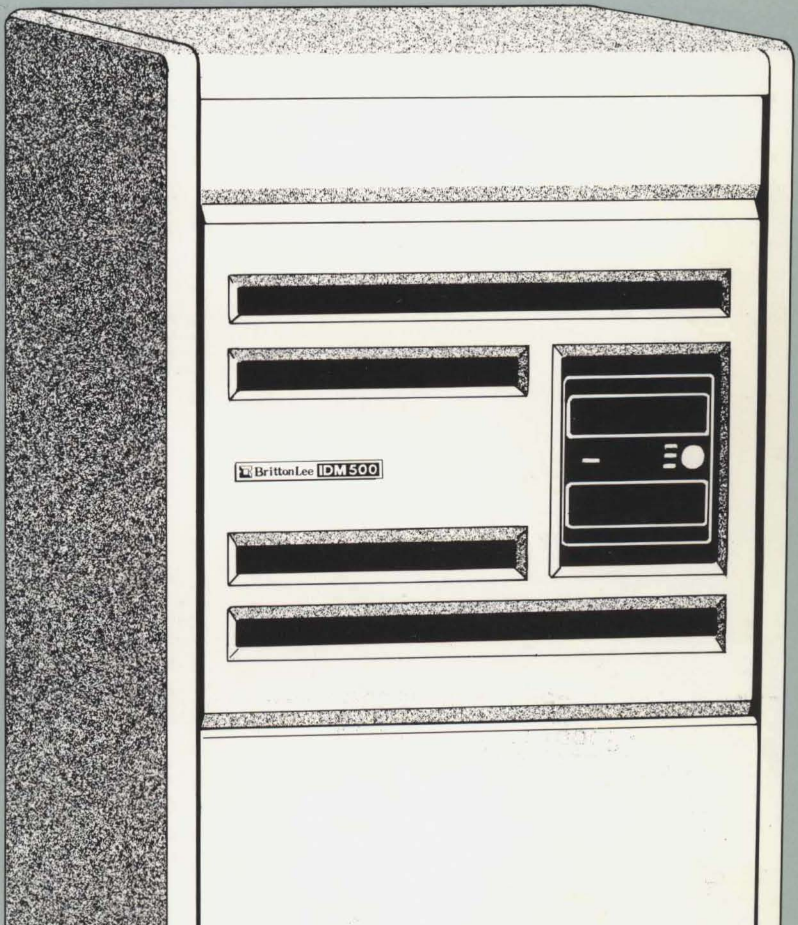
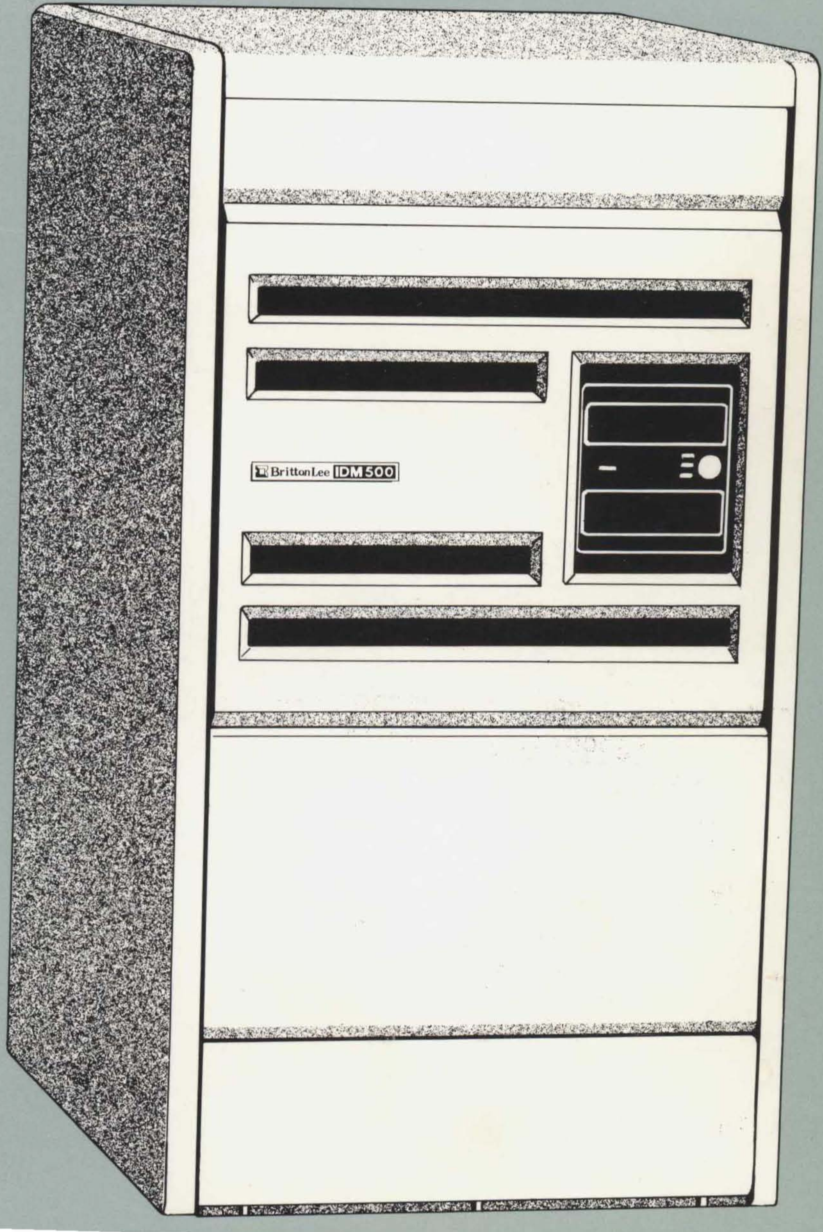
Britton Lee, Inc.

INTELLIGENT DATABASE MACHINE

IDM SOFTWARE REFERENCE MANUAL VERSION 1.7




Britton Lee IDM 500

BRITTON LEE INC.

IDM SOFTWARE REFERENCE MANUAL VERSION 1.7

NOVEMBER 1984
Part Number 202-0500-017

This manual corresponds to
release 31 of the IDM System Software

First Printing, November 1984

This document supercedes all previous documents. This, the first edition is intended for use with software release number 3.0, and future software releases, until further notice.

FOREWORD

General

This manual contains version 1.4 pages and additional supplement version pages. Please note that the supplement version pages have an update note on the bottom of each page, for your convenience.

Because the version 1.7 supplement pages are typeset, they no longer match the word spacing of the older text. This causes many pages to appear to be shorter. The last sentence of a page may end mid–page. The text per page is identical to the page it replaced except where noted by the margin character. The sentences end with the same word as the older page. This uneven ending has been filled with a bar "————".

TABLE OF CONTENTS

Appendices are now to be found in the manual titled, “IDM SYSTEM – APPENDICES A, B, C, D, E”.

Britton-Lee Inc.

IDM Software Reference Manual

Document Summary

The Intelligent Database Machine (IDM) is a functionally complete hardware/software database management machine. It is designed to be either a stand-alone system, supporting several programmable terminals, or a back-end system for several general purpose computers. In either case, the front-end system must provide the interface between the end-user and the IDM.

The front-end system must take the database command from the end-user, translate it to IDM-internal form, then send it to the IDM. The purpose of this document is to present a detailed explanation of the functions provided by the IDM and the alternate user interfaces that may be provided by the front-end system. It is presumed that the reader has previously read the "IDM 500 Product Description" and is therefore familiar with the architecture of the IDM.

There are eight sections in this document. The first three comprise the description of the IDM data management system. The next three contain information regarding the front-end programming. The seventh section is a detailed description of each IDM command, and the last includes the appendices.

Section 1 is an introduction to the basic concepts of the IDM database management system. The database command language IDL (Intelligent Database Language) is introduced in Section 2. It is necessary to understand IDL in order to understand the full IDM power and to follow the examples presented. However, the front-end system is not constrained to use IDL as its database command language.

Section 3 is a general description of how to use the IDM for data management. Section 4 is an overview of the programming requirements for any system that front-ends an IDM. It includes extensive examples. The front-end programming can be broken into two parts: the first part is the application program that takes the user database command and translates it to a form understood by the IDM; the second part includes the operating system functions necessary to send the command to the IDM and receive the results. The first part is discussed in Section 5, "End-User Interfaces". The second is in the following section, "Communicating with the IDM", which also includes the channel protocols.

Section 7 is the complete list of the IDM commands. It includes the IDL syntax, the IDM-internal form and examples of each of the commands.

Note that a vertical bar in the right margin, as seen to the right of this paragraph, represents a change from the previous version of the manual.

1. Introduction to IDM Data Management

The IDM is a relational database management system. This section introduces the basic concepts of that system. However, before proceeding with that description, there are two necessary explanations.

First, this section simply describes what is within the IDM; the OEM determines the interface to it. There are two types of users of the IDM: those who develop the applications (whom we call the OEM) and those who see only the OEM-provided interface (whom we call the end-user, or user). Some features that exist in the IDM may not be provided by some OEMs to their users if the OEMs decide that such functions are not necessary for their particular applications. Conversely, the OEM may provide features not found in the IDM, but which will appear to the user to be part of the data management system.

Second, we use IDL for our examples. IDM commands are given in IDL (Intelligent Database Language) instead of IDM-internal form because IDL is clearer. It should be noted that the OEM may choose to implement any database command language, or none; the actual IDM-internal form for the commands is given in Section 7.

There are two parts to this section. Section 1.1 discusses the user concept of data as implemented in the IDM's data management system. Section 1.2 introduces the functions necessary for a complete data management system.

1.1. User Concept of the Data

The IDM is used to maintain a collection of data. Data is stored in "relations". Relations are kept in "databases".

1.1.1. Databases

Data stored in the IDM is stored in a "database". In common parlance, a database is both a logical grouping of data, and a physical place to store data. In the IDM, a database is a physical place to store data. Thus you may hear people speak of the "employees database" and the "stockroom database", and find that they are both kept in an IDM physical database named "corporate". This means that the collection of data which people think of as describing the company's employees, and the separate collection of data which describes the company's inventory, are both kept in the same IDM database.

In the IDM a database is an independent collection of objects. These objects contain information about one or more applications. IDM commands may not span more than one database, thus all information about one application must be in a single database; although a single IDM database may hold information about many applications.

When the IDM is first installed it contains only one database, the system database. That database is used to keep track of all the other databases. To make other databases, the user sends the "create database" command to the IDM. This new database can be used to maintain data about any application the user wishes. This new user database is, initially, almost empty; only a few objects exist in the new database, just enough to let the user begin.

1.1.2. Relations

An IDM database holds many objects. The most important kind of objects are called "relations". Relations are the objects that actually hold the data.

The data in a relational database management system is organized into tables, called relations. You can think of them as sets of records. The rows of the tables are called "tuples", the columns are called "attributes". Tuples are like records, and attributes are like data fields. The following example shows two relations:

Relation: products

Attributes: name,
part (part name),
quan (amount of this part in the product)

name	part	quan
TV	transistor	15
radio	antenna	1
radio	cabinet	1
radio	transistor	12
radio	speaker	1
stereo	transistor	10

```
Relation: parts
Attributes: name (part name),
            cost,
            min_amt(re-order when "curr_amt"
                    falls below this amount),
            curr_amt (the current inventory total)
```

name	cost	min_amt	curr_amt
antenna	323	25	50
cabinet	2140	40	32
picture tube	8000	25	40
speaker	5225	25	20
transistor	50	225	325

The above relations contain information about the components of several products. The components are in the "parts" relation; the products in the "products" relation.

Relations are related to each other through their data values. For instance, in order to determine the unit costs of the parts in a radio the query (in IDL) is:

```
(1)   range of p is parts
(2)   range of pr is products
(3)   retrieve (p.cost, p.name) where p.name = pr.part
        and pr.name = "radio"
```

Statements (1) and (2) are "range" statements that bind the variables "p" and "pr" to the relation names "parts" and "products". "range" statements are explained in detail in Section 2. Statement (3) is the IDL command to display the cost and name attributes of all parts that belong to the component "radio".

results:

name	cost
antenna	323
cabinet	2140
speaker	5225
transistor	50

The parts used in making a radio are referred to explicitly, by value, rather than by storage position or schema definition. Since tuples are never referred to by physical or schema position, but only by value, the relation's physical storage structure can change without disturbing application programs.

The values that associate tuples of one relation with those of another we will call the "linking attributes". The linking attributes in the above case are the attribute "part" in the products relation and the attribute "name" in the parts relation. The linking attributes are determined, used, and supported by the user. There is no need to declare the presence of a linking attribute to the IDM; in fact, often what is a linking attribute for one query is simply a data value for another.

1.1.3. Normalized Relations

Relations maintained by the IDM must be in first normal form. This means that no attributes are composite; i.e., each attribute of each relation is atomic, consisting of a single value chosen from the domain over which the attribute is defined. "Repeating Groups" are not allowed.

Beyond first normal form, there are increasing degrees of normalization. Second, third, and fourth normal forms are design concepts of crucial importance to the database administrator, but the IDM requires only first normal form.

1.1.4. Views

A view is a virtual relation which is defined in terms of real relations or other views. The definition of a view looks exactly like a retrieve statement. When a range variable in a query is a view, then the view definition is substituted for the view before running the query.

View processing is divided into two parts: view definition and view substitution. When the view is defined, certain modifications are done to the query tree and then the tree is stored in an identical manner to stored commands. When the view is subsequently referenced, the view definition is fetched, and the view definition is substituted wherever required.

1.1.5. Data Values: Duplicates and Unique Keys

Many data management systems require that one of the attributes of the tuples contain a "unique key". A "unique key" is a value associated with each tuple such that no other tuple in the relation can have the same key value. The IDM does not require unique keys. CORRECT OPERATION DOES REQUIRE THAT EACH TUPLE BE UNIQUE; that is, that no two tuples can contain the same values for all attributes. Duplicates are removed whenever a relation is sorted and when tuples are added to a sorted relation (a relation is sorted when a clustered index is created on it). Duplicates are deleted and a warning message is returned (this is done by setting a status bit in the "Done packet").

Often an application is such that a unique key exists for the relation and, furthermore, the user wants the uniqueness to be enforced. For example, part number is usually a unique key; no two parts can have the same part number. Normally the user would want attempts to assign the same number to two parts to be flagged as errors, and not allowed by the IDM. This is provided for in the IDM by the create unique index command. The syntax and meaning of this command are explained in Section 2. This command is further explained in Sections 3 and 7.

After a unique index on "number" is created for the "parts" relation, attempts to add tuples with the same part number as one that exists in the relation will not be allowed. Also, if the part number for any part is changed to be the same as an existing part number, the change is not allowed and flagged as an error.

1.1.6. Ordered Data

Tuples in the IDM relations can be arranged in any order and the order changed at any time. When data is retrieved from the IDM, the only way to assure that the tuples are returned in a specified order is to state the order as a part of the command:

```
retrieve (p.name, p.number)
        order by number:ascending
```

The "order by" clause specifies the order in which the tuples are returned.

1.2. Data Management Functions in the IDM

A functionally complete database management system must provide for protection, crash recovery, data consistency, and fast access paths to the data. In this section we describe the philosophy behind the IDM implementation of these features. The features are further discussed in Section 3, with the description of how to use them.

1.2.1. Security

The person who creates the database is the database administrator (DBA) for the database and has the right to grant access privileges. The protection types are read, write, read tape, write tape, execute, create, create index, and the destroys corresponding to create, and create index.

Users are identified through the host-id and host-user-id provided by the front-end system(s). The "host_users" relation is a system relation which maps the host-id and host-user-id number pair into a single user-id for that database. The "users" relation is a system relation which maps the user-id into a user name and group. The DBA assigns access rights to a database by putting a user in the "host_users" relation and the "users" relation. This provides a cross-reference between the host-id/host-user-id pair and user's name and group. Then the DBA assigns access rights by user name or group through the use of permit and/or deny commands.

Users may also create and protect their own private relations within the database. More details can be found in section 3.5, "Protection".

1.2.2. Consistency: Transaction Management

A "transaction" is a set of one or more IDM commands. The IDM insures that the user, within a transaction, has a consistent database. A "consistent database" means that if two or more users are simultaneously operating on the same data, the result will be as though only one user was operating at a time. This is done through transaction management, where the user declares that a set of IDM commands form a "transaction" that should be treated as a single operation on the database. If the user first commands begin transaction and then reads a tuple of a relation, the tuple will not change value (unless that user changes it) until after the transaction has ended. Any tuple that the user changes will not be accessible by other users until the transaction is complete. If the transaction is aborted for any reason before it is ended (due to a system crash or the user typing abort transaction) the data changed by the transaction is restored to its pre-transaction value.

1.2.3. Transaction Logging

The IDM provides transaction logging for three reasons: transaction management, crash recovery, and audit support. The transaction log includes a record of which values were changed (their previous and new values are recorded), the time the change took place, and the IDM user id for the user who made the change.

The transaction log is kept in the system relation "transact". It is used in transaction management to "backout" those changes made during a transaction if an "abort" is encountered. If the relation was not created with the parameter "logging" specified, logging is to the "batch" system relation. The "batch" relation provides only the transaction management and crash recovery functions of the "transact" relation; the audit trail function is not provided, since the "batch" relation is periodically purged.

If the relation that is changed was created with the parameter "logging" specified, the records are kept for audit and recovery support. The user should periodically write the log to another device (either to the front-end, to tape or to another disk on the IDM) with the dump transaction command. The result of the dump transaction command is that all records of transactions that have ended are written to the dump device, and deleted from the transaction log.

The transaction log dump can take place while the database is being used. The dumped transaction log can then be used to create an audit report with the audit command, and/or used as an incremental dump to restore databases lost in a disk crash.

1.2.4. Crash Recovery

The IDM provides a database dump and a database load facility. The database dump provides a means of dumping the entire contents of a single database to a file on the IDM, to tape or to the front-end system. The database load will take for input the result of a prior database dump and will restore the contents of a database to what it was when the dump was taken.

