for those items.

Figure 12 Typical TEMPLATE FOR Type 2 Database

```
DBD NAME=@DBNAME-1 \n
@BEGINTABLE<\n>
SEGM NAME=@TABLENAME ,PARENT=@PARENTNAME ,BYTES=@TABLELENGTH \n
@BEGINFIELD<\n>
FIELD NAME=@RFNAM
@BEGINLENGTH<><,BYTES=><> @RFLEN @ENDLENGTH
,START=@RFPOS
@ENDFIELD
@ENDTABLE
\n
```

Appendixes

Appendix 1 - Installing Template Generator Software

The template generator is written using a software platform called the Intelligence/Compiler, a product of Intelligence-Ware, Inc. The user is supplied with a runtime version only.

The steps listed below explain what must be done to install the
software.

1. The AUTOEXEC.BAT file in the root directory must be modified so that the path is extended and so that a line is added to set an environment variable for the Intelligence/Compiler. Include the following:

```
PATH = <whatever was in path previously>;C:\IC;
SET ICPATH=C:\IC
```

2. The CONFIG.SYS file in the root directory must be modified to provide for enough files and buffers: (The numbers shown here are the minimum to be specified.)

```
FILES = 12
BUFFERS = 8
```

3. Make a new directory under the root. Call it IC. (If a different directory name is used, change the modifications to your AUTOEXEC.BAT accordingly.) The following set of files are supplied to be put in the \IC directory:

```
ICX.EXE
ERRORS.SYS
HINSTALL.EXE
IC.BAT
IC.DD
IC-COLOR.EXE
ICP.EXE
```

4. Make three additional directories if you want to generate all three types of templates: for defining a database in a relational database language, for defining a database in a hierarchical or network database language, and for database queries.

The first directory will contain the module to run sessions of type 1 and 2a (see overview). The module generates templates for defining databases in relational database languages. I suggest this directory be called \IC**DAVID** and will refer to it below using this name. However, a different directory name could be used.

The second directory will contain the module to run sessions of type 1 and 2b. The module generates templates for defining databases in hierarchical or network database languages. I suggest this directory be called \IC\DAVIDH and will refer to it below using this name. However, a different directory name could be used.

The third directory will contain the module to run sessions of type 3 and 4. The module generates templates for database queries. I suggest this directory be called \IC\DAVIDQ and will refer to it below using this name. However, a different directory name could be used.

A unique set of files is supplied for each of the three directories. The installation disks are clearly marked indicating which files should be saved in \IC**DAVID**, which in \IC\DAVIDH, and which in \IC\DAVIDQ. All of these files are of the form *.BAT or *.ICP or *.FRM or *.LST .

5. For each of the directories made in Step 4, there must be a subdirectory called EXAMPLE. Therefore if in Step 4 you created the directory called \IC**DAVID**, there should now be a directory called \IC**DAVID**\EXAMPLE.

Similarly create \IC\DAVIDH\EXAMPLE and \IC\DAVIDQ\EXAMPLE if indicated.

These directories will be used to save examples entered during previous sessions so that they can be reused and modified as desired. A few samples are included in the installation disks and are clearly marked indicating which files should be saved in \IC**DAVID**\EXAMPLE, which in \IC\DAVIDH\EXAMPLE, and which in \IC\DAVIDQ\EXAMPLE.

Appendix 2 - Running TGS

If you want to generate a template for a defining a database in a relational database language:

1. Get into the directory \IC\DAVID.

2. First, run a session to generate log-on and log-off templates. Start this session by typing DAVID0 (ending with the number 0).

3. Second, run a session to generate the complete template. Start this session by typing **DAVID** or BIGDAVID. **DAVID** is appropriate for entering information about a database language which is not too verbose: i.e. The language does not use a lot of punctuation or long keywords, etc. BIGDAVID is appropriate for more verbose languages. If you get an error message saying you ran out of memory during a session started by typing **DAVID**, rerun the session using BIGDAVID. **DAVID** is somewhat faster, but BIGDAVID can be used for all cases.

4. At the end of the session there will be two new files recorded to disk in the current directory. These files are called TEMPLATE and TYPETABL. Before running another session make copies of these files in another directory (possibly renaming them), if they are to be saved for future use.

If you want to generate a template for a defining a database in a hierarchical or network database language:

1. Change to the directory \IC\DAVIDH.

2. First, run a session to generate log-on and log-off templates. Start this session by typing DAVIDH0 (ending with the number 0).