In the case of disk failure, the user loads the affected databases one at a time with the load database command. Then the transaction logs are used to bring the databases up to date. This is done by the user who issues rollforward commands per database affected. The transaction log carries the information to update the database such that rollforward will work correctly even if a part of the database was on a disk that did not have to be restored.

The transaction log is therefore used exactly like an incremental dump. It records all the changes to relations that were created with the "logging" with-node-option, and can be used, together with the complete dump, to restore the database. How often complete dumps are taken is a function of the application.

1.2.5. Performance

The IDM provides for fast access to data through user-defined indices. An index is a directory that contains data values together with pointers to the physical location of the data. An index is created to facilitate access to data. For example, if we wanted part number A12 in the "parts" relation, one way to find it is to read the entire relation, searching each tuple to determine if it is part A12. That would be an inefficient way of finding part A12, but is exactly what would happen if there is not an index on part number. If there is an index on part number, the index is first searched for part number A12. Then the block that contains the part A12 is directly accessed.

The IDM keeps all indices in the form of a B-tree. A B-tree is a data structure that has the property of very fast access. B-trees have the additional advantage that once they are created they do not require periodic maintenance by the user.

There are two types of indices: clustered and non-clustered. Creating a clustered index on an attribute (or group of attributes) of a relation causes the relation to be sorted on that attribute (or attributes) and <u>physically stored</u> in the disk blocks in sort order. The attributes are specified in the <u>create clustered index</u> command. The DBA should create a clustered index for almost every relation in the database, since storing the tuples in sort order means that they can be quickly accessed when that order is needed.

A relation can have only one clustered index, since the data can be physically stored only one way. The indices on other attributes (those for which the data is not in order) are the nonclustered indices.

The design philosophy of the IDM is that the DBA is the only one with the knowledge of which attributes to create indices on; so the DBA is responsible for commanding the IDM to create indices.

2. Introduction to IDL

The Intelligent Database Language (IDL) is the general purpose database command language designed by Britton-Lee to best use the functions of the IDM.

This section on IDL is included for three reasons: first, to provide an example of a command language that will fully use the functionality of the IDM; second, to help the reader understand the examples in the following text; and third, as a tutorial for those OEMs who will implement IDL as their general-purpose command language.

Most front-end systems will take a user-generated database command and translate it to an IDM-internal form. IDL easily translates to the IDM-internal form, and we provide help (in Section 5) for writing the programs that perform that translation. However, since the translation task is the OEM's responsibility, the OEM can elect to use any command language suitable to the application. The IDM never sees the original user-generated command. It sees only the IDM-internal form, and has no idea whether it was generated from IDL.

2.1. Beginning Commands

The following commands are given in their simplest form. Section 7 contains the complete definition of each command.

2.1.1. create database

To make a new database, the command is create database.

Let us assume we wish to put data in a new database called "inventory". Then the command is:

```
create database inventory
```

There are several optional parameters for the create database command, which are discussed in Section 7. In the above example, since we specify no parameters, the IDM uses its default values.

In response to the above create database command, the IDM sets up the database called "inventory" for the user.

2.1.2. open

To communicate to the IDM that the user will be working on the "inventory" database, the command is:

```
open inventory
```

In order to do any work on a database, it must first be the object of an open command.

2.1.3. create

The command create sets up a relation. For instance, suppose the inventory database is to contain two relations: "parts" and "products". The commands to set up these relations are:

```
create parts(name = c14, cost = i4, min_amt = i4,
        curr_amt = i4)

create products(name = c14, part = c14, quan = i4)
```

After these commands are executed, the "parts" relation will contain four attributes: "name" (the name of the part, a maximum of 14 characters), "cost" (how much the part costs, a 4-byte integer), "min_amt" (re-order when "curr_amt" falls below this number, a 4-byte integer), and "curr_amt" (the current amount in inventory, a 4-byte integer). The "products" relation contains three attributes: "name" (the 14-character product name), "part" (a 14-character part name), and "quan" (a 4-byte integer that shows how many of the part are contained in the product).

The valid types for the attributes of relations are:

Attribute Types

c	compressed character string (trailing blanks are not stored) max size: 255 characters
uc	uncompressed character string max size: 255 characters
i1	1-byte integer
i2	2-byte integer
i4	4-byte integer
f4	4-byte floating point (only limited support: store, retrieve, and 2's complement compare)
f8	8-byte floating point (only limited support: store, retrieve, and 2's complement compare)
bcd	compressed binary-coded decimal (leading zeroes are not stored) max size: 17 bytes, which is 31 digits
ubcd	uncompressed binary-coded decimal max size: 17 bytes, which is 31 digits
bcdflt	compressed binary-coded decimal (BCD) floating-point (leading and trailing zeroes not stored) max size: 17 bytes, which is 31 digits
ubcdflt	uncompressed binary-coded decimal (BCD) floating-point max size: 17 bytes, which is 31 digits
bin	variable-length binary byte string (trailing zeros not stored) max size: 255 bytes
ubin	fixed-length binary byte string max size: 255 bytes

The attribute type and maximum attribute size are specified in the <u>create</u> statement. For instance, the command

```
    create new (name = c25, salary = bcd8, address = c200)
```

specifies that the relation "new" will have three attributes: "name", which is up to 25 characters long, "salary", which is up to 8 digits (6 bytes) long, and "address", which is up to 200 characters long. A tuple that is stored must be no longer than 2000 bytes. A relation may be created with 250 attributes maximum; however, only 140 to 180 attributes may be accessed at one time, depending on the complexity of the command. This limit will increase in future releases.

2.1.4. append

One way to get data into the relations is through the _append_ command, which is in the example below. The other way is through the _copy_ command, which is discussed in Section 7.

```
    append to parts(name = "transistor", min_amt = 225,
                    cost = 50, curr_amt = 325)
    append to products(name = "stereo",
                    part = "transistor", quan = 10)
```

The above _append_ commands cause the IDM to store the data in the relations specified. In an _append_ command, the names of the attributes must be specified.

2.1.5. retrieve

Let us assume that we have entered data in the relations "parts" and "products" defined above, and that the relations are as shown below.

```
    Relation: products
    Attributes: name, part (part name),
            quan (amount of this part in the product)
```

name	part	quan
TV	transistor	15
TV	speaker	2
TV	cabinet	1
TV	antenna	1
TV	picture tube	1
radio	antenna	1
radio	cabinet	1
radio	transistor	12
radio	speaker	1
stereo	transistor	10
stereo	cabinet	1
stereo	speaker	4
tape recorder	tape reel	2
tape recorder	transistor	20

```
Relation: parts
Attributes: name (part name),
            cost,
            min_amt(re-order when "curr_amt" falls below
                 this amount),
            curr_amt (the current inventory total)
```

name	cost	min_amt	curr_amt
antenna	323	25	50
cabinet	2140	40	32
picture tube	8000	25	40
speaker	5225	25	20
tape reel	327	30	22
transistor	50	225	325

In order to display data from these relations, we must first provide the range statements. The range statement binds a variable name to a relation name:

```
range of p is parts
range of pr is products
```

Suppose we want to know the quantity and type of each part that goes into making TV´s. The query is:

```
retrieve (pr.part, pr.quan) where pr.name = "TV"
```

The names of the attributes to be "retrieved" (displayed to the user) are "part" and "quan". The list of the attributes to be retrieved is called the "target list". The "qualification" is the specification of which tuples to get the data from; the qualification in this case is "where pr.name = "TV"". The qualification is also called the "where clause".

The above query was a one-variable (single relation) query. The results are:

part	quan
transistor	15
speaker	2
cabinet	1
antenna	1
picture tube	1

Now suppose we want to know the cost of the components that are used in making TVs. This requires that information from two relations be combined: the part name acts as the linking attribute. The query is:

```
    retrieve (p.name, part_cost = p.cost * pr.quan)
            where p.name = pr.part and pr.name = "TV"
```

result:

```
    |name          |part_cost   |
    |----------------------------|
    |antenna       |         323|
    |cabinet       |        2140|
    |picture tube  |        8000|
    |speaker       |       10450|
    |transistor    |         750|
    |----------------------------|
```

In the above example, we gave the expression "p.cost * pr.quan" the name "part_cost". When an expression appears in the target list, IDL requires that it be named so the front-end program can display the name when the value is sent by the IDM. Expressions can appear in the target list and in the qualification.

2.1.6. replace

The command to change the value of a tuple is the replace command. The tuples in which data is to be replaced, and the new values, must be specified. Suppose we wanted to increase the number of transistors used in making a TV to 20. Then the command is:

```
    replace pr (quan = 20) where pr.name = "TV"
            and pr.part = "transistor"
```

Multiple attributes can be specified in the replace command:

```
    replace p (cost = p.cost + 10,
               curr_amt = p.curr_amt + 20)
            where p.name = "transistor"
```

The above command increases the cost by 10 and the current amount by 20 for the "transistor" part in the "parts" relation.

The qualification may contain an arbitrary number of expressions. If we received a shipment of parts for TVs, where there were 10 of each part, the command to update the attribute "curr_amt" would be:

```
    replace p (curr_amt = p.curr_amt + 10)
            where p.name = pr.part and pr.name = "TV"
```

A replace command can only change tuples in one relation. To change tuples in several relations, you must send (at least) one replace command for each relation.

2.1.7. delete

To remove zero or more tuples from a relation, the delete command is used. The command to remove all the parts for radios is:

```
delete p where p.name = pr.part and pr.name = "radio"
```

All parts used in the construction of radios have now been removed. Later you will see how to remove ONLY the parts used to construct radios.

Of course, if we remove all the parts for radios, we should remove the radio tuple itself:

```
delete pr where pr.name = "radio"
```

2.1.8. retrieve into

The command to create a new relation from one or more old ones is the retrieve into command. Suppose we want to order parts where the inventory amount is below the minimum amount and the amount that we order is always three times the difference between the minimum stock amount and the stock on hand. Then we can create a new relation, called "re_order":

```
retrieve into re_order (p.name,
                   amt = (3 * (p.min_amt - p.curr_amt)))
        where p.curr_amt < p.min_amt

range of r is re_order
retrieve (r.name, r.amt)
```

This relation contains the following parts, and amount to order:

```
|name          |amt           |
|-----------------------------|
|cabinet       |            24|
|speaker       |            15|
|tape reel     |            24|
|-----------------------------|
```

If these parts are ordered and all arrive, the amount in the "parts" relation can be updated:

```
replace p (curr_amt = p.curr_amt + r.amt)
        where p.name = r.name
```

To see the new value for the number of cabinets on hand:

```
retrieve (p.name, p.cost, p.min_amt, p.curr_amt)
        where p.name = "cabinet"
```

Results:

name	cost	min_amt	curr_amt
cabinet	2140	40	56

2.1.9. create view

The command to create a virtual relation from one or more relations is the create view command. A view looks like a relation, but it does not have any data. When we did the retrieve into command above, the IDM actually copied the data from the old relations into a new relation. A view just creates a different way of looking at the old data.

Suppose we want to create a view like the relation "re_order". Then we can type:

```
    create view need_to_order (p.name,
                        amt = (3 * (p.min_amt - p.curr_amt)))
            where p.curr_amt < p.min_amt

    range of n is need_to_order
    retrieve (n.name, n.amt)
```

This view appears to contain the following parts, and amount to order:

name	amt
cabinet	24
speaker	15
tape reel	24

The data is not really stored in "need_to_order", it is only stored in "parts"; when "parts" is changed, the values and tuples shown in the view "need_to_order" will also change.

2.1.10. destroy

To completely eliminate an object, the command is destroy <object list>. If we wish to destroy the "re_order" relation created above, the command is:

```
    destroy re_order
```

If we wish to destroy both the "re_order" relation and the "need_to_order" view, the command is:

```
    destroy re_order, need_to_order
```

The only ones who can destroy a relation are the user who created it and the DBA. If there are views or stored commands based on some relations or views, then the base relations or views cannot be destroyed without destroying the dependent objects first.

2.1.11. destroy database

To eliminate an entire database, the command is destroy database.

```
    destroy database inventory
```

The effect of the above command is to completely eliminate the database "inventory" from the IDM, freeing its disk space for other databases that may be created.

2.1.12. Summary

The above examples show the basic syntax of IDL: there is a keyword (e.g. retrieve), a target list (p.name) and a qualification (where p.cost < 20). The keywords are the IDL commands: the complete list is given in Section 7. The target list and qualification generally can contain arbitrary expressions involving multiple relations.

2.2. More Powerful IDL

2.2.1. Range Statements

A range statement is used to associate a variable name with a relation name. Range statements are necessary in order to compare tuples in the same relation to each other. Suppose we want to know what parts cost more than a tape reel. The query is:

```
    range of p is parts
    range of ps is parts
    retrieve (p.name, p.cost) where p.cost > ps.cost
            and ps.name = "tape reel"
```

It was necessary to use two different relation variable names in the above query because we were comparing tuples in a relation to each other.

The delete and replace commands each use a single range variable as the target list; that is to make it clear which tuple is to be deleted or changed. In the case of the append, the actual relation name is used since all that needs to be specified is the relation, not a particular tuple.

2.2.2. The use of aggregates

Aggregates are powerful tools of IDL (and of the IDM). A scalar aggregate is an arithmetic expression that operates over one or more relations and returns a single value. Aggregate functions return a set of values. The aggregation operators supported by the IDM are:

```
    min
    max
    count
    sum
    avg
    any
    once
```

Aggregate "any" returns 0 if no tuples qualify; otherwise "any" returns 1. Count, sum, once and avg (average) can have the optional modifier "unique" which specifies that only the non-duplicated values of the expression are included in the aggregate.

Aggregate "once" is used when one and only one value should qualify. It will generate an error message if no values exist or if more than one value exists. The one value is returned. "Once" can only be used as a scalar aggregate.

A). Scalar Aggregates

The following command counts the unique (non-duplicated) values of "part" in the "products" relation.

```
    range of p is parts
    range of pr is products
    retrieve (num = count unique(pr.part))
```

The result of the aggregate has to be given a name (in this case "num") so the returned result can be identified. The IDM will count all the unique part names in the "products" relation, then return the count.

In order to find the total of the unit costs of all the parts in the "parts" relation, the command is:

```
    retrieve (tot_cost = sum(p.cost))
```

The results:

```
    |tot_cost     |
    |-------------|
    |        16065|
    |-------------|
```

Suppose we wanted to know the total amount of stock on hand; that is just the sum of the "curr_amt" attribute of the "parts" relation:

```
    retrieve (tot_amt = sum(p.curr_amt))
```

The results:

```
    |tot_amt      |
    |-------------|
    |          489|
    |-------------|
```

It is more likely, however, that we would want to know the total dollar value of the stock on hand: this is the total of the "curr_amt" and "cost" attributes:

```
    retrieve (tot_amt = sum (p.curr_amt * p.cost))
```

This page has been intentionally left blank.

The results:

```
    |tot_amt      |
    |-------------|
    |       532574|
    |-------------|
```

So there is $5325.74 worth of inventory parts. Now to determine the total inventory amount excluding the "speaker" inventory:

```
retrieve (inv = sum (p.curr_amt * p.cost
                     where p.part != "speaker"))
```

The results:

```
    |inv          |
    |-------------|
    |       428074|
    |-------------|
```

Or $1045.00 of the inventory is due to the stockpiling of speakers.

Note that the qualification (where p.part != "speaker") is written <u>inside</u> the parentheses with the object of the aggregation. The qualification refers to the objects being summed and not to the query as a whole. This is an important distinction which allows considerable flexibility. An aggregate is always a self-contained query embedded inside another query. For example:

```
    retrieve (p.pname, inv = p.cost * p.curr_amt)
        where
           p.cost * p.curr_amt >
           max(p.cost * p.curr_amt where
                    p.curr_amt < p.min_amt)
        and
           p.cost > 500
```

In the above query we want the name and inventory value of those parts which cost more than 500 and whose inventory value is greater than the inventory value of any parts which need to be reordered. Note that the qualification "p.curr_amt < p.min_amt" refers to the tuples used to compute the "max" function and not to the query as a whole.

Aggregates can be multi-variable:

```
    retrieve (tot = sum (p.cost where p.name = pr.part
             and pr.name = "TV"))
```

The above command gives the total of the unit costs of the parts that make up a TV. Again note that the qualification for the aggregate goes inside the aggregate.

B) Aggregate Functions

Aggregate functions return a set of values. The difference in syntax between an aggregate function and a simple aggregate is the by clause. The by clause is the "group by" operator.

The command to display the inventory cost for each of the parts in the inventory is:

```
retrieve (p.name,
          tot_cost = sum (p.cost * p.curr_amt by p.name))
```

The by clause in this case is "by p.name". This specifies that the sum is to be grouped by the name of the part, then the sums printed out.

The results:

```
    |name           |tot_cost    |
    |----------------------------|
    |antenna        |       16150|
    |cabinet        |       68480|
    |picture tube   |      320000|
    |speaker        |      104500|
    |tape reel      |        7194|
    |transistor     |       16250|
    |----------------------------|
```

Now suppose we do not want to see all the inventory costs, but only of those items where there are more than 30 of them on hand. Then the query is:

```
retrieve (p.name,
          tot_cost = sum (p.cost * p.curr_amt by p.name))
          where p.curr_amt > 30
```

The results:

```
    |name           |tot_cost    |
    |----------------------------|
    |antenna        |       16150|
    |cabinet        |       68480|
    |picture tube   |      320000|
    |transistor     |       16250|
    |----------------------------|
```

Note that the qualification "where p.curr_amt > 30" does not appear in the parentheses with the aggregate function, since it is not needed to evaluate the function; instead, it is needed to specify which amounts to print out, and therefore is a general "where clause".

In the above query it should also be noted that the "by-clause" is global to the whole query. The "p.name" in the target list is the same as the "p.name" in the by-clause and is the same tuple referred to in the qualification list. This is a fundamental difference between scalar aggregates and aggregate functions.

Multi-variable aggregate functions provide powerful facilities; the following example illustrates a multi-variable aggregate function. The purpose of the query is to find the sum of the costs of the parts that make up each of the products.

```
    retrieve (pr.name, tot_cost = sum (p.cost * pr.quan
            by pr.name where pr.part = p.name))
```

The result:

```
    |name            |tot_cost     |
    |-----------------------------  |
    |TV              |        21663|
    |radio           |         8288|
    |stereo          |        23540|
    |tape recorder   |         1654|
    |-----------------------------  |
```

The clause "by pr.name" signifies that the aggregate is to sum the cost * quan into groups according to pr.name. The clause "where pr.part = p.name" identifies the correct cost and quantity to multiply.