3. Run a session to generate the complete template. Start this session by typing DAVIDH or BIGDAVH or VBIGDAVH. DAVIDH is appropriate for entering information about a database language which is not too verbose: i.e. The language does not use a lot of punctuation or long keywords, etc. BIGDAVH is appropriate for more verbose languages and VBIGDAVH for even more verbose languages. If you get an error message saying you ran out of memory during a session, rerun the session using a version capable of handling a more verbose language. DAVIDH is faster than BIGDAVH which in turn is faster than VBIGDAVH, but VBIGDAVH can be used for all cases.

4. At the end of the session there will be two new files recorded to disk in the current directory. These files are called TEMPLATE and TYPETABL. Before running another session make copies of these files in another directory (possibly renaming them), if they are to be saved for future use.

If you want to generate a template for a database query:

1. Change to the directory \IC\DAVIDQ.

2. First, run a session to generate log-on and log-off templates. Start this session by typing DAVIDQ0 (ending with the number 0).

3. Second, run a session to generate the complete template. Start this session by typing DAVIDQ.

4. At the end of the session there will be three new files recorded to disk in the current directory. These files are called TEMPLATE and BOOLTABL and LCTABL. Before running another session make copies of these files in another directory (possibly renaming them), if they are to be saved for future use.

Appendix 3 - TGS Detailed Information

The software for each session is modularized so that it can fit into available memory. If you are using a lot of memory resident software, you might have to disable some of this software to allow the Intelligence/Compiler software to run.

As each module is being loaded from disk, a load bar appears at the bottom of your screen showing the progression of the load.

The modules being loaded are identified on disk with the extension *.ICP. Each module has a specific logical task to perform in the reasoning process. When a module starts to execute, messages usually appear on the screen telling the user what that module is trying to do.

The tables below show the logical tasks performed by each module. It should be noted that not all modules in a directory are invoked during the same session.

MODULES in \IC\DAVID

DAVID0 Generates introductory and exit templates

DAVID1A Elicits a sample statement from the user and parses it

DAVID1C Looks for:
1) database name keywords, table name keywords, and field name keywords
2) various delimiters

DAVID1D Looks for:
1) things we call database delimiters
2) various separators, suffixes, and terminators
3) any extra lines which we can't account for

DAVID1E Looks for:
1) things we call database delimiters
2) any extra lines appearing after the line containing
the database name but before
the line describing the first table
3) something we call the database name suffix

DAVID1F Looks for:
1) various things we call separators, suffixes, and terminators
2) any extra lines appearing after the last table description

DAVID2A Elicits information about field definitions from the user

DAVID2B Constructs the field definition template

DAVID3A Generates a template for database definition

DAVID3B Elicits information about data types from the user

DAVID3C Generates the final TEMPLATE and TYPETABL

MODULES in \IC\DAVIDH

DAVIDH0 Generates introductory and exit templates

DAVIDH1A Elicits a sample statement from the user and parses it

DAVIDH1C Looks for:
1) database (or schema) name keywords, record name keywords, and field name keywords
2) various delimiters

DAVIDH1C1 Looks for database (or schema) name keywords

DAVIDH1C2 Looks for record name keywords and associated delimiters

DAVIDH1C3 Looks for field name keywords and associated delimiters

DAVIDH1D Looks for:
1) things we call database (or schema) delimiters
2) any extra lines appearing after the line containing the database (or schema) name but before the line describing the first record
3) various separators, suffixes, and terminators
4) any extra lines appearing after the last record description

DAVIDH1E Looks for:
1) things we call database (or schema) delimiters
2) any extra lines appearing after the line containing the database (or schema) name but before the line describing the first record
3) something we call the database name suffix

DAVIDH1F Looks for:
1) various things we call separators, suffixes, and terminators
2) any extra lines appearing after the last record description

DAVIDH1X Deciphers any extra lines appearing after the line containing the database (or schema) name but before the line describing the first record

DAVIDH1Z Figures out how the user supplies information on

parent-child links

DAVIDH2A Elicits information about field definitions from the user

DAVIDH2B Constructs the field definition template

DAVIDH3A Generates a template for database definition

DAVIDH3B Elicits information about data types from the user

DAVIDH3C Generates the final TEMPLATE and TYPETABL

MODULES in \IC\DAVIDQ

DAVIDQ0 Generates introductory and exit templates

DAVIDQ1 Elicits query example from user ; Parses the query; Separates example into initial, boolean, and ending lines

DAVIDQ2 Makes a template from the initial lines

DAVIDQ3 Makes a template from the boolean lines

DAVIDQ3B Elicits information from user about Boolean Operators and Linguistic Conventions; Generates the BOOLTABL and LCTABL

DAVIDQ4 Makes a template from the ending lines
Generates the whole query template

Appendix 4 - Table Documentation

To facilitate the creation of templates, the expert system will create the following table which will subsequently be read by the Interface Driver Program.