Aggregates can be used in both the target and qualification lists of the query. Several aggregations can be performed in the same query. The following query asks for the name and total cost of all the products displayed where the total cost of the product is greater than the average of the total costs of all the products.

```
    retrieve (pr.name, tot_cost = sum (p.cost * pr.quan
            by pr.name where pr.part = p.name))
        where sum (p.cost * pr.quan
            by pr.name where pr.part = p.name)
         > avg (sum (p.cost * pr.quan
            by pr.name where pr.part = p.name))
```

Note that the aggregate for the total cost of the product must be repeated wherever it is referred to in the query. Note also that "tot_cost" is used only as a column heading for output. "Tot_cost" cannot be used as one would a programming variable.

Aggregates can be nested. In the example above, the average sum of costs was specified by nesting the "sum" aggregate inside the "average" aggregate.

2.2.3. order by

The user specifies the order of the data with the order by clause:

```
    retrieve (p.name, p.cost) order by cost:ascending
```

The above command lists the parts names and costs, and produces the output in ascending order:

Results:

```
|name          |cost          |
|-----------------------------|
|transistor    |            50|
|antenna       |           323|
|tape reel     |           327|
|cabinet       |          2140|
|speaker       |          5225|
|picture tube  |          8000|
|-----------------------------|
```

The only way to assure that data will appear in a given order is to explicitly state that order in an "order by" clause. If an order by clause is not present, the IDM will return tuples in the order the IDM found most efficient for processing.

2.2.4. data conversions

Occasionally data will need to be converted to a different type for comparison, arithmetic, or output. In that case, one of the IDM's conversion functions must be used. The functions are:

Conversion Functions

Function	Converts To:
int1	1-byte integer
int2	2-byte integer
int4	4-byte integer
bcd	compressed BCD integer
fixed bcd	uncompressed BCD integer
bcdflt	compressed BCD floating-point
fixed bcdflt	uncompressed BCD floating-point
bcdfixed	uncompressed BCD fixed-point
string	compressed character string
fixed string	uncompressed character string
binary	binary string

Attributes of any of the above types can be converted to each other. Length parameters are used to specify the length of the result when converting to character, binary string, or BCD formats. An example is:

```
retrieve (new_quan = string(10, pr.quan))
```

The above command converts the integer value "quan" to ASCII, then sends it to the front-end.

```
replace p (curr_amt =
        p.curr_amt - (avg(int2(p.curr_amt))/2)
```

The above command shows the conversion of "curr_amt" to a two-byte integer, then the average computed, and finally each of the "curr_amt" values for parts replaced by the current value less half of the average.

Arithmetic expressions which contain a mixture of int1, int2, and int4 terms are evaluated by first converting all terms to the length of the longest term.

2.2.5. String Manipulation

The IDM provides three string manipulation functions: substring, concatenate, and pattern matching. Substring and concatenate can be used with both character and binary objects; pattern matching can only be used with character objects. An example of pattern matching follows.

Suppose we want to print the names and costs of all parts that begin with a "t". Then the command is:

```
retrieve (p.name, p.cost) where p.name = "t*"
```

The results:

```
|name        |cost    |
|------------|--------|
|tape reel   |     327|
|transistor  |      50|
|------------|--------|
```

The "*" is a general pattern matching character. The pattern matching characters are:

* – matches zero or more characters.
? – matches any one character.
[– begin a group of characters any one of which may be matched.
] – end the group of characters.
\ – escape any of the above (used before * \ ? [ ] to indicate that the following character is not a special pattern matching character).

A hyphen can be used within the square brackets to indicate a range of characters: "[a–e]" is the same as "[abcde]". A hyphen always has its special meaning within the brackets, its meaning cannot be escaped.

Pattern matching characters can only be used in the qualification portion of commands.

"concat" takes two character or binary strings, strips any trailing blanks (zeros for binary strings) from the first string (all but one, if the string is all blank), strips all trailing blanks from the second string, and appends the second to the first. For instance,

```
retrieve (newname = concat(pr.name, pr.part) )
```

returns the product names and component part names concatenated together.

```
|newname                         |
|--------------------------------|
|TVtransistor                    |
|TVspeaker                       |
|TVcabinet                       |
|TVantenna                       |
|TVpicture tube                  |
|radioantenna                    |
|radiocabinet                    |
|radiotransistor                 |
|radiospeaker                    |
|stereotransistor                |
|stereocabinet                   |
|stereospeaker                   |
|tape recordertape reel          |
|tape recordertransistor         |
|--------------------------------|
```

Note that it is only trailing blanks that are deleted, not internal ones.

The syntax for "substring" is substring (beg, length, expression). "substring" takes bytes beginning at position "beg" of a character or binary string "expression" and copies them to a result position. The number of characters to copy is denoted by "length". "beg" and "length" are one-byte integer constants. For instance,

```
retrieve (little = substring (2,5,p.name))
```

will display the characters beginning at position 2 in the attribute "name" of the "parts" relation for a length of 5 characters.

3. IDM: Database Management

This section contains a description of the functions of the IDM that can be used to provide a good data management environment. These include stored commands, data dictionary facilities, transaction management, data dump and restore facilities, data protection, and performance tuning support.

3.1. Stored Commands

The IDM has a facility for storing commands and command sequences on the IDM itself in a pre-processed form. When the command is to be run, the host system simply specifies the parameters and the command name or number; then the command is executed.

An example of the use of stored queries is the following: let us assume we wish to define an "add_parts" command for the "inventory" database of Section 2. It is convenient for the users if an "add_parts" command is defined, so the user does not have to type the full "append" command for each part added to the "parts" relation. The _define_ command is:

```
define add_parts
append to parts (name = $1, min_amt = $2,
        cost = $3, curr_amt = $4)
end define
```

After the keyword _define_ comes the name of the stored command. Then the _commands_ making up that stored command are listed. Finally the phrase _end define_ terminates the stored command. Parameters are _indicated,_ in IDL, by a dollar sign.

The IDM command _execute_ causes the IDM to execute the stored command that _is_ specified. The positions of the parameters in the "execute" command correspond to the lexical sort sequence of the parameters in the _define_ command.

```
execute add_parts "knobs", 112, 50, 125
```

The above _execute_ command caused the IDM to add a tuple to the "parts" relation, exactly as if the command had been received as:

```
append to parts (name = "knobs",
                 min_amt = 112,
                 cost = 50,
                 curr_amt = 125)
```

The reasons to use stored commands are:

1) Efficiency

Stored commands are pre-processed when they are stored in the IDM to save time when they are executed. Also, there is less information that has to be passed to the IDM when a stored command is executed because only a short name or number and the parameters, instead of the representation of a full command, are sent to the IDM. Therefore using stored commands saves both IDM execution time and host/IDM transmission time.

2) Controlled access

In the above example, a stored command was used to append data to a relation; similarly, sets of stored commands can be used to control access to relations. If the OEM, through the application program, only allows the users to execute stored commands, then the data added to and retrieved from relations can be tightly controlled. As an example, let us assume we have the "parts" and "products" relations from Section 2. The OEM has decided that, as far as the user is concerned, there are simply products and parts that make up those products, and the application is such that the user never has to be explicitly aware of the structure of the actual data. Now suppose there is a new product added, which possibly involves new parts. Then a stored query can be executed which will update both relations without explicit user intervention.

Assume the command was defined as follows:

```
    define add_product
    append to products (name = $1, part = $2, quan = $3)
    append to parts (name = $2)
    end define
```

Then the user types:

```
    execute add_product "clock", "gear", 3
    execute add_product "clock", "box", 1
    execute add_product "clock", "hand", 2
```

The define command directs the IDM to accept the commands between it and the end define command as a single command sequence. That command sequence is pre-processed, then stored. The execute command directs the IDM to execute the entire sequence.

3) New Commands

Stored commands can be used to define a new command for the user, a command that is a combination of IDM commands. For example:

```
    define got_part
    range of p is parts
    replace p (curr_amt = p.curr_amt + $2)
            where p.name = $1
    range of r is re_order
    replace r (amt = r.amt - $2)
            where r.name = $1
    delete r where r.amt <= 0
    end define
```

The above example shows how to define a command which, when the user types

```
    execute got_part "cabinet", 12
```

will cause the current amount field in the "parts" relation to be increased by 12 for the part "cabinet". It will also cause the amount in the "re_order" relation to be decreased by 12, and will delete from the "re_order" relation any parts that have less than 0 parts remaining to be ordered.

3.2. Data Dictionary

A data dictionary is a service provided by a data management system to its users, to enable the users to interactively define the data schema and to look up that schema once it is defined. The data dictionary functions of the IDM are performed through the use of the IDM system relations and stored commands. There are several system relations per database; a complete list is given in Appendix A. The relations of interest in performing data dictionary functions are:

"relation" relation

This relation contains a list of all relations in the database, including the IDM-assigned relation id (relid), the name of the relation (name), the user id of the owner of the relation (owner), the number of tuples, and other information that the IDM needs to process commands on that relation.

"attribute" relation

The "attribute" relation contains information about each attribute of each relation in the database. The attribute name, type, relid of the relation that the attribute belongs to, and IDM-assigned attribute id (attid) are kept (along with other information needed by the IDM) in the "attribute" relation.

"descriptions" relation

The "descriptions" relation is used to associate one or more descriptions (up to 255 characters long) with a relid/attid pair. If the attid is zero, it is assumed that the description is associated with the relid alone.

The "relation" and "attribute" relations are automatically updated when the user creates a relation. After creating a relation, the user can add a tuple to the "descriptions" relation, using the 255-character "text" attribute to store information about the relation.

When the relation is destroyed, the associated tuples in the "attribute", "relation", and "descriptions" relations are deleted automatically by the IDM.

To facilitate access to the data dictionary, stored commands can be used to access these relations. For example:

```
    range of d is descriptions
    range of a is attribute
    range of r is relation
    define help
    retrieve (r.name, d.text) where r.relid = d.relid
            and d.attid = 0 and r.name = $rel
    end define

    define attinfo
    retrieve (a.name, d.text)
       where r.relid = d.relid  and
             a.relid = r.relid and
             r.name = $rel
    end define

    define rename
    replace r (name = $2)
            where r.name = $1 and r.owner = userid
    end define
```

The user enters the above commands, then (at any time in the future) types:

```
    execute help rel = "parts"
```

The query "help" retrieves the description of the named relation. In the above example, the information about the "parts" relation would be displayed.

In the above example the parameter was specified with a name ("rel") rather than the positional specification used for most stored commands in this document. Either form is acceptable for IDL.

To see the information for the attributes the user would execute "attinfo". It is up to the host system or database administrator to add the information to the "descriptions" relation.

To allow the owner of a relation to change the relation's name, the "rename" command can be used. "userid" is a special IDM function which returns the user identifier of the current user.

Creating and destroying both relations and views are simple commands. It is anticipated that they will be common occurrences, especially when an application is first implemented. The definition of a database is often an evolutionary process. In the following example, a database is created (line 1). Then it is opened (to open a database is to declare that you are going to be accessing the data in it). A relation is created (line 3), and several tuples added (lines 4 - n). (For clarity we used the append command to add data; the bulk load copy facility described in Section 7 could have been used as well.)

```
    (1)     create database inventory
    (2)     open inventory
    (3)     create parts (number = i4, description = c30)

    (4)     append to parts (number = 21,
             description = "-ntenna")

    (5)     append to parts (number = 45,
             description = "cabinet")
                   .
                   .
                   .
    (n)     append to parts (number = 4112,
             description = "speaker")
```

Now suppose part number A1311 must be added, and that it is discovered that several part numbers contain non-numeric information. The decision therefore is made to change the parts number field to be a character attribute. One of the design goals of the IDM is to make such data definition changes as simple as possible; the following commands can be used to change the definition of the "parts" relation:

```
    (1)     range of p is parts
    (2)     create nparts (number = c10, description = c30)
    (3)     append to nparts
                    (number = string(10, p.number),
                     description = p.description)

    (4)     destroy parts

    (5)     execute rename nparts, parts
```

Line (1) is the previously-discussed "range command". Line (2) creates the new relation. Line (3) is an append which causes the IDM to copy data from the old relation "parts" to the new relation "nparts". Since there is no where clause in the append command, every tuple in "parts" is copied into "nparts". The "number" attribute is the character string resulting from converting the integer part number in "parts" to a maximum of 10 characters. The "description" attribute is a direct copy of the "parts" "description" attribute.

The retrieve into command could have been used in place of lines 2 and 3. The DBA chose to use the create command in order to have explicit control over the type of the attribute "number".

When the append command is finished executing, there are two copies of the data in the "parts" relation; that in "parts", and the data with the character part number in "nparts". So the "parts" relation is removed (line (4)) with the destroy command. The "relation" relation keeps track of the names of all the relations in the database; line (5) results in changing the name of "nparts" to "parts" using the previously defined stored command.

In the above five lines we redefined a data structure to meet changing needs of the database. That is the kind of action that is anticipated in the evolutionary process of bringing up an application on an IDM database. The OEM can define the necessary stored commands when installing a system.

3.3. Transaction Management

The default for a transaction in the IDM is a single command. If more than one command is to be included in a transaction, the begin transaction and end transaction commands must explicitly be used.

The reason for transaction management is to maintain database consistency over a set of commands. Database consistency means that if there are two or more transactions that affect the same set of data, the end result will appear as if only one user at a time had been operating on the data. That is, the results will appear as if the transactions were "serialized". Consistency is not a problem in single-user systems, where the only changes to the data are those that the one user makes. However, in multi-user environments consistency must be explicitly enforced.

For example, consider the following problem. User 1 at terminal 1 is ordering parts for which the current inventory level is below the minimum level. He types:

```
    range of p is parts
    retrieve (p.name, p.cost, p.curr_amt)
            where p.curr_amt < p.min_amt
```

Then he reviews the parts that are printed out, determining (based on cost and lead time) which parts to order. He decides to order those parts for which the total cost of the order is less than $100, and to keep track of the rest of the parts in the existing relation, "needs". The command he types is:

```
    append to needs (p.name, p.cost,
            amount = p.cost * (p.min_amt - p.curr_amt))
        where p.curr_amt < p.min_amt
            and p.cost * (p.min_amt - p.curr_amt) >= 10000
```

Now suppose that, before the second command from terminal 1 is issued, at terminal 2, the stockroom clerk issues some parts:

```
    replace p (curr_amt = p.curr_amt - 10)
            where p.name = "cabinet"
```

Then the user at terminal 1 would not see the part "cabinet" in the list of parts that need to be ordered, but it would mysteriously appear in the "needs" relation! In this case we say that user 1 has seen an "inconsistent" database. To keep this from happening, user 1 must surround those commands that demand consistency with the explicit transaction-delimiting commands begin transaction and end transaction.

3.3.1. Begin, End and Abort Transaction

The commands used to delimit a transaction are begin transaction and end transaction:

```
    begin transaction
    retrieve (p.name, p.cost, p.curr_amt)
            where p.curr_amt < p.min_amt
```

If user 1 in the above example types the above command sequence, the "parts" relation is locked: it cannot be changed by anyone except user 1 until the transaction is ended or aborted. When user 1 types:

```
    append to needs (p.name, p.cost,
            amount = p.cost * (p.min_amt - p.curr_amt))
        where p.curr_amt < p.min_amt
            and p.cost * (p.min_amt - p.curr_amt) > 10000

    end transaction
```

the lock on the "parts" relation is released, and updates by other users on the "parts" relation can be processed.

If the transaction involves changing values, an abort transaction command backs out the changes, returning them to their original value. For example, assume that in our example database there is a "customers" relation that contains information about customers, including the customer name, credit limit ("cr_line"), and current amount owed ("bal").

Then, assume that "Stereo City" orders 10 TVs. First, we make certain that there are enough parts to make 10 TVs.

```
    begin transaction
    replace p (curr_amt = p.curr_amt - 10 * pr.quan)
            where p.name = pr.part and pr.name = "TV"

    retrieve (p.name, p.cost, p.min_amt, p.curr_amt)
            where p.curr_amt < 0 and p.name = pr.part
               and pr.name = "TV"
```

Printed results:

```
|name          |cost          |min_amt       |curr_amt      |
|-----------------------------------------------------------|
|-----------------------------------------------------------|
```

If any were less than zero, we would abort the transaction and the relation would be restored to its original condition. Since there were no negative amounts, we can continue:

```
    range of c is customers
    replace c (bal = c.bal + sum (p.cost * pr.quan * 10
            where pr.name = "TV" and p.name = pr.part))
            where c.name = "Stereo City"

    retrieve (c.name, c.bal, c.cr_line)
            where c.name = "Stereo City"
```

Printed results:

```
|name                          |bal           |cr_line       |
|------------------------------------------------------------|
|Stereo City                   |        656630|        450000|
|------------------------------------------------------------|
```

Since the transaction would result in the customer's owing more than is allowed, the transaction is aborted: all relations return to their original condition.

```
    abort transaction

    retrieve (c.name, c.bal, c.cr_line)
            where c.name = "Stereo City"
```

Printed results:

```
|name                          |bal           |cr_line       |
|------------------------------------------------------------|
|Stereo City                   |        440000|        450000|
|------------------------------------------------------------|
```

In the above example, note that the changes to the database are immediately reflected to the user. That is, the user who changes the database during a transaction will see the changes immediately, although no other users can see the changed relation until the transaction is ended.

3.3.2. Performance Implications of Transaction Management

The IDM implements transaction management through the use of "locking" relations or blocks (2K-byte blocks of data). A block, group of blocks, or an entire relation is "read-locked" (no one can change it) while it is read. Read-locks are shared: multiple users can read the same data concurrently. The item (block, group of blocks, or relation) is read-locked until all users who hold read locks on the locked items have completed their current transactions. If the item is not read-locked, a user can change data within that item. In that case, the block, group of blocks, or entire relation, is "write-locked". No user can "write-lock" a relation that is currently "read-locked"; the IDM will delay the "writer" transaction until all "readers" have finished. If new "readers" begin, the "writer" may wait again. The IDM automatically enforces that no new "readers" can block a "writer" who has been waiting a pre-determined amount of time.