Type Conversion Table

RESIDENT DATA TYPE	TEMPLATE TYPE #	SEPARATOR (e.g. .,:;)	CORRESPOND DAVID TYPE	MIN LEN	MAX LEN	DEFAULT LENGTH

RESIDENT DATA TYPE is columns 1-19
TEMPLATE TYPE is column 21
SEPARATOR is column 23
CORRESPONDING DATA TYPE is columns 25-34
MIN LEN is columns 36-38
MAX LEN is columns 40-42
DEFAULT LEN is columns 44-50

DAVID Type Conversion Table

Values in the third column of the above table come from the following table of possible **DAVID** data types:

DAVID DATA TYPE	**DESCRIPTION**
INT	binary integer
CHAR	character string
NUM	character string containing only numbers
FLOAT	floating point
SCHAR	single character

Length Parameter Table

Values in the second column of the table above come from the following table of possible lengths for the template types:

TYPE	DESCRIPTION
1	No length field
2	One length parameter
3	Two parameters, the first is total digits; the second is the number of digits to the right of the decimal point.
4	Two parameters, the first is the number of digits to the left of the decimal point; the second is the number digits to the right of the decimal point.
5	Two parameters, the first is the total number of digits including one for the decimal point; the second is the number of digits to the right of the decimal point.

In cases 3, 4 and 5 above, there are several choices for the separator between parameters. The possibilities are: , ; . This information should be solicited from the user in the interactive session.

Boolean Operator Table

Boolean Operation	Your Operator	Precedence	
and			
or			
not			
>			
<			
=			
>=			
<=			
!=			
"			Delimiters around literals
(			Left and right parens are
)			used to override precedence

Boolean operation is columns 2-19
Your Operator is columns 22-30
Precedence Levels columns 38-47

Linguistic Convention Table

Question	Remarks	Answer
Spaces around operators required?	Y or N	
Spaces around operators not required?	Y or N	
Schematic qualified field name:		

Question is columns 1-40
Remarks is columns 42-49
Answer is columns 52-80

Appendix 5 - Integration of Interface Driver Software

This section discusses how to integrate the interface drive software into
DAVID and points out routines that may have to be modified later. The sections is divided the same as the above section on "program structure".

david.c

The main program is only a driver. To install it into **DAVID**, the following changes have to be made.

Name of the template file:

We have been using TEST1.TMP as the default template name. It should be changed to whatever the name stored in the GSQL-ROW.

GSQL-ROW:

Pass GSQL-ROW to the main program instead of creating it using create.c

Name of the type conversion file:

Change it to the correct name from TEST1.TBL.

gettemp.c

The naming scheme is DBMS followed by Operation Type. If the template files are called using other conventions, change it accordingly.

tree.c

No change is necessary.

create.c

This module should be removed.

gettable.cb

The name changes are the same as gettemp.c.

buildht.c

If more information is needed to fill the template, the function build_hash_table must be modified accordingly. The pack_hash function is provided to store information in the hash table.

typeconv.c & utility.c

No changes are necessary.

Appendix 6 - IDMS Code for First, Next , etc.

How do we get the first record:

```
/*  READY AREA(area_name). */
SUBSCHEMA_CTRL.DML_SEQUENCE = SEQUENCE;
CALL IDMS (SUBSCHEMA_CTRL,IDBMSCOM (37)
           ,AREA_NAME);      /* a string of 16
characters */

IF (ERROR_STATUS <> '0000') THEN DO;
    STATUS_CODE = ERROR_STATUS;
    GOTO END_STATUS;
END;

/*  BIND RECORD(name). */
 SEQUENCE = SEQUENCE + 1;
 SUBSCHEMA_CTRL.DML_SEQUENCE = SEQUENCE;
 CALL IDMS (SUBSCHEMA_CTRL,IDBMSCOM (48)
            ,RECORDNAME  /* a string of 16
characters*/
            ,IN_OUT_RECORD);   /* the actual record */
IF (ERROR_STATUS <> '0000') THEN DO;
    STATUS_CODE = ERROR_STATUS;
    GOTO END_STATUS;
END;

/* OBTAIN FIRST RECORD(name) AREA(area_name). */
 SEQUENCE = SEQUENCE + 1;
 SUBSCHEMA_CTRL.DML_SEQUENCE = SEQUENCE;
 CALL IDMS (SUBSCHEMA_CTRL,IDBMSCOM (19)
            ,RECORDNAME   /* a string of 16
character*/
            ,AREANAME          /* a string of 16
characters*/
            ,IDBMSCOM (43));
IF (ERROR_STATUS = '0307') THEN STATUS_CODE =
ERROR_STATUS;
```