A write-lock is not shared. After a user has changed a relation, the "write-locked" blocks of that relation are not freed until the end of the transaction.

This method of transaction management provides what is called degree 3 consistency and insures that a "reader" will always see the same data within a transaction no matter how many times it is read. However, the interactive user who begins a long transaction and may lock out updates for a long period of time. The IDM allows the user to specify in the RANGE statement a lower level of consitency on reads (called degree 2). When degree 2 is specified a process releases the lock on a block when it is finished with the processing of that block, allowing "writers" to update the block prior to the end of the transaction. The performance of the system will be degraded when degree 2 locking is specified since more accesses to the lock table are required. This does not handle the user who begins a transaction, changes several relation blocks, and goes to coffee without ending the transaction, he will effectiv ly lock all other users out of those relation blocks for the duration of his coffee break. The IDM will know if a host system has crashed (see section 6.3.2), but it cannot know if the user has disappeared or is just thinking for a long time.

OEMs who are implementing applications with high amounts of concurrent access should have a "time out" feature in the host that either aborts or ends a transaction after an application-determined wait period.

A second problem is deadlock. Deadlock occurs when two or more users are blocked, waiting for locks that the other

This page has been intentionally left blank.

holds. The frequency of deadlock is application dependent but in practice deadlock is usually a rare event. If a deadlock occurs, the IDM detects it, chooses a transaction to kill and aborts it, backing out all data changes done by the transaction. The IDM does not restart the killed transaction; if appropriate, the host software can resubmit the command.

The host program that contains a "begin transaction" must, upon receiving a "transaction aborted" error signal from the IDM, go back to the "begin transaction" and start over. This is a rare occurrence, but one that the OEM should be aware can occur.

3.4. Audit Support and Crash Recovery

Both audit support and crash recovery are provided through the use of the IDM transaction log. The transaction log is a relation that keeps a record of each change to the logged relations. There is one transaction log per database. A relation is declared to be a logged relation at the time it is created. The transaction log is used for two purposes: to provide reports for auditors, and to provide backup facilities for the important relations. The IDM will automatically bring a database to a consistent state after a crash using the current transaction log, if it is intact. Non-logged relations are also brought to a consistent state. This process is speeded by "checkpoints". A checkpoint forces all disk blocks which have new information to the disks. The IDM will automatically do checkpoints at intervals set by the DBA. The checkpoint is noted in the log and recovery can proceed from the checkpoint rather than the beginning of the log.

3.4.1. Which Relations to Log

Relations that the auditors will be interested in and relations that need a back-up capability should be logged. The cost of maintaining the transaction log is the number of disk writes necessary to keep track of every change to a relation; if the relation is a temporary relation, used only, for example, to create a report, logging is probably not necessary for that relation. Such relations are still protected against system crashes that do not damage the disks.

3.4.2. Dumping the Transaction Log

The transaction log grows as the relations it is monitoring are updated. Therefore it is necessary to periodically move the contents of the log to other media, for example, to a tape.

The following is an example of dumping the transaction log to a file on the IDM:

```
(1)     open backup
(2)     dump transaction inventory transact to transl
```

Command (1) in the above command sequence tells the IDM that we are operating within the database named "backup". Command (2) causes the transaction log for the "inventory" database to be written into the file "transl".

When the transaction log is dumped it is shortened; all records preceeding the first active update are removed. Other processing continues while the transaction dump is done.

3.4.3. Using the Transaction Log for Crash Recovery

In case of a disk malfunction, the transaction log can be used to restore the logged relations to their state at the time of the last readable log. This is done by first restoring the database from the last dump, then applying the transaction log:

```
(1)     open backup
(2)     load database inventory from file2
(3)     rollforward inventory from transl
```

The first command declares that we are working in the database named "backup". Command 2 restores the "inventory" database from the last dump taken, which is on "file2" of the database "backup". The third command causes the transaction log to be applied.

Backup may be done from the host system:

```
(1)     load database inventory
(2)     load transaction scratch inventrans
(3)     open scratch
(4)     rollforward inventory from inventrans
```

The database "scratch" is used to hold the transaction log for the _rollforward_ command. The log must not be loaded to the database which is being restored. The _rollforward_ command can only work from a relation.

3.4.4. Using the Transaction Log for the Auditors

The transaction log is one of the few system relations that is practically unreadable by the casual user. That is because the data is kept in a compressed, binary form for efficiency in writing and applying the log. To display the log for an auditor, the command is:

```
range of t is transact
audit (t.relid, t.user, t.type)
```

Section 7 contains the complete set of parameters for the _audit_ command. The _audit_ command must be run in the database to which it applies. If we dumped the transaction log to another database or to the host, it must be first loaded.

```
(1)  open backup
(2)  load transaction inventory newtrans from transl
(3)  open inventory
(4)  range of n is newtrans
(5)  audit (n.user, n.type, n.value)
              where n.relid = rel_id("parts")
```

Commands (1) and (2) load a copy of the transaction log into the "inventory" database. Then in (3) and (4) the audit report is retrieved. The requirements for backup and auditing are different and so it is necessary to move the log before retrieving the audit.

3.5. User Authentication and Protection

Authentication is the process of securely recognizing the identity of a user of the IDM system. Once a user's identity has been authenticated, the IDM protection system takes over. The protection system determines what database, relations, views and stored commands a user may use.

3.5.1. User Authentication for the IDM

Users interact with a host computer which, in turn, interacts with the IDM database management system. In some environments, the host computer will authenticate the user (typically by asking the user for a login name and password). In other environments, it is necessary for the IDM to authenticate the user. The designer of a database application has the choice of

1. Host computer user authentication
2. IDM user authentication
3. Both Host and IDM user authentication

This choice is based on whether a host computer is "trustworthy" or "untrustworthy". A trustworthy host environment is one in which the host operating system authenticates the user and prevents the user from directly telling the IDM the user's identification. Examples of trustworthy systems are mainframes and minicomputers running multi-user operating systems. An untrustworthy host environment is one in which the user can directly communicate his identification to the IDM. Examples of untrustworthy systems are personal computers and computers connected over some networks. On untrustworthy systems, the user must supply the IDM a password in order to use the IDM.

Users are identified either by name or by number. Host computers are identified by number. These identifications are called:

Hid	Host identification
Huid	User identification number on host computer
Huname	User identification name

on host computer

3.5.1.1. Types of Hosts Using Authentication System

The method of user authentication is determined by the host type and the "login" relation on the IDM. There are four basic host types:

1. Trustworthy host with user numbers
   (examples:VAX/VMS, VAX/UNIX)
2. Trustworthy host with user names
   (example:VM/CMS)
3. Untrustworthy host with user numbers
   (unlikely to be used by anyone)
4. Untrustworthy host with user names
   (example:IBM PC, Ethernet based hosts
   with standard driver)

The IDM protection system requires knowing the hid and huid for every user. Accounting of user dbp usage in the "account" relation of the "system" database also uses the hid and huid to identify every user. Host types 1 and 3 provide the hid and huid directly. For other host types, the huname must be translated to an huid. The "login" relation in the IDM's "system" database provides this translation. The "login" relation has the following attributes

```
login(type, hid, huid, huname, password, class)
```

This relation is used to

1. translate hid, huname pair into huid
   (for host types 2 or 4)
2. verify passwords
   (for host types 3 or 4)
3. provide password protection
   (optional check for host types 1 or 2)

3.5.1.1.1. Trustworthy Hosts with User Numbers

If the host computer is trustworthy and provides the IDM with the user's huid, no entry is required in the login relation. If an entry is present, the user must match the specified password. The optional login entry gives the user additional protection. Someone must know the user's host password and the user's IDM password to use the IDM.

The BLI parallel and serial drivers for VAX/VMS and VAX/UNIX use this technique. Either VMS or UNIX authenticates the user and identifies the user with a 4 byte number. The IDM "trusts" the number supplied by VMS or UNIX.

3.5.1.1.2. Trustworthy Hosts with User Names

If the host computer is trustworthy and securely provides the IDM with the user's name, the IDM will look for an entry in the login relation which matches the given hid and huname. In this environment, an entry must be present in the login relation for the user to use the IDM. The password here, is completely optional. If it is present in the login relation, the user must supply a matching password.

The BLI block multiplexor driver uses this technique for VM/CMS. Users are identified on CMS by name. The driver securely passes that name to the IDM. The IDM "trusts" that the name is correct and cannot be forged.

If a host sends trustworthy hunames and the login relation is empty, any user from this host will be assigned an huid of 0. This simplifies installing the IDM, and is useful in development environments.

3.5.1.1.3. Untrustworthy Hosts with User Numbers

If the host is untrustworthy, the user must supply a password which matches an entry in the login relation. The host supplies an hid and huid but the IDM will not trust the correctness of those numbers. The login relation is searched for an entry with a matching hid and huid. If no entry is found or if the user does not provide a matching password, no access to the IDM is allowed.

3.5.1.1.4. Untrustworthy Hosts with User Names

If the host is untrustworthy, the user of the IDM must provide a name and password. The host still provides the hid. An entry must be found in the login relation with a matching hid, huname and password.

This environment is typical of personal computers and work stations. The host is untrustworthy because the user can directly program the interface to the IDM.

3.5.1.2. Configuring Hosts to Specify Type of Authentication

Hosts can be marked as sending trustworthy or untrustworthy huids or hunames in the configure relation of the system database. Hosts can be marked as sending untrustworthy huids by setting bit 11 (octal 04000) in the value field for parallel, serial, blkmux and especially Ethernet channels (type = "P", "S", "B", or "E"). This would be required for host type 3 and 4. Although host type 4 normally sends an huname, if no huname were sent, the huid would be used for the login authentication. This value would also be untrustworthy and so host type 4 should also be configured as sending untrustworthy huids. Hosts are assumed to send trustworthy huids by default.

Hosts can be marked as sending trustworthy hunames by setting bit 10 (octal 02000) in the value field. This would be required for host type 2, such as a VM/CMS system.

Hunames are untrustworthy by default.

3.5.1.3. Login Relation

The login relation maps an hid/huname into an hid/huid and it provides passwords. It is a system relation which exists only in the system database. Only the DBA of the system database can access this relation. The DBA is responsible for appending tuples for all users requiring login tuples (generally all users from hosts not sending a trustworthy huid). This relation is initially empty. In the typical environment, the DBA can provide a stored command which allows a user to change his own password. The "type", "hid", "huid", "huname" and "class" fields can be made readable if desired. The definition of the login relation is as follows:

```
login(type, hid, huid, huname, password, class)
```

type - determines whether the tuple applies only to trustworthy hosts, all host, or is disabled.
- "T" - untrustworthy hosts will not be allowed access for login tuples with this type.
- "N" - no host will be allowed access for login tuples with this type. This is a way of disabling an account without deleting the tuple.
- "A" or " " - any host can match this tuple.

If the type field is anything else, it is assumed the same as "N".

hid - this matches the hid supplied by the host. A value of -1 is a "wild card" and will match any hid if there is no exact match.

huid - this matches the huid supplied by the host for host types 1 and 3. If the login is by name, this becomes the assigned huid for the user.

huname - this matches the huname for host types 2 and 4. A value of "guest" will match any huname from a given host if there is no exact match. If the login is by number (host types 1 and 3), this attribute is ignored.

password - this is the password which the user must match for host types 3 and 4. If the password is assigned the value "" (empty string), it means no password is present. For host types 3 and 4, the password must be present and the user must match it. For host types 1 and 2, the password is optional. If it is present, the user must match it.

class - This attribute is available for the DBA to use as desired. The IDM ignores its value. The attribute could be used to hold additional information about the user or it could be used to note the "lifetime" of the account. For example, it might the show the expiration date of the account. The DBA can then run a REPLACE command to set the type attribute to "N" (disabled) based on examining the value in the class attribute.

3.5.1.4. Sending Identification to the IDM

3.5.1.4.1. Sending a Host User Id

The huid is sent to the IDM in the "host communications packet" (see section 6.3.1 "Communications Packets and Data Packets"). This packet is normally trustworthy since it is usually sent by the host operating system and cannot be specified by the user. In an Ethernet environment it may be possible for a user process to send its own huid (and hid), so the login identification from any host communicating over an Ethernet could be untrustworthy. By default, huids are considered trustworthy, as described above in the configuration section.

3.5.1.4.2. Sending a Host User Name

The huname is sent in the same data packet as the first open database command sent by any user process (see section 4.4 "Example of Flow of Control" and section 6.3.1). This is the open database command sent for DBIN = 0. A DBIN of 0 is sent when a user process is first created on the IDM. The IDM creates a new DBIN (not = 0), which is returned to the host for use with succeeding commands from the same user process. The huname is sent before this open database command. The HUNAME token is sent followed by the length of the huname and then the huname. If more than one huname is sent before the first open database command, the IDM accepts only the first one sent. Hunames may only be sent with the first open database command for a user process on the IDM (DBIN = 0).

If an huname is sent, it overrides whatever huid was sent in the "host communications packet". If the host operating system normally sends an huname and is able to validate this huname, it can be considered trustworthy. If the host operating system does not normally send a validated huname, any huname sent should be considered untrustworthy, since any huname sent will override the huid. By default, hunames are considered untrustworthy, as described above.

3.5.1.4.3. Sending a Password to the IDM

Passwords can be sent to the IDM in the same data packet as the first open database command sent by any user process (see section 4.4 and section 6.3.1). This is indicated by the PASSWORD token. It is followed by the length of the password and then the password. The password is sent before this open database command. Passwords may only be sent with the first open database command for a user process on the IDM (DBIN = 0).

3.5.1.4.4. Changing a Password

A password may be changed by updating the login relation in the system database or by using the "new password" command to dbin 0 (see section 7.5.29A). This command is only valid to dbin 0 and must follow a valid huname and password. It is not possible to change the "guest" login tuple password with this command.

3.5.1.5. Valid Types for Login Tuples

Users sending trustworthy or untrustworthy logins must have a type of "A" (all) or blank (uninitialized) in their login tuple. Users sending trustworthy logins may also have no login tuple at all or may have a type of "T" (trustworthy only) in their login tuple. A type "T" means the account may only be accessed if the login is trustworthy. This can increase security for users sending trustworthy login ids.

The DBA may turn off a user´s account by setting the type of a user´s login tuple to an invalid value such as "N" (no access). This prevents the user from logging onto the IDM.

3.5.1.6. Matching Login Tuples

When sending login identification to the IDM, the login relation is first searched for an exactly matching login tuple, matching either by hid and huname if an huname is sent, or by hid and huid if not.

If no exact match is found, the login relation is searched for a tuple with the same huid or huname and an hid of -1. A login tuple with an hid of -1 may be convenient if the user has accounts on many similar hosts, all with the same huid or huname. This avoids having a login tuple for the user´s account on every host.

For users sending an huname, if neither an exact match nor a matching login tuple with an hid of -1 is found, there may be a guest tuple in the login relation with the huname "guest" and a matching hid or hid of -1. If present, this tuple matches all unknown users from a given host or all hosts.

For any of the above matching tuples, the login tuple and the login id sent from the host must meet the requirements of a valid password and type, as described above; otherwise the user is denied access to the IDM.

3.5.1.7. Error Message Returned By Authentication System

Whenever users are denied access to the IDM, the message "Permission Denied" is returned for the database they were trying to open at the time. If users do not send a valid login id, they will get the message even if the database does not exist.

3.5.1.8. Overriding the Login System

The system DBA may override the protection mechanism of the IDM login system by turning the switch to maintenance mode and sending set 14 before the first open database command (see "Getting Out of Trouble" below). This allows the system DBA to configure any host as sending trustworthy hunames or to insert, delete or replace any tuples in the login relation. Hosts sending trustworthy hunames will need to use this feature when installing the IDM.

3.5.2. Protection For Individual Databases

3.5.2.1. Database Creation and Ownership

Database creation is controlled by the DBA of the SYSTEM database. To create a database, you must either (1) be the DBA of the SYSTEM database or (2) be granted permission to create a database by the DBA. In either case, the "creator" of the database is the only one who can destroy the database. The creator of the database is also made the default DBA of the newly created database. The DBA of a database is the owner of the system relations.

Every database in the IDM is independent and has its own protection system. If, for example, user 2 created a database called "demo", three events would happen:

1. An entry would be placed in the "databases" relation
2. The database would be created
3. The "host_users" relation in "demo" would be initialized

The entry in the "databases" relation would show that user 2 in the "system" database is the owner of the database. In addition, the DBA of "demo" would be any user in "demo" with uid 2. The "host_users" relation in "demo" would have one tuple with

```
|hid   |huid          |uid   |
|----------------------------|
|     1|            12|     2|
    |----------------------------|
```

To enable another user to be the DBA of "demo", a tuple is appended to "host_users", for example,

```
    append host_users(hid=1,huid=10,uid=2)
```

would let user 10 on host 1 also become the DBA of "demo".

A "host_users" relation exists in every database. It identifies who can use the database and their user identification, for example:

```
    open system
    range of h is host_users
    retrieve (h.hid,h.huid,h.uid)
    go
    |hid   |huid          |uid    |
    |----------------------------|
    |     1|            10|      1|
    |     1|            12|      2|
    |     1|            20|      1|
    |----------------------------|
```

In this example, there are three users allowed to use the database. Users 10 and 20 on host 1 are assigned IDM user id 1 in the database. User 12 on host 1 is assigned IDM user id 2. This relation allows many users on the same or different hosts to be the same logical user in a particular database in the IDM.

3.5.2.1.1. Helpful Hints

The IDM has two built-in functions which give the DBA and current user id, for example:

```
    retrieve (administrator = dba, user = userid)
```

"Dba" returns the id of the DBA of the current database. "Userid" returns your id in the current database.

After the IDM is initially formatted, the SYSTEM database is created and is left unprotected, that is, everyone is the DBA. This changes as soon as a tuple is appended to "host_users". If you are initially setting up an IDM and are not sure what your "hid" and "huid" are, you can use the "dbinstat" relation to find out. Run the command:

```
    open system
    range of d is dbinstat
    retrieve (d.hid,d.huid)
    where d.dbin > 1
    go
```

If you are the only user on the IDM, the tuple returned will show you what your host told the IDM your "hid" and "huid" are.

3.5.2.1.2. Effects of LOAD DATABASE

The IDM allows whole databases to be dumped and reloaded. The IDM guarantees that the owner of a database is always the DBA of the database after a LOAD DATABASE is completed. It does this by appending a tuple to the "host_users" relation with the hid, and huid of the database owner and the uid of the DBA. This is necessary since the owner of the database may not be the DBA of the database being loaded. This can happen, for example, if a user tries to load someone else's database or a database from another IDM system. The owner of the database does not change.

3.5.2.1.3. Getting out of Trouble

It is possible for the DBA of a database to accidentally change the host_users relation such that no one is the DBA. This problem can be corrected with help from the DBA of the system database. The DBA of the system database can override the normal protection system with command-option 14. With this option set, the DBA of the system database can become the DBA of any database. For example,

```
open system go
set 14
open demo go
```

The user is now the DBA of "demo". As the DBA, (s)he can now fix up the host_users relation. Note that command-option 14 only has effect if it is used in the system database by the DBA of the system database. It is silently ignored in all other cases.

If the system database is inaccessible, the IDM can be brought up in "maintenance" mode and command-option 14 can be used to open the system database. Users of hosts sending hunames will need to use this feature when installing the IDM. For example:

1. turn IDM to MAINT
2. in response to the line "IDM 500/<n> Filename:" type "kernal" on the console
3. when the IDM comes up enter the following commands from a host:

```
mc
   set 14
   open system
   go
```

4. change "host_users" or "login" relations, or both, as required
5. turn IDM to RUN

3.5.2.2. Making Users Known in Individual Databases

The protection of relations, vi ws, files, and stored commands is communicated to the IDM with permit and deny commands. A typical protection command is:

```
    permit read of parts to george
```

The above command gives the user "george" permission to read the relation or file "parts". The permission is given by user name. This is the name in the "users" relation. It is not necessarily the same as the huname which the host computer may send to identify the user and which is used in the login relation of the system database. For each command that it processes, the IDM checks to be certain that the user who issued the command has the right to do so. For instance, if "george" accesses the relation "parts", the permissions for "parts" are checked to see if "george" has read permission for "parts".

However, when a command is sent to the IDM, the name of the user issuing the command is not necessarily supplied by the host system. Instead, the following identification is provided: a host ID (hid), which uniquely identifies to the IDM the host from which the command is issued, and either a user identification number (huid) or name (huname) which the host assigned to the user issuing the command. If the host computer sends an huname, the IDM authentication system maps the huname and hid into an huid when the user sends the first command to the IDM. This is done using the login relation in the system database. For every command sent to the IDM, the hid and huid are available so the protection system can check the user´s permission for that command.

To illustrate the host ID, let us assume that there are three hosts attached to an IDM: A, B, and C. When the IDM is turned on and each host establishes the first communication, each gives the IDM its hid: let us say A´s hid is 124, B´s is 312, and C´s is 515. The value chosen by the host system programmers as the hid does not matter to the IDM, as long as the hid is unique across all hosts attached to that IDM and as long as it is less than 32767 (fits in a 2-byte integer).

To illustrate the host user ID and host user name, let us assume that "george" has accounts on all three machines: on A he has huid 1212, on B he has huid 25, and on C, which we will say identifies users by name, he has huname "kinggeorge". Now, when he is on machine C and issues his first command to the IDM, the command is sent to the IDM with the hid/huname pair 515/"kinggeorge". For each command he sends, the IDM must associate that pair correctly with the protection associated with the user "george". This is done with three relations: the "login" relation, which maps every hid/huname pair to an huid (and is used for password checking if required); the "host_users" relation, which maps each hid/huid pair to the proper_uid; and the "users" relation, which relates the user name "george" (and other information about a user) with the unique uid for this user. The "login" relation is a system relation which exists only in the system database (see Authentication above). The "host_users" and "users" relations are system relations, and are present in each database.

```
    host_users (sl, hid, huid, uid)
    users (stat, id, name, gid, passwd)
```

For the system database, "users" and "host_users" are initially empty. This means that anyone who opens the database will be the DBA. The first person opening the system database should be careful to add a tuple to "host_users" to make them the DBA.

For user databases, "host_users" is initialized so that the creator of the database is the DBA for that database. The "users" relation is initially empty.

The DBA must fill in the "host_users" relation, supplying the host ID and host user ID for each user, and assigning the uid. Only users with a tuple in the "host_users" relation will be allowed to open that database. The "users" relation must also be updated, filling in the applicable information.

Continuing the example of "george", to allow him to communicate with the IDM from the C host, the following update must be made to the login relation in the system database:

```
    append to login (hid = 515,
                     huname = "kinggeorge",
                     huid = 464)
```

This single tuple is required only for "george's" account on the C host, which sends hunames to identify users. This tuple maps the huname "kinggeorge" on the C host (hid 515), to the huid 464. The huid 464 is not used in communicating with the host computer. No entry is required for his accounts on the other hosts, which send huids rather than hunames. The DBA of the system database is responsible for maintaining the "login" relation.

To allow "george" access to any given database, the following updates must be made to the "host_users" relation for that database:

```
    append to host_users (hid = 124,
                          huid = 1212,
                          uid = 7)
    append to host_users (hid = 312,
                          huid = 25,
                          uid = 7)
    append to host_users (hid = 515,
                          huid = 464,
                          uid = 7)
```

A record must be entered in the "host_users" relation for each different hid/huid pair that correspond to the same user. The uid is chosen by the DBA, and uniquely corresponds to the user. The huid for the user's account on any host sending hunames must match the value in the "login" relation for that user. Here, the huid must be 464 for the host with hid 515 (the C host).

The "users" relation maps between the "name" and the internally-used uid. The DBA also is responsible for maintaining the "users" relation:

```
    append to users (id = 7,
                     name = "george",
                     gid = 10,
                     status = 1)

    append to users (id = 10,
                     name = "clerks",
                     gid = 10)
```

The first command gives to user number 7 the name "george" and places him in group 10. The second command gives group 10 the name "clerks". A person can belong to only one group.

The IDM uses the "name" attribute to identify the uids of the users specified in the protection commands. To grant permission to an individual (or group), that individual (or group) must have a name, and therefore must have a tuple in the "users" relation.

3.5.2.2.1. Special Cases

The DBA should NOT use zero (0) for either uids or gids. These values are reserved for indicating permissions granted to all users. A tuple may be placed in the "users" relation to indicate this:

```
    append to users (id = 0,
                     name = "EVERYONE",
                     gid = 0)
```

Sometimes the DBA wishes to allow anyone to access a database. This can be done by adding a "guest tuple" to "host_users". If the DBA places in "host_users" a tuple with hid = -1 and huid = 0, then that is a "guest tuple":

```
    append to host_users (hid = -1,
                          huid = 0,
                          uid = 100)
```

Anyone may now open the database. If someone who does not have another tuple in "host_users", tries to open this database, (s)he will become (in this example) user 100. User 100 can be given a name:

```
    append to host_users (id = 100,
                        name = "ANYONE",
                        gid = 10)
```

and granted permissions, just like any other user.

If there are no tuples in "host_users", then anyone can open that database, and (s)he will be the DBA. This is the initial condition of the system database. It can be generated in any database, if the DBA deletes all the tuples in "host_users".

3.5.2.3. Granting Permissions

There are two protection commands: permit and deny. The types of protection are read, write, and execute. Relations, views, specific attributes of relations and views, files and stored commands can be protected. An example of a permit command is:

```
    permit read of parts to george
```

Now the user "george" is allowed to read the entire relation "parts".

```
    deny write of parts (cost) to george
```

The above command denies "george" the ability to change the "cost" attribute in the "parts" relation.

Groups are used to specify sets of users who have the same access rights. To use a group instead of an individual in any of the protection commands, the group name is specified instead of the user name. For instance,

```
    permit read of employee (salary) to clerks
```

The above command allows all the clerks to read the salary attribute of the relation "employee".

The order of protection commands is important. If no protections are issued for a relation it has protections equivalent to:

```
    deny all RELNAME to all
    permit all on RELNAME to RELOWNER
```

In the above commands, "RELOWNER" is the user who created the relation "RELNAME". If the owner now issues:

```
    deny write of RELNAME to all
```

then the owner could not write the relation. The effect of the above three commands is the same as the following command sequence:

```
    deny all on RELNAME to all
    permit read on RELNAME to RELOWNER
```

In addition to controlling access rights the permit and deny statements can be used to grant rights to use the create, create index and create database commands. Create database may only be granted in the system database. For example, the system administrator may issue the command:

```
    permit create database to dbas
```

which would allow those in the "dbas" group to create databases.

Users other than the database administrator may grant access to their relations, views and stored commands. Only the database administrator can grant access to objects owned by others and grant the use of the commands given above.

3.5.2.4. Views and Stored Commands

Views and stored commands can be used to refine the level of protection. If the owner of a relation defines a view on that relation, the rights granted to the view take precedence over those of the base relation. In particular, a user must have write permission on all attributes of a relation to append or delete tuples. If users are granted write permission on all attributes of an updatable view they may append to and delete from the base relation using the view. Stored commands written by the owner of the relations and views they reference also take precedence over protections of the objects. Never does a view or stored command which references objects owned by others affect the protections of those objects. The access rights to referenced objects are rechecked when such a view or stored command is accessed.

3.6. Performance Improvement Support

There are three areas for performance tuning the IDM:

1) creating or re-creating indices
2) assigning databases to separate disks
3) adjusting IDM hardware configuration

3.6.1. Create Index

An index is a directory that relates the physical location of each tuple of a relation to the value of an attribute or group of attributes of that tuple. The creation of an index improves the performance of the system by providing a direct access path to the data.

Indices are either clustered (the data is physically sorted according to the index) or non-clustered. To create a clustered index on the "parts" relation, the command is:

```
    create clustered index on parts (number)
```

The above command causes the IDM to sort the relation "parts" on the part number, then create a directory that relates the part number to the physical location of the associated part tuple. If the relation is already sorted (clustered) on another attribute, then the command to create an index on number is:

```
    create nonclustered index on parts (number)
```

One of the parameters of the create index command is whether the attribute or group of attributes to be indexed is unique. A unique attribute is one for which no duplicate values exist within that relation. The knowledge that an attribute is unique is useful to the IDM in planning fast command processing strategies. It is also helpful for the users to have the IDM enforce that an attribute is unique. There are some attributes, such as part number, that should be unique: no duplicate part numbers should ever be allowed into the relation. The command to create a unique index is:

```
    create unique clustered index on parts (number)
                          or
    create unique nonclustered index on parts (number)
```

After either of the above commands is executed, the IDM will check to see if a duplicate attribute exists before adding a tuple with that part number to the relation, and before updating the part number in the relation. If a duplicate part number exists, the IDM will either silently delete the part in the "parts" relation, or notify the user of the error and abort the update. Which action is taken depends on the command-option specified in the "command-options" section of the command. If the index is created with the delete-dups option, any duplicates will be "silently" deleted and a warning message issued. See Section 7 for more detailed information.

Indices should be created when the accesses to the data are mostly on a given key. The creation of a clustered index causes the relation to be sorted by the value of the index. Then if the data is mainly needed in that order, it can be read from the disk efficiently. If the relation is growing, addition of extra records may force the allocation of data blocks to become less than optimal, so that retrieving the data in sort order is less efficient. In that case, the index should be created again.

3.6.1.1. The Meaning and Use of Fillfactor and Skip

A fillfactor tells the IDM how full to make the blocks of a relation when a clustered index is created on it:

```
    create clustered index on parts (name)
            with fillfactor = 50, skip = 2
```

The above command causes the previously created "parts" relation to be sorted and an index created on it. The fillfactor is 50, which means that the relation blocks are to be filled 50% full. The fillfactor is necessary because if the relation is initially created with completely full blocks, random growth will tend to spread its blocks over several cylinders, thus creating head movement when the entire relation is read. The fillfactor reserves empty space within the relation blocks to minimize the "growth spread" from the random addition of data.

An additional way to reserve space is to use the "skip" command-option. "Skip" tells the IDM how many blank blocks to leave between data blocks.

The DBA should specify the fillfactor whenever creating a clustered index on a relation. Creation of an existing clustered index forces the relation to be re-sorted so blocks are filled with data in the order specified. Using the "with recreate" option prevents the data from being re-sorted and the fillfactor is ignored.

3.6.1.2. Setting the Index card Field

The attribute CARD in the indices relation has special meaning IDM's internal access path selection operations. If the attribute is non-zero and the index is non-unique, it is assumed to be an indication of how many tuples would qualify on a retrieve with an equal clause on the keys defined in the index. This is the selectivity of the index.

For example, assume a parts relation with an index on color. (indid = 1, relid = 6042)

partno	type	color	....
123	A	white	
234	B	white	
542	C	black	
679	B	green	
768	B	black	
470	A	yellow	

441	C	green
451	D	green

The selectivity for the color index should be 2. The best way to set this is to find the average of the count of distinct values in the index. In IDL this would look like:

```
range of i is indices
range of p is parts
replace i (card = avg ( count (p.color by p.color) ) )
        where i.relid = 6042 and i.indid = 1 go
```

This also holds for indices defined on more than one key. In this case, the count by list should reference all the key attributes. In the above example if a second index were defined on color and type, the IDL statement would be:

```
range of  i is indices
range of p is parts
replace i (card = avg (count (p.color by p.color, p.type) ) )
        where i.relid = 6042 and i.indid = 2 go
```

3.6.2. Adding a Disk

Data that may be referenced concurrently can be put on separate disks to reduce disk head contention, and therefore reduce access time.

When the IDM first arrives at a site it is configured with at least one disk. That disk is in IDM format, and databases can be created on it. To add new disks, the steps below are followed:

(1) The IDM is brought into "maintenance" mode by turning the key switch on the front panel to "maint".
(2) The new disk is plugged into the back panel.
(3) The disk formatting program is loaded from the diagnostic device, and executed.[1]

3.6.3. Putting Databases on Different Disks

A database is a physical entity: it consists of at least one zone (group of cylinders) on a given disk. The number of (2k–byte) disk blocks within a zone is variable, ——————

[1] See the IDM Operation Manual (part number 201–1078) for details of the disk formatting procedure.

This page has been intentionally left blank.

depending on the particular disk track size.

When a database is created, the logical disk name and number of zones are specified:

```
create database inventory with demand = 12500 on "disk2"
```

The above command created the database "inventory" and allocated it 12500 2-K byte blocks on the disk named disk2. The default value for the number of zones is 1; the database will reside in the first available space if the disk name is unspecified.

Databases should be put on separate disks to minimize disk head movement. If the DBA is aware that two databases will be highly active at the same time, she should allocate them to separate disks so they will not interfere with each other's performance.

3.6.5. Putting Relations on Different Disks

Section Removed.

3.7. Updating Views

Views have certain abnormalities when they are used as the update variable (or relation) in a delete, replace, or append. The following "rules" determine whether an update is legal:

1. Aggregates in target list (delete, replace, append): If an aggregate exists in the target list of the view, that is, an attribute in the view is defined as an aggregation, then the view cannot be updated.
2. Multi-variable views (delete, append): If the view is multi-variable, then it cannot be the object of a delete or append.
3. Multi-variable replace (replace): If a replace statement is changing more than one attribute of the view and the attributes involve more than one relation, then the update is illegal.
4. Non-simple view attributes (replace): If an attribute of a view is an expression other than name = variable, the attribute "name" cannot be updated.
5. Appends whose attributes are qualified (append): For all practical purposes, a view cannot be the object of an append statement. If any variable used in the target list of the view is also used in the qualification of the view, then the view cannot be appended to. This prevents phantom updates.

3.7.1. View definition

A view definition looks like a "retrieve into". The target list is a collection of RESATTR nodes (see Appendix B) and the qualification and target list expressions have no special restrictions. The tree is syntactically checked as if it were a "retrieve into". This results in the type and length of each RESATTR being computed.

Certain information is computed into the "view tree" in order to speed subsequent processing. The view as a whole and each of the target list elements are examined for updatability.

3.7.2. View substitution

When a command refers to a view, the view definition is substituted for the view references. This includes (1) appending the view qualification to the root of the query and to the root of all aggregates which reference the view, (2) If the view is being updated, verify that the update is legal and replace each update attribute number with the corresponding one in the view, (3) Every VAR node which is used in the view is replaced by the definition.

As previously mentioned, certain checks for updatability are done when the view is stored. The policy for updating views is:

> A view is updatable if the result of performing the update on the view and then materializing that view is the same as the result of materializing the view and then performing the update.

3.8. The Sizes of Tuples

3.8.1. Introduction

No tuple may be longer than 2000 bytes. Included within this 2000 byte limit are some fields required for the internal functioning of the IDM. This section shows how to calculate the length of a tuple.

3.8.2. The Declared Width of a Relation

Once a relation is created, its declared width can be calculated. To do this you must:

(1) Sum the width of all the attributes (using the maximum width for compressed attributes).
(2) Add 1 for the tuple ID.
(3) Add 1 if there are any compressed attributes.
(4) Add 1 for each compressed attribute.
(5) If the total is greater than 255:
    (5a) Add 1 if there are any compressed attributes.
    (5b) Add 1 for each compressed attribute.

This yields the declared width of a relation. A tuple can never be wider than 2000 bytes, thus, only a relation with compressed attributes can be declared wider than 2000 bytes. Even if the relation's declared width is wider than 2000 bytes, ALL TUPLES stored in that relation must still meet the 2000 byte limit.

3.8.3. The Length of a Tuple

To calculate the length of a tuple, you must know the declared width of the relation. To calculate a tuple's length, you must:

(1) Sum the width of all the attributes (use the width of the data for compressed attributes, ignore blanks)
(2) Add 1 for the tuple ID.
(3) If the declared width is less than or equal to 255:
    (3a) Add 1 if there are any compressed attributes.
    (3b) Add 1 if for each compressed attribute.
(4) If the declared width is greater than 255:
    (4a) Add 2 if there are any compressed attributes.
    (4b) Add 2 if for each compressed attribute.