How do we get the next record:

```
/* OBTAIN NEXT RECORD(name) AREA(area_name). */
 SUBSCHEMA_CTRL.DML_SEQUENCE = SEQUENCE;
 CALL IDMS (SUBSCHEMA_CTRL,IDBMSCOM (11)
            ,RECORDNAME       /* a string of 16
```

```
characters */
                    ,AREANAME          /* a string of 16
characters */
                    ,IDBMSCOM (43));
        IF (ERROR_STATUS = '0307') THEN STATUS_CODE =
ERROR_STATUS;

      How do we close IDMS:          ----------------------

        /* FINISH. */
         SUBSCHEMA_CTRL.DML_SEQUENCE = SEQUENCE;
         CALL IDMS (SUBSCHEMA_CTRL,IDBMSCOM (02));
        IF (ERROR_STATUS <> '0000') THEN STATUS_CODE =
ERROR_STATUS;
```

Appendix 7 - SUBSTITUTING COMMAND NAMES

For commands beginning with "@"

NAME	MEANING
@schema_NAME	name of database schema
@file_NAME	name of physical file upon which data-
@area_NAME	base sits name of physical area for database
@dbname	database name
@begintables	start of repeating table info
@table_NAME	name of table in database
@parent_NAME	name of parent table
@table_length	length of table (usually in bytes)
@table_position	starting position of table in database
@table_comment	comment about table
@endtables	end of repeating table info
@beginfields	start of repeating field info
@rfnam	name of field in database (formerly
@rftyp	called column)
@rflen	datatype of field in database
@RFSIZ	length of field in database
@sfield_NAME	size of field (length =)
@sfield_type	name of source field (used only in
@sfield_length	

```
@sfield_position  queries)
@sfield_comment     datatype of source field (used only in
@@sfield_position queries)
@sfield_comment     length of source field (used only in
@RFPOS            queries)
@RFCOM              starting position of a field
@endfields          comment about field
@beginsets          starting position of a field
@set_NAME           comment about field
@set_owner_NAME     starting position of a field
@set_member_NAME    comment about field
@endsets          end of repeating field info
                  start of set info
                  name of set of records
                  name of owner of set
                  name of member of set
                  end of set info
```

COMMENTS: Some of the naming conventions have changed. They include:

-- the word column was changed to field. Column is too confusing.

-- rather than rfield_NAME, sfield_NAME, I used simply field_NAME for the result fields; this way, the definition generators don't have to be concerned with "source" and "result", which have no meaning in definitions.

In this scheme, then, the result fields will be simply "field_NAME".

```
field_NAME
field_sname
field_rname
```

-- items were added, such as "sets" (used in network databases) and
"schemas" and "areas" (used in many big dbmss).

1. Report No. 8	2. Government Accession No.	3. Recipients Cat. No.
4. Title and Subtitle A Generalized Strategy for Building Resident Database Interfaces		5. Report Date May 12, 1990
		6. Performing Org. Code n/a
7. Authors Ken Wanderman & Marsha Moroh		8. Performing Org Report Number n/a
9. Performing Organization Name and Address Ken Wanderman & Associates, Inc 160 Bement Avenue Staten Island, NY 10310		10. Work Unit No. 11. Contract No. NAS5 30304
12. Sponsoring Agency Name and Address Dr. Barry E. Jacobs Code 634, NASA Goddard Space Flight Center Greenbelt, MD 20771		13. Type of Report/ Period Covered Final/ May,1988 - May,1990 14. Sponsoring Agency Code 634
15. Supplementary Notes n/a		
16. Abstract We have developed a statratgy for building resdient interfaces to host heterogeous distributed database management systems and have used the strategy to construct several interfaces. We have developed a set of guide lines for users to construct their own interfaces.		
17. Key Words interface, heterogeneous distributed DBMS	18. Distribution Statement	
19. Security Classif. U	20. Security Classif (page) U	21. No. of Pages 133