The total must be less than or equal to 2000.

3.9. The Sizes of Indices

3.9.1. Introduction

This section describes algorithms for calculating the size of an index. The result size is expressed in terms of disk blocks (2048 bytes). There is an algorithm for non-clustered indices and one for clustered indices. Both are iterative solutions. Both describe the size of an index immediately after its creation, before updating causes any changes to the structure of the index. Thus the results should be regarded as minimum index sizes, since replace and append commands will only increase the size of an index or leave the size unchanged. Delete commands may decrease the size of an index but are not guaranteed to do so.

The exactness of the results of these algorithms relies on the exactness of the attribute sizes used in the calculations. If any attributes are compressed then average attribute sizes must be used; the resulting index sizes are then approximate.

3.9.2. Clustered Indices

A clustered index tuple consists of a key value(s) and a pointer to a data page. A clustered index has one index (tuple) for each data block, and the data blocks themselves are the leaves of the index tree. The algorithm is as follows:

(1) Set B[0] = the number of data blocks in the relation.

If B[0] = 1, the clustered index will occupy no space.

(2) Set W = the width of an index entry (tuple).
W must be <= 255 bytes. W is the sum of the widths of the attributes on which the index is defined, plus 1 byte for each compressed attribute in the index, plus 3 bytes for a pointer, plus 1 byte if there are any compressed attributes in the index.

(3) Set N = 1.
N is the index level; "1" denotes the bottom level (ignoring the data blocks).

(4) Set E = 2034 / W.
Discard any remainder after the division. E is the number of entries per block in the index.

(5) Set B[N] = B[N - 1] / E.
If there is a nonzero remainder after the division, add 1 to B[N]. B[N] is the number of index blocks at level N of the index.

(6) If B[N] is not 1, set N = N + 1 and go back to step 5 above; otherwise the sum of all the B[N]'s for N >= 1 is the number of blocks in the index.

3.9.3. Nonclustered Indices

A nonclustered index tuple consists of a key value(s) and a pointer to a data tuple. A nonclustered index has one entry (tuple) for each data tuple in the relation, and the data tuples themselves are the leaves of the index tree. The algorithm is as follows:

(1) Set W = the width of an index entry at the leaf (bottom) level.[2]

W is the sum of the widths of the attributes on which the index is defined, plus 1 byte for each compressed attribute in the index, plus 4 bytes for a pointer, plus 1 byte if there are any compressed attributes in the index.

(2) Set E = 2034 / W.

Discard any remainder after the division. E is the number of entries per block in the leaf level of the index.

(3) Set B[1] = (number of tuples in the relation) / E.

If there is a nonzero remainder after the division, add 1 to B[1]. B[1] is the number of index blocks at the leaf level of the index.

(4) If B[1] = 1, the index has only one block and the procedure is complete. Otherwise, proceed.

(5) Set W = W + 3.[2]

Each entry is 3 bytes larger above the leaf level.

(6) Set E = 2034 / W.

Discard any remainder after the division. E is now the number of entries per block in all levels above the leaf level of the index.

(7) Set N = 2.

N is the index level.

(8) Set B[N] = B[N - 1] / E.

If there is a nonzero remainder after the division, add 1 to B[N]. B[N] is the number of index blocks at level N of the index.

[2] W must be <= 255 bytes at both steps.

(9) If B[N] is not 1, set N = N + 1 and go back to step 8 above; otherwise the sum of all the B[N]'s for N >= 1 is the number of blocks in the index.

3.10. Tape Specification

3.10.1. Functionality

A maximum of one tape controller (TPC) is allowed per IDM. The TPC interfaces to a Pertec type PCC Microformatter. A single TPC can be connected to two microformatters which in turn can be connected to four transports each. Thus, up to 8 tape transports can be controlled by one TPC. At this time, there are at least four major companies building tape drives that are Pertec PCC Microformatter compatible. These are Control Data Corp., Pertec, Kennedy, and Cipher Data. They all offer a variety of traditional and streamer type transports.

In general, the IDM Tape Controller supports industry standard 1600 BPI, PE, 9-track tape, with the guarantee that it can read every byte of data that anyone conforming to the American National Standard Code for Information Interchange, standards document X3.39-1973, writes on a tape. In addition, 800 BPI NRZI format and 3200 BPI PE format are supported, without any assurance of industry level compatibility. If the user generates a tape that conforms exactly with our format specifications, we will have no problems reading it even if it has not been generated on an IDM. Our tape system supports tape velocities up to 100 IPS and data transfer speeds up to 160 kbytes per second.

The commands which can send output or receive input from tape are:

```
copy in | out
dump database | transaction
load database | transaction
read | write file
```

3.10.2. Tape command-option

To indicate to the IDM that the input or output of a command is a tape, an OPTIONS token (41) followed by a tape command-option token (13) should precede the ENDOFCOMMAND token (208). The tape command-option token must be followed by 28 bytes which are the parameters to the tape operation. Thus the tape command-option uses 29 bytes: 1 byte for the tape command-option token itself and 28 bytes for the parameters to the tape command. The parameters are:

transport number	(1 byte)
this parameter is ignored	(1 byte)
file number	(1 byte)
mode	(1 byte)
old name of tape	(12 bytes)
new name of tape	(12 bytes)

The allowed transport numbers are in the range 0-7. Files on a tape are sequentially numbered starting at 0. The allowed file numbers on a write to tape are:

0	overwrite tape
1	append to end of tape

Modes are specified by turning on various bits in the mode parameter. The allowed modes are:

0001 check name
The "old name" parameter should match the name on the tape.

0002 do not write name
If this bit is not set, the "new name" parameter will overwrite the current name of the tape. Note that the entire header is overwritten in that case. If a tape has never been written before, the header must be written and thus the bit should not be set. If the bit is set, the header will not be overwritten.

0004 erase
Protection erase of tape before overwrite.

0010 host translate for copy
If the bit is set, data should be translated according to the host characteristics supplied with the identify communication command. If the bit is not set, no translation is performed, allowing the tape to be copied into a host with different characteristics.

0100 ASCII tape bit for copy
If the host translate bit is off and this bit is set, any character data on the tape being read (for a copy in) will be assumed to be ASCII and will be translated only if the data is being copied into an EBCDIC database. For copy out, this bit specifies that the character data on the resulting tape should be ASCII and will be translated only if the data is being copied out of an EBCDIC database.

0200 EBCDIC tape bit for copy
This is the EBCDIC corollary of the ASCII tape bit. If the host translate bit is off and this bit is set, it means, create EBCDIC tape for copy out, or assume EBCDIC tape for copy in.

Depending on the operation, some of these parameters become irrelevant and will therefore be ignored. The density can be set manually on the tape drive. An example of a command which requests input or output to tape is the following:

This page has been intentionally left blank.

```
    <command>
    <parse tree>
    <command-option token>
    29
    <tape token>
    <tape operation parameters>
    ENDOFCOMMAND
```

where the <tape operation parameters> take up 28 bytes. Other command-options may precede or follow the tape command-option.

3.10.3. Copyin and Copyout with tape

The copyin and copyout commands for use with tape, are sent to the IDM in the format described in the section above. For a copyin command, the host sends the copyin parse tree, followed by tape command-options, followed by the ENDOFCOMMAND token. The host then waits for the DONE structure from the IDM indicating that the copyin from tape has completed. Notice that the IDM does not expect a final ENDOFCOMMAND from the host which normally follows the data when copying from the host. The IDM expects to read the data followed by an ENDOFCOMMAND from the tape.

For a copyout command, the host sends the copyout parse tree, followed by tape command-options, followed by the ENDOFCOMMAND token. The host then waits for the DONE structure from the IDM indicating that the copyout to tape has completed. The done structure that is returned to the host is the final done that is normally received by the host following a copy out to the host. The IDM also places this final done structure on the tape followed by the ENDOFCOMMAND token, which is the format expected by the copyin command.

3.10.4. Configure relation

Entries in the configure relation which pertain to the tape have the following attributes:

type	"T"
number	transport number (in the range 0-7)
value	the two rightmost bits are ignored for the moment. The third bit from the right is for speed. 0 indicates low speed and 1 indicates high speed.

3.10.5. Permission checking and validation

To read or write to tape, the user must have tape permission. To permit or deny use of the IDM tape the Permit and Deny commands can be used. The modes for tape are:

mode	octal code
read tape	4
write tape	10
all tape	14

The DBA of the working database (currently open) grants or denys permissions on the IDM tape by issuing, while in the working database: permit/deny read/write tape to <username>.

If an incorrect tape name is provided on a read or write to a tape which had been given a name earlier, the IDM will respond with an error and will return the correct name of the tape. Thus, any user who has permission to use tape is a trusted user. The name of the tape is not a password but a protection against accidents.

Copy in parse. tree
tape command - options
ENDOF Command token

This page has been intentionally left blank.

3.10.6. Multiple tapes

The IDM can handle commands which have input or output which exceeds the capacity of one tape. When the IDM encounters the end of a tape while processing a command, it will send a DONE token to the host computer. The DONE status word will have its DONE_VOLUME bit (bit 16) set. The IDM will then wait for a response from the host. The host should respond with a one byte transport number followed by an ENDOFCOMMAND token. Upon receiving the response from the host, the IDM will resume operation on the indicated transport.

3.10.7. Tape Format

Tapes which are supported by the IDM must have a header block at the beginning. A header block is 8K bytes long. The first 12 bytes contain the name of the tape and are followed by a 1 byte tape sequence number. The remaining bytes of the header are ignored. The tape format supported by the IDM is the following:

```
    Header block (8K bytes)
    EOF
    File 0
    EOF
      .
      .
    File n
    EOF
    EOF
```

Two consecutive EOFs indicate the logical end-of-tape. If the physical end-of-tape is encountered on File n, the file has to be continued on another tape. In this case the format of the tapes is the following:

First Tape	Second Tape
Header blk (seq no 0)	Header blk (seq no 1)
EOF	EOF
File 0	Rest of file n
EOF	EOF
.	EOF
.	
File n	
Header blk (seq no 0)	
EOF	
EOF	

The names on the first tape and the second tape should be the same. Each file, in turn will consist of a sequence of records. Each record will be of 8K bytes followed by an end of record gap. Thus a file will look as follows:

```
    record 0
    eor
      .
      .
    record m
    eor
    EOF
```

All records in the file must be of 8K bytes. The last record is zero padded if necessary.

3.11. BCD Data Types

3.11.1. Introduction

The IDM supports both an integer and a floating point binary coded decimal (BCD) data type. The integer BCD data type is supported by integer arithmetic and the floating point BCD data type is supported by decimal floating point arithmetic in the IDM. Relational operators and conversion functions to and from other numeric data types and character strings are also provided. For applications involving strictly integer decimal numbers, the host may use integer BCD. However, for applications requiring decimal numbers with fractional parts or, numbers with greater magnitudes than can be supported by a fixed point integer BCD the user is provided with floating point BCD.

Traditionally, decimal numbers have been supported on hosts strictly as integers, if they are supported at all. In applications where a decimal point is employed, the application program or language compiler is required to maintain decimal point information and to adjust results in order to obtain meaningful data. The IDM support of decimal floating point numbers eliminates the need for this complicated processing in the host and provides the host with arithmetic capabilities that perhaps were not available previously.

3.11.2. BCD Constant Format

BCD numbers are sent to the IDM in a format similar to character strings, i.e. the BCD token is followed by the length of the BCD string in bytes, which is followed by the actual BCD string. Two BCD tokens exist, one for integer BCD numbers, "BCD", and one for floating point BCD numbers, "BCDFLT".

3.11.2.1. Integer Format

Section Removed.

3.11.2.2. Integer and Floating Point Format

This format allows the user to represent both integer and floating point BCD numbers. The "BCD" token is used to specify that the following BCD string is an integer and that integer arithmetic should be performed. The "BCDFLT" token indicates that the following BCD string is a floating point number and that floating point operations should be performed. In this format, the maximum string length is 17 bytes which allows for representation of 31 digits. BCD numbers are represented in this format by a sign (S) , an exponent (E) , and a significant digit field (I.F) such that the BCD number X is:

$$X = (-1)^{S} * 10^{E-1024} * (I.F)$$

and is stored as:

```
   BCD string
   ------------------------------------------------------
   |S|     E      | I |              F                   |
   ------------------------------------------------------
bit 0 1          12   16
```

```
        S = sign bit
        E = exponent, 11 bit binary integer
                biased by 1024
        I = leading digit
        F = fractional digits packed 2 digits per byte
                with a maximum of 15 bytes or 30 digits
```

Examples of this format are given below. The BCD string itself is specified in hexadecimal and numbers followed by "x" are hexadecimal constants.

NUMBER   FORMAT

token	length	string.	...		
1	2	3	4	5	6

+123. (integer) =

$1.23 * 10^{2}$ S = 0 E = 2x + 400x I = 1x F = 2x 3x

BCD	3	4 : 0	2 : 1	2 : 3

-1234. (floating point) =

$-1.234 * 10^{3}$ S = 1 E = 3x + 400x I = 1x F = 2x 3x 4x

BCDFLT	4	C : 0	3 : 1	2 : 3	4 : 0

+12.34 (floating point) =

$+1.234 * 10^{1}$ S = 0 E = 1x + 400x I = 1x F = 2x 3x 4x

BCDFLT	4	4 : 0	1 : 1	2 : 3	4 : 0

-.0006 (floating point) =

$-6. * 10^{-4}$ S = 1 E = 7FCx + 400x I = 6x

BCDFLT	2	B : F	C : 6

+.0006 (floating point) =

$+6. * 10^{-4}$ S = 0 E = 7FCx + 400x I = 6x

BCDFLT	2	3 : F	C : 6

+9000. (integer) =

$+9. * 10^{3}$ S = 0 E = 3x + 400x I = 9x

BCD	2	4 : 0	3 : 9

Numbers which contain the BCD token must be integers. A number containing a fractional part must not be represented in the format as a BCD integer and will be regarded as an illegal operand.

Numbers with the largest and smallest exponents are reserved operands, i.e. numbers with biased exponent E = 0x or E = 7FFx. Therefore, the exponent range supported is:

$$10^{-1023} \text{ to } 10^{1022}$$

Zero may be represented as a zero length string as in the previous format. Additionally, the biased exponent E = 0 and significant digit field I.F = 0 is also used to represent zero. The remaining reserved operands are currently not accepted as legal BCD numbers and will not be returned as results of any operation. These operands are currently reserved for future enhancements.

```
E = 0     I.F <> 0
E = 7FFx  I.F  = 0
E = 7FFx  I.F <> 0
```

3.11.3. Creating a BCD Attribute

Both BCD data types may be stored in the IDM in a compressed (variable length) form or an uncompressed (fixed length) form. Therefore, the IDM supports four BCD data types:

bcd	variable length integer
ubcd	uncompressed integer
bcdflt	variable length floating point
ubcdflt	uncompressed floating point

To create a relation with a BCD attribute, the user must specify one of the above data types followed by the number of significant decimal digits desired. For example, to create a relation, "employees", with a variable length floating point BCD attribute, "salary", the CREATE statement is used as follows:

```
create employees (salary = bcdflt7)
```

which specifies a floating point BCD attribute with 7 significant digits. An uncompressed integer BCD attribute would be specified:

```
create employees (salary = ubcd7)
```

in which case the attribute field would contain a BCD integer with 7 decimal digits.

Due to the storage format of the BCD data type, a BCD number is always stored with an odd number of significant decimal digits. The user may create a BCD attribute of even length, n, in which case storage will be allocated for n + 1 decimal digits.

The number specified in the "create" statement has a different meaning for the intege and the floating point BCD data type. In the case of an integer BCD attribute, the number specified indicates the number of decimal digits to store, implying the range of the numbers that may be stored in the resulting attribute. For floating point BCD, this number specifies the number of significant digits, i.e. the precision of the number. The range of all floating point BCD numbers is determined by the exponent provided in the BCD data type.

3.11.4. Arithmetic and Conversion Functions

Both integer and floating point BCD data types are fully supported with comparison, arithmetic, and conversion functions. The set of arithmetic operators include:

```
compare  (<, >, <=, >=, =, !=)
add      (+)
subtract (-)
multiply (*)
divide   (/)
mod
negate   (-)
```

Conversion is supported to and from the data types: bcd, bcdflt, string, int1, int2, and int4. Conversion to a BCD data type from an expression of type bcd, bcdflt, string, int1, int2, or int4 is provided by the following functions. Strings are used as the expression to be converted in the examples below.

```
bcd(precision, expression)
  converts "expression" to a BCD integer
  with a maximum of "precision" digits
  e.g.  bcd(5, "123") returns a BCD integer
               whose value is 123

        bcd(3, "12345") returns OVERFLOW

        bcd(4, "1234.56") returns the
               truncated integer 1234

bcdflt(precision, expression)
  converts "expression" to a floating point
  BCD number with a maximum of "precision"
  significant digits, rounding the value if
  necessary
  e.g.  bcdflt(4, "123.45") returns a floating
               point BCD number whose value is
               123.4:
                          2
                1.234 * 10

        bcdflt(5, "1234567.89") returns a floating
               point BCD number whose value is
               1234600:
                           6
                1.2346 * 10

bcdfixed(precision, fraction, expression)
  converts "expression" to a fixed floating point
  BCD number with a maximum of "precision" digits,
  and a maximum of "fraction" significant fractional
  digits, rounding the value if necessary
  e.g.  bcdfixed(5, 2, "768.534") returns a fixed
               floating point BCD number whose
               value is 768.53:
                           2
                7.6853 * 10

        bcdfixed(4, 3, "123.45") returns OVERFLOW

        bcdfixed(8, 2, "35.478") returns a fixed
               floating point BCD number whose
               value is 35.48:
                               1
                3.54800000 * 10
```

If the precision parameter is 0, the number of digits required to store the converted expression is used. In addition, the functions "bcd" and "bcdflt" can be preceeding by "fixed" to indicate that the "precision" parameter specifies the exact number of digits to be generated. The number is padded with zeros to the right of the decimal point to accommodate the field. It does not affect the arithmetic routines.

Conversion from the BCD data types is supported by the various other data type conversion functions:

```
string(length, expression)
int1(expression)
int2(expression)
int4(expression)

where:
        expression = a BCD data type
```

The user may use the bcd, bcdflt, and bcdfixed conversion routines to convert from character strings to the BCD data type, as well as to convert from one BCD precision format to another. The precision parameter always indicates the "maximum" number of digits required and the resulting BCD number will only be padded to support the maximum number if the "fixed" option is specified.

3.11.5. Rounding and Exception Handling

The rounding performed for conversion and arithmetic operations guarantees that a single rounding error produces a result that is within half of a unit in the last digit from the exact result. This ensures that the result produced is the closest number representable in the format to the exact result. In the case of a tie, the even number is chosen.

The following exceptions are detected during BCD arithmetic operations:

overflow - integer	number of digits in result exceeds the maximum BCD field or specified precision - defaults on/returns largest integer with appropriate sign
overflow - floating	result exceeds exponent range for floating point BCD - defaults on/returns largest floating point number with appropriate sign
underflow - floating	exponent of result is smaller than smallest floating point BCD exponent - defaults on/returns 0
divide by zero	division operation occurred with divisor zero - defaults on/returns largest integer or floating point number with appropriate sign
inexact - floating	rounding error occurred meaning a loss of significant digits - defaults off/returns correctly rounded result
badbcd	an illegal BCD representation was sent - defaults on/returns 0

The user may reverse these defaults by specifying the appropriate options in query control. If the option is set, the exception will cause the IDM to stop processing and return the error status. If it is not set, the exception will be ignored and processing will continue with the default value specified in the table above. Care should be taken in ignoring the "inexact" exception. If the user does not specify a sufficient number of significant digits to contain a result, any loss of significant digits (providing the exponent of the number is still within range) will result in a rounding error or "inexact" exception. This includes digits in the integer part of the number.

3.12. EBCDIC Support

It is possible to use the IDM with host computers which use EBCDIC character representation. The IDM uses ASCII representation internally. For EBCDIC hosts, the IDM will translate all character data communications from EBCDIC to

ASCII on the way in, and from ASCII to EBCDIC on the way out. Examples of commands which communicate character data are retrieves, appends and copy in and out. The dump, load, file read and write commands send and receive physical data, hence no translation is done for these commands.

When ASCII character data is sorted on the IDM, the ordering will be the normal ASCII sort order. This ordering has digits first, then upper case letters, then lower case letters.

To use an EBCDIC host, the user must specify in the host identify packet that the character representation is EBCDIC (see chapter 6). Also, the host representation of pattern matching characters must be as follows:

pattern matching character	EBCDIC host	ASCII host
*	0334	0200
?	0335	0201
[	0336	0202
]	0337	0203

For host computers that use the EBCDIC character set, user databases (other than the system database) may be created so that characters are stored internally using the EBCDIC character representation. This will improve performance by avoiding the translation step between the IDM and the host. It will also mean that character data sorted on the IDM will be in the same order as on the host. This normal EBCDIC ordering has lower case letters first, then upper case letters, then digits. ASCII hosts can access EBCDIC databases. Character data will be translated appropriately when going between the host and the IDM.

To create a database that uses the EBCDIC character set, the EBCDIC with node option (value 14) can be sent with the create database command. Databases may be explicitly specified as being ASCII by sending the ASCII with node option (value 13). By default, databases will be created to use the ASCII character set. To change this default so that databases are created to use the EBCDIC character set (without having to send an EBCDIC with node option), a tuple may be appended to the configure relation of the system database with a type of "D" and a value of 1. A value of 0 with this type, or no configure tuple with type = "D" means create ASCII databases by default.

The stat field of the databases relation of the system database indicates the character representation of the database. The second bit (value 2) is set if the database is EBCDIC and clear for ASCII databases.

Dumping and loading databases do not translate character data. If an ASCII database is loaded into a database that was EBCDIC, the type of the database will be ASCII.

This page has been intentionally left blank.

ASCII to IBM EBCDIC:
Using the ascii bit pattern as an index into the array, the value is the corresponding ebcdic bit pattern.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0	1	2	3	37	2d	2e	2f	16	5	25	b	c	d	e	f
10	10	11	12	13	3c	3d	32	26	18	19	3f	27	1c	1d	1e	1f
20	40	5a	7f	7b	5b	6c	50	7d	4d	5d	5c	4e	6b	60	4b	61
30	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	7a	5e	4c	7e	6e	6f
40	7c	c1	c2	c3	c4	c5	c6	c7	c8	c9	d1	d2	d3	d4	d5	d6
50	d7	d8	d9	e2	e3	e4	e5	e6	e7	e8	e9	ad	e0	bd	5f	6d
60	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
70	97	98	99	a2	a3	a4	a5	a6	a7	a8	a9	c0	4f	d0	a1	7
80	dc	dd	de	df	24	15	6	17	28	29	2a	2b	2c	9	a	1b
90	30	31	1a	33	34	35	36	8	38	39	3a	3b	4	14	3e	e1
a0	41	42	43	44	45	46	47	48	49	51	52	53	54	55	56	57
b0	58	59	62	63	64	65	66	67	68	69	70	71	72	73	74	75
c0	76	77	78	80	8a	8b	8c	8d	8e	8f	90	9a	9b	9c	9d	9e
d0	9f	a0	aa	ab	ac	4a	ae	af	b0	b1	b2	b3	b4	b5	b6	b7
e0	b8	b9	ba	bb	bc	6a	be	bf	ca	cb	cc	cd	ce	cf	da	db
f0	20	21	22	23	ea	eb	ec	ed	ee	ef	fa	fb	fc	fd	fe	ff

IBM EBCDIC to ASCII:
Using the ebcdic bit pattern as an index into the array, the value is the corresponding ascii bit pattern.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0	1	2	3	9c	9	86	7f	97	8d	8e	b	c	d	e	f
10	10	11	12	13	9d	85	8	87	18	19	92	8f	1c	1d	1e	1f
20	f0	f1	f2	f3	84	a	17	1b	88	89	8a	8b	8c	5	6	7
30	90	91	16	93	94	95	96	4	98	99	9a	9b	14	15	9e	1a
40	20	a0	a1	a2	a3	a4	a5	a6	a7	a8	d5	2e	3c	28	2b	7c
50	26	a9	aa	ab	ac	ad	ae	af	b0	b1	21	24	2a	29	3b	5e
60	2d	2f	b2	b3	b4	b5	b6	b7	b8	b9	e5	2c	25	5f	3e	3f
70	ba	bb	bc	bd	be	bf	c0	c1	c2	60	3a	23	40	27	3d	22
80	c3	61	62	63	64	65	66	67	68	69	c4	c5	c6	c7	c8	c9
90	ca	6a	6b	6c	6d	6e	6f	70	71	72	cb	cc	cd	ce	cf	d0
a0	d1	7e	73	74	75	76	77	78	79	7a	d2	d3	d4	5b	d6	d7
b0	d8	d9	da	db	dc	dd	de	df	e0	e1	e2	e3	e4	5d	e6	e7
c0	7b	41	42	43	44	45	46	47	48	49	e8	e9	ea	eb	ec	ed
d0	7d	4a	4b	4c	4d	4e	4f	50	51	52	ee	ef	80	81	82	83
e0	5c	9f	53	54	55	56	57	58	59	5a	f4	f5	f6	f7	f8	f9
f0	30	31	32	33	34	35	36	37	38	39	fa	fb	fc	fd	fe	ff

3.13. Safepoints and checkpoints

The IDM defaults to "checkpointing" all active databases every ten minutes. The default can be changed by specifying the checkpoint in the "configure" relation. For example, to checkpoint every five minutes:

```
    open system
    append configure (type = "C", value = 5)
```

Type "C" specifies "checkpoint". The value is the time in minutes between checkpoints. A count of zero turns off checkpointing. If no tuple is present with type "C", then a default of ten minutes is used. A safepoint is a checkpoint which occurs when no transactions are any longer active on a database.

3.14. Performance Monitoring

Two relations, "monitor" and "devmonitor", are included in the "system" database for reporting performance information. All times measured in these relations are in sixtieths of seconds. The updating of these relations is enabled through the "configure" relation.

To enable performance monitoring, append a tuple with type "M" to the configure relation of the system database. The value field specifies the number of minutes in the monitoring interval. A value of zero (0), or no configure tuple with type "M", turns monitoring off.

If monitoring is enabled, every monitoring interval a tuple is appended to the monitor relation, and a tuple for each peripheral device is appended to the devmonitor relation. The size of the monitor relation is limited to 3 2K blocks of disk space. This allows up to 54 tuples. When a new tuple is appended that would cause the monitor relation to grow beyond 3 blocks, the oldest tuple is first deleted. The devmonitor relation is also limited to 54 tuples for each device being monitored.

3.14.1. Monitor Relation

Each tuple in the "monitor" relation contains performance information for a time interval. The date and time at the end of the interval are specified in the _date_ and _time_ attributes. The amount of time in the interval is contained in the _length_ attribute. The attribute _seqno_ is used as link to tuples in the "devmonitor" relation.

A number of attributes in the "monitor" relation are described below. Attributes concerned with the uses of primary memory and activities of active dbins are described further on.

The attribute _cpu_ contains the amount of time the cpu or accelerator were busy during the interval. The attribute _dac_ gives the amount of time the database accelerator was busy or, in its absence, the amount of the time the cpu spent executing logically equivalent code. The attribute idle contains the amount of time none of the cpu, accelerator, tape controller or disk controllers were busy.

The attribute deadlocks gives the number of deadlocks that occurred during the interval.

The attribute cmnds holds the number of commands (interactions with the IDM, not full transactions) that completed during the interval. The attribute avgcmnd gives the average completion time of the completed commands. These times are roughly the same as the times returned using the "set 5" option (command-option 5).

This page has been intentionally left blank.

3.14.1.1. Primary Memory Usage

The IDM requires a fixed amount of primary memory to hold text and data for its kernel and system dbins. The remaining memory is used for disk page caching, dbin data space, the dbin table, host input buffering and host output buffering. Attributes are included in the "monitor" relation to measure usage for each of these memory categories and the penalties incurred for insufficient allocation.

If too little memory is used for any one purpose, bottlenecks can occur. Such bottlenecks can currently be corrected by adding memory to the IDM, thereby increasing the amount of memory for each category of usage. The IDM makes default choices on how memory is used. In later versions of the IDM it will be possible to override the defaults and specify how memory is to be used through the configure relation.

3.14.1.2. Disk Page Caching

The number of disk cache hits and misses are recorded in the _hits_ and _reads_ attributes. The number of writes are recorded in the _writes_ attribute. Because a page may have to be written out before it can be replaced in the cache, the number of disk page writes can also be reduced by enlarging the cache.

3.14.1.3. Dbin Data Space

During the processing of a transaction a dbin requires memory for its data space. The minimum number of unused pages that are reserved for this purpose is contained in the _unmem_ attribute. The _memloss_ attribute measures the amount of time the cpu is idle while there are dbins waiting for memory to process a command.

3.14.1.4. Dbin Table

The minimum number of dbins that were unused during the interval is recorded in the _undbin_ attribute. The number of times that a dbin could not be allocated for lack of a dbin table entry is given in the _dbinfails_ attribute.

3.14.1.5. Host Input Buffering

The minimum amount of unused host input buffering (in bytes) is given in the _unin_ attribute. The amount of idle cpu time while the kernel cannot accept host input from a channel is given in the _indelay_ attribute.

3.14.1.6. Host Output Buffering

When the number of output buffers for a process reaches a predetermined number, the process gives up the cpu until the output buffer count falls below an acceptable level.

IDM processes always expect to be able to use this predetermined number of output buffers. If there is no output buffer available when a process expects one, it does not give up the cpu until one becomes available. This will pose a serious performance deficiency if there are other active processes since they will not run until the output space is freed. The minimum amount of unused host output buffering (in bytes) is given in the unout attribute. The amount of time processing is suspended for lack of output buffers is given in the outdelay attribute.

3.14.1.7. Dbin Wait Sums

While a dbin is processing a transaction, it will wait for certain events. The following are a group of attributes that are useful for determining where active dbins spend their time. These are the sums of waiting times for all active dbins during the interval.

The attribute inwait gives the accumulated amounts of time that active dbins spent waiting for input from a host.

Dbins that are waiting for input but are not inside a transaction may have their memory stolen from them if another dbin is in need. When a dbin receives a command from the host, if no dbin memory is available, it will have to wait until some active dbin finishes its transaction. The attribute memwait gives the accumulated time that dbins spend waiting for memory.

The attribute cpuwait contains the accumulated time that dbins spend waiting to use the cpu.

The attribute diskwait gives the amount of time that dbins spend waiting for disk commands to complete.

The attribute tapewait contains the amount of time that dbins spend waiting for tape commands to complete.

The attribute outwait gives the accumulated time that dbins spend waiting for their output to drain to a host.

The attribute blockwait contains the amount of time that dbins spend waiting for a data "lock" or are temporarily suspended for a checkpoint.

3.14.2. Devmonitor Relation

The "devmonitor" relation tuples contain performance information for a single device during a time interval. The time interval for a "devmonitor" tuple may be determined by finding the "monitor" relation tuple with the equivalent seqno attribute. The attribute type in "devmonitor" gives the type of device. The possible types are given below. The attribute slot gives the IDM slot number of the device. Attributes d1, d2, d3 and d4 contain performance information depending on the type of device.

3.14.2.1. Disk Drives

Disk drives have the value "D" for the type attribute. The slot attribute contains the slot number of the disk controller the disk drive is connected to. The d1 attribute contains the amount of time the drive was in use. Attribute d2 contains the number of commands (page transfers) on the drive. The attribute d3 contains the low block number of the drive. To find the drive name, drive use and number of commands for each time interval (with the date and time ending the interval) the query would be:

```
range of m is monitor
range of dm is devmonitor
range of d is disks
retrieve(d.name, use = dm.d1,
               cmnds = dm.d2, m.date, m.time)
        where dm.type = "D"
        and dm.d3 = d.low
        and dm.seqno = m.seqno
```

3.14.2.2. Disk Controllers

Disk controllers have the value "N" for the type attribute. The attributes d1 and d2 contain the amount of time the controller was used and the number of disk commands completed, respectively.

If there are simultaneous commands for two different drives on the same controller, it is most probable that one of the commands will have to wait for the other. This is because once the arm of a drive reaches the correct cylinder, the controller dedicates itself to that drive until the completion of the command. To determine the amount of time a controller has commands for two or more drives simultaneously, subtract the use time for the controller from the sum of the use times for the relevant drives. This is done in the following query:

```
range of m is monitor
range of d is devmonitor
range of n is devmonitor
retrieve(m.date, m.time, n.slot, multiuse =
        sum(d.d1 by d.seqno, d.slot
                where d.type = "D") - n.d1)
        where n.type = "N" and n.slot = d.slot
        and n.seqno = d.seqno and d.seqno = m.seqno
```

Controller multi-use may be reduced by adding a controller or rearranging drives among controllers.

3.14.2.3. Tape Controller

The value of the type attribute for a tape controller is "T". Attribute d1 contains the controller use time for tape reads and writes. Attribute d2 contains the number of pages read or written. Attribute d3 contains the use time for commands other than reads and writ s.

3.14.2.4. Channels

The values of the type attribute are "S" and "P" for serial and parallel channels respectively. Determining whether additional channels will help system performance is a difficult problem that is not fully addressed by the monitoring information. However, one measurement of channel over-use is included.

Channels buffer results in their local memory. If the memory of the channel is filled with results that are not being requested by hosts and there are results from dbins that are not in channel memory that are being requested by a host, the channel will return some of its unrequested results to the IDM kernel to make room for the requested results. This process is known as aged results. The number of such occurrences is recorded in the d1 attribute.

3.14.3. Per-dbin Measurements

The IDM will return some of the above measurements on a per-dbin basis. The user can request these numbers by specifying options 33-46 (see section B). The measurements are returned after any data and before the DONE structure. The format of the messages is:

```
MEASURE n <type> xxx
```

where n is the option number. This is followed by one or more values as in an error message. The value will always be proceeded by its type as in an error message. All times returned are in 1/60 of a second intervals. Values may overflow on long commands.

3.14.4. Specifying Memory Usage

The amount of memory the IDM uses for certain purposes is djustable. The number of disk buffers, actively open relations, concurrently running dbins, and the amount of channel input and output space can be set by "K" tuples in the configure relation. The "number" attribute identifies the parameter to be adjusted. The "value" field indicates how the parameter should be changed.

NUMBER	VALUE
1	Sets the descriptor count, the maximum number of simultaneously open relations. Each open database requires 13 descriptors for its system relations; the system database is always open and needs 22. All dbins that access a relation share the descriptor, so the number of active dbins is not limited by the number of descriptors. Each descriptor uses 40 bytes in the "descriptor table". The default value is 128 (5K total in descriptor table).
2	Sets the dbin count, the maximum number of simultaneous connections to the IDM. Each dbin uses 64 bytes in the "process table". The default value is 96 dbins for the first megabyte of memory, 160 dbins for each megabyte of memory beyond the first megabyte (1K in process table per half-meg). The minimum is 30 dbins.
3	Sets the proportion of the free memory pool that will be used for disk cache buffers. See "Weighted Parameters"
4	Sets the proportion of the free memory pool that will be used for channel input space (queries sent to the IDM). See "Weighted Parameters"
5	Sets the proportion of the free memory pool that will be used for channel output space (results returning from the IDM). See "Weighted Parameters"
6	Set the number of megabytes of memory to use. This flag allows one to determine how the IDM would respond if it had less memory. The ´value´ field indicates how many megabytes of memory the system will use.

3.14.5. Weighted Parameters

The disk cache buffers, channel input and output space, and dbin data pages are adjustable based on a percentage of free memory after the IDM code has been loaded and descriptor and process tables have been allocated. There is a minimum allocation for each of the regions. After all the initial requirements have been fulfilled, the remaining memory is distributed among the different regions in the proportions specified, or in the following default proportions:

Default Memory Usage Weighting Table

Region	Proportion Of Free Pages Allocated To The Region
disk cache buffers	65%
input space	4%
output space	8%

The remaining memory is dynamically allocated and used by dbins to process queries and to maintain concurrency locks. Each active dbin can require from 5 to 9 2K dbin data pages; the exact amount of memory required depends on the size and complexity of each query.

Any or all of the above weights may be set independently. Thus, it is possible to increase the amount of output space by appending the following tuple to the configure relation:

```
append configure(type="K", number=5, value=12)
```

The weights for buffers and input space will be unchanged. The number of process and lock pages will be reduced by 4%.

If the sum of the user-specified percentages is more than 95, the message

```
Invalid kernal tuples in configure
```

will appear on the console and the default parameters will be used. This prevents accidentally setting the memory usage so there are too few dbin data pages for the IDM to run.

<u>Summary Of Configurable Memory Usage</u>

number	value	default	minimum	maximum
1	# of descriptors	128	30 descriptors	400 descriptors
2	# of dbins	1%	30 dbins	4094 dbins
3	% of buffer space	65%	20 2K pages	1000 2K pages
4	% of input space	4%	4 2K pages	40 2K pages
5	% output space	8%	8 2K pages	80 2K pages
6	# of megs to use	<all mem>	1 Meg	6 Meg

3.15. Query Processing Plans

3.15.1. Sending a plan to the IDM

When a query is executed, a plan is built by the IDM which describes the processing order and the access path used for each relation in the query. It is possible but not necessary to directly influence this plan by sending information on the PLAN statement. By specifying the relation number, an index id (or -1), and an order number, the user can tell the IDM which index to use for the relation and the relative order in which it should be processed. An index id of -1 tells the IDM to use a relation scan to process the relation. The PLAN statement for a relation must follow the RANGE statement that describes the relation.

It is also possible to send down a partial plan (i.e. not all the relations have corresponding PLAN statements). In this case, the IDM will finish constructing the plan for those relations not mentioned in the user plan.

The user plan is validated to ensure that any indices specified actually exist and that they are "useful". "Useful" means that a join or simple clause which involves that relation, references attributes which are keys in the index. An order number must be supplied. The order number is the relative processing sequence number of the relation with respect to other relations in the query.

For example using IDL syntax:

```
range of a is a with dindex=1, dorder=1
range of b is b with dindex=0, dorder=2
.....query...
```

Relation "a" will be processed first using the first nonclustered index and then relation "b" will be accessed via the clustered index.

```
range of c is c
range of a is a with dindex=3, dorder=10
range of b is b with dorder=2
range of d is d
.....query...
```

Relation "b" will be processed first using a relation scan (an index id of -1 was generated by the parser). Relation "a" is second in the processing sequence and it will be accessed via the nonclustered index with id = 3. Relations "c" and "d" have no user plans, so the IDM will select the access paths and the processing order for these two relations.

The plan information is stored in the range table entry for each relation. Stored commands can also have user plans, however, there are no dependencies stored for the indices that are referenced. If an index is destroyed, the next time that stored command is executed, the plan validation routine on the IDM will generate a user error.

See section 7 under PLAN for the complete description of the tokens that are required.

3.15.2. Displaying the Plan

Setting opti~n 60 returns informational messages about the plan that was used to execute a query. The plan describes the order in which relations were processed and what index (if any) was selected as the access path to each relation. This information is returned like the per-process monitoring statistics. (See section 3.14.3). A MEASURE token is returned followed by 60 (the option number), and 4 plan parameters.

The first parameter identifies the piece of the query. This will either be ROOT (for the main query) or AOPxxx (where xxx is SUM, CNT, etc) for each independent aggregate. If aggregates were combined, only the first type in the list is returned. Also, some aggregates automatically generate a CNT. This CNT is always the first in the list, even if a count was not in the query, the type on the plan will be AOPCNT. MIN, MAX, ONCE and AVG will always generate a CNT aggregate. See Appendix B for the actual token values.

The next parameter is the type of processing. The possibilities are: no variables, one variable, tuple substitution, or temp index made. For example, no variables, would look like:

```
retrieve (x=1).
```

"Temp index made" means the relation was restricted and a temporary clustered index was built to perform the join.

The next parameter is the relation nam ~r TEMP. TEMP is printed to indicate a temporary relation such as the result of an aggregate function or a reference to DBINSTAT or LOCK. Finally, the index number used is printed. 0 means the clustered index. Relation scan means no index was used, instead a serial scan of the relation occurred.

3.16. User Accounting

Accounting of the DBP and DAC usage of each user of the IDM is done in the account relation of the system database. The account relation has a tuple for each user, identified by the host computer id (hid attribute) and the user id on the host computer (huid attribute). The usage attribute is the number of 1/60ths of a second of DBP or DAC time for the user. The time and date attributes represent the last time the usage was updated. The accounting relation is updated by the IDM when a dbin is created on the IDM, when the dbin exits (sends a EXITIDM token), and when the time to be added to the usage for the user exceeds one minute and the dbin is between commands.

A system dba would periodically charge people by copying the account information and decrementing the usage count back down. For example:

```
range of a is account
retrieve into charge (a.all) where a.usage > 0
range of c is charge
replace a (usage = a.usage - c.usage)
    where a.hid = c.hid and a.huid = c.huid
```

4. Host Programming: An Overview

This section provides an overview of the necessary programming functions in a system that hosts an IDM. The following functions are performed by such systems:

Host Functions

1) Communicate with the users.
   The host system takes the user command as it is interactively typed at a terminal or as it appears in a batch program.

2) Translate user commands to IDM-internal form.
   This is usually a parsing function.

3) Send commands to the IDM.
   IDM communication consists of reading and writing data to/from the IDM. The protocols are defined in this document.

4) Receive results from the IDM.
   The IDM buffers results until the host requests them.

5) Format the results and display to the user.
   Formatting information is provided by the IDM.

Each of the above functions is described in more detail in Sections 5 and 6. This section consists mainly of the narration of a single example query, from the time the user enters it into the host system until the answer is displayed to the user. It is intended as an extensive, detailed analysis of the treatment of a single query: the general cases are addressed in Sections 5 and 6.

Section 4.1 is a discussion of the end-user interfaces required by functions (1) and (5), above. In Section 4.2 we trace the translation of an example user query, function (2). Section 5 is the detailed description of the end-user interfaces and the command translation.

Section 4.3 is a general discussion of the operating system functions required in functions (3) and (4). These functions are further described in Section 6.

Finally, Section 4.4 is the summary.

4.1. End-user Interfaces

The level of complexity of the end-user interface programs is a function of the needs of the application system. We will discuss two types of interfaces: those that support ad-hoc database commands, and those that support pre-planned commands.

In an ad-hoc system, the database commands are directly entered by the user; the major function of the host interface program is to parse the commands and send them to the IDM. The pre-planned system is more complex: the commands are not directly entered by the users, but implemented in a program that communicates with the users. The two types of interfaces actually overlap: one host can perform both types of interfaces. For clarity they will first be discussed separately.

4.1.1. Ad-hoc Systems

In an ad-hoc system, the user (either interactively, by typing on a terminal, or through the use of a previously written file) phrases all commands in a general purpose query language. An ad-hoc system allows the users to directly manipulate the IDM databases subject to the protection provided by the IDM. It allows the most latitude for casual users. Since it can also be used as an application development tool, it will typically be the first interface implemented on any host, even those that will be primarily dedicated to pre-planned commands.

An example of the use of such a system follows:

Example A. Ad-hoc user

User input:

```
range of d is department
retrieve (avgsal = avg(d.sal))
```

Host interface program:

1) parses each statement and checks for syntax errors
2) calls the operating system to send commands to the IDM.
3) formats and displays result to the user

If the application system is one that will support a great deal of interactive ad-hoc use of the IDM databases, it is recommended that a complete terminal monitor be implemented. Such a terminal monitor could include:

(1) an editor, to allow users to correct mistyped commands;

(2) a screen definition facility, to allow convenient formatting of the data;

(3) a facility to direct IDM output to other host programs, such as data analysis packages;

(4) macro definition and substitution capabilities which allow more freedom for the ad-hoc user;

(5) a process that reads a data file and puts it in tuple format for bulk loading the IDM using the copy command.

4.1.2. Pre-planned Commands

In many applications, the same commands are repeated continually. For such systems pre-planned commands are the most convenient to use.

The simplest implementation of pre-planned commands is a subroutine library that contains the IDM-internal form of the commands that are used. Such an implementation avoids the parsing of user commands completely, and simply calls the required subroutine when a database command is to be processed. Section 7 shows the IDM-internal form of the database commands.

A subroutine library implementation is a highly inflexible system. In most cases customized user interfaces will be implemented by embedding a general purpose query language in a high level programming language. An embedded command is one that appears in the midst of a program in another language. The following program fragment is an example of embedding IDL in FORTRAN.

Figure 1. Embedded IDL Example

```
C ***** FRAGMENT OF THE CUSTOMER ORDER TRANSACTION

200     CALL GETINFO (CUSTNO, ORDERNO, PARTNO, AMT)

M       RANGE OF P IS PARTS
M       RANGE OF C IS CUSTOMERS

C ***** FIRST GET AMOUNT ON HAND AND PRICE

M       BEGIN TRANSACTION
M       DO 300 RETRIEVE(ONHAND = P.AMTONHAND,PR = P.PRICE)
M               WHERE P.NUMBER = PARTNO

C ***** EXTRA PROCESSING OF RETURNED TUPLES GOES HERE
        .
        .
        .

M 300   CONTINUE


C ***** IS THERE ENOUGH TO SATISFY THIS ORDER?
        IF (ONHAND.LT.AMT) GO TO 400

C *****         YES - PROCESS ORDER
M               REPLACE P (AMTONHAND = P.AMTONHAND - AMT)
M                       WHERE P.NUMBER = PARTNO

C *****         UPDATE CUSTOMER´S AMOUNT OWED
M               REPLACE C (OWED = C.OWED + (PR * AMT))
M                       WHERE C.NUMBER = CUSTNO
        .
        .
        .
        GO TO 500
400     CONTINUE
C ***** IF THERE WAS NOT ENOUGH ON HAND, ADD TO ON ORDER

M       APPEND TO ONORDER (NUMBER = PARTNO,
M               AMOUNT = AMT - ONHAND)
        PRINT  30
30      FORMAT (" CAN´T DO ORDER")
        GOTO 600
        .
        .
        .
C ***** END OF TRANSACTION; RETURN TO ACCEPT ANOTHER
500     CONTINUE
        PRINT 40
40      FORMAT (" TRANSACTION COMPLETE")
600     CONTINUE
```

```
M       END TRANSACTION
        GO TO 200
          .
          .
          .
```

Figure 1 represents a section of a parts order system. The end-user, at a terminal, enters only the customer number, part number, price, and order number. The program, written in FORTRAN, controls what calls are made to the IDM. These calls are given a special symbol (in this example, an M in the first column) to separate them from the normal FORTRAN statements. The program, on receiving a transaction type "order", first checks to see if the parts on hand will satisfy the order (if not, the part must be ordered). Then the customer's account is updated with the amount owed from this order; the number of items on hand is changed to reflect the order, and so on.

Embedding the data management commands in the FORTRAN program has several advantages:

(1) It is a simple method of combining the power of the IDM with the flexibility of a high-level, procedural programming language. Note that the example FORTRAN program in Figure 1 does not need to perform any data management tasks. The IDM is taking care of back-up and recovery, access rights, query optimization, concurrency control, etc.

(2) Complex commands involving many different types of data are easy to write using the IDM. IDL makes the database manipulation a clearly separate function from the user customization so the structure of the program is more easily understood.

(3) High-level programs written with embedded IDL are easily maintained. Since the structure of the program is obvious it is more readable and readily understood.

(4) The IDM provides the necessary speed for good user response time; the host system designer can concentrate on writing clear, highly functional end-user interfaces.

(5) The final program can be smaller and require less resources than the full ad-hoc system.

The design of a host system that supports embedded IDL is shown in Figure 2.

Figure 2. Embedded Command System

```
pre-processing (done in host system):

           program              high-level language
              |                 with embedded IDL commands
              |
             \|/
          IDL-parser            produces calls to
              |                 run-time subroutines
              |
             \|/
            compiler            produces runnable programs
                                links to IDL library routines

runtime:

         host
       ------------
       | program  |
       ------------
internal |      /|\
  form   |       | unformatted results
        \|/      |
       ------------
       | IDM      |
       ------------
```

IDL can be embedded in any high-level language. For clarity, let us assume that we are embedding IDL in FORTRAN, as in the Figure 1 example. Then the host system must include a pre-processor that will read a FORTRAN program, find all the IDL calls, and send them to the IDL parser. This is represented in Figure 2 by the function labelled "IDL Parser". The IDL parser produces calls in FORTRAN to the run-time subroutines that interface to the IDM. These calls are substituted in the high-level program for the IDL calls; then the entire program is compiled. At run-time the subroutine calls result in the execution of the IDM commands. The host system designer may wish to have the pre-processor/IDL parser create stored commands for the IDL commands. Stored (defined) commands, as explained in Section 3, are much more efficient.

4.2. Example Query

The following is the tracing of the processing of an entire database command.

4.2.1. Translating the query

Let us assume that a user at a terminal types in the query:

```
range of e is employees
retrieve (e.name, e.salary)
        where e.number > 100
        or    e.salary >= 1000
```

This command displays the employee names and salaries of all those employees who make 1000 or over, or whose employee numbers are over 100.

The function of the translating program is to put the user command into IDM-internal form. This is a standard parsing procedure. The result of parsing the query is a parse tree. A conceptualization of the parse tree for this example is shown in Figure 3.

Figure 3. Parse Tree

```
                       root
                     /      \
                    /        \
               attr1          \
               / \             \
              /   ~ployee       \
         attr2     (name)        \
         /  \                     or
        /    employee            /   \
     end     (salary)           /     \
                               >        >=
                              / \       /  \
                      employee   100 employee \
                       (number)      (salary)  \
                                                1000
```

The left side of the root of the parse tree is the target list; the right is the qualification. Actually, the parse tree will be formed with IDM internal symbols; the complete list of the symbols is included in Appendix B. The Figure 3 tree, with the nodes labelled with their correct IDM symbols, is shown in Figure 4.

Figure 4. Parse Tree - IDM format

```
                           ROOT
                          /    \
                         /      \
                        /        \
                       /          \
                      /            \
                RESDOM              OR
               /     \             /  \
              /       \           /    \
             /       VAR 5       /      \
       RESDOM       (0 name)    GT       GE
       /     \                 / \      /  \
      /       \               /  INT1  /    INT2
     /         \          VAR 7 100   /     1000
TLEND         VAR 7  (0 number )     /
          (0 salary )             VAR 7
                               (0 salary )
```

The symbol RESDOM denotes that the node is a domain (attribute) in the target (result) list. There are variables (the symbol VAR in the above tree) in both the target list and the qualification. These represent references to "range variable" and to "attribute". A range variable ranges over a relation and stands for an instance of a tuple in that relation. These are shown with the lengths of the arguments that must be passed to the IDM, the number of the range variable, and the attribute name. For example,

```
      VAR 5
    (0 name )
```

means that the node denotes a variable. The data in the node is 5 bytes in length. The value of the 5 bytes is 0 for one byte (the variable number is 0) and the ASCII characters n, a, m, and e for the remaining 4 bytes. The constants involved are represented by their types and their values. TLEND denotes the end of the target list. The derivation of the variable number is discussed in the next section. A complete description of trees required for each query type is contained in Section 7. To communicate this tree to the IDM the host must put it in postfix order. This is shown in Figure 5.

Figure 5
Translated Query

symbol	data following	actual octal representation*
TLEND		0001
VAR	7 0 salary	0050 0007 0000 "salary"
RESDOM		0200
VAR	5 0 name	0050 0005 0000 "name"
RESDOM		0200
VAR	7 0 number	0050 0007 0000 "number"
INT1	100	0060 0144
GT		0203
VAR	7 0 salary	0050 0007 0000 "salary"
INT2	1000	0064 0003 0350
GE		0204
OR		0211
ROOT		0264

*Items in quotes ("") are sent to the IDM as character strings.

The left-hand column in Figure 5 is a traversal of the tree, starting at the root and recursively taking the left descendant of the node, then the right descendant, then the node itself. The next column shows additional data for those tokens longer than one byte. To transmit this to the IDM the "symbol" words (in capital letters) must be filled in with the values from Appendix B, as is shown in the "actual octal representation" column. In the remainder of this document, only the symbol will be used; it is assumed that the host system correctly translates the symbol to the octal code specified in Appendix B. The names in quotes, such as "salary", are sent as character strings.

4.2.2. Range Variable Numbers

All accesses to relations are done by specifying range variables. The host must therefore maintain a "range table" to keep track of which variables are associated with which relations. There should be three values in this table: the relation name, the user-defined range variable name, and a unique number chosen by the host. The appropriate entries in the range table are sent to the IDM with each query. The IDM will accept up to 16 variables for each command and they must be numbered anywhere between 0 and 15.

For example, if the user types the following:

```
    range of emp is employee
```

The host should update its range table for the user with the information:

host range table

relation name	user variable name	unique number
employee	emp	0

Whenever a command is sent to the IDM that specifies "emp", the host sends the "range" token, and the information that range variable 0, relation "employee", is being used.

4.2.3. Sending the Command

All commands to the IDM begin with a "command" token (see Section 7 for the list of commands). Following the command token is the "range" if it is used in the command. At the end of the query, the ENDOFCOMMAND symbol must appear. The commands sequence sent to the IDM is summarized in Figure 6.

Figure 6
Commands Sent to the IDM

symbol	following data
RETRIEVE	
RANGE	9 0 "employee"
translated query (from Figure 5)	
ENDOFCOMMAND	

The command is then given to the operating system. When the data is returned to the host, it must format it and display it to the user. If two tuples qualified, Figure 7 shows the results as they are received from the IDM.

Figure 7
Data Sent to Host

symbol	following data
FORMAT	3,CHAR,20,INT4
TUPLE	5,"Jones"
	5000
TUPLE	7,"Johnson"
	300
DONE	NORMAL,2

The data returned by the IDM for this query begins with the format specification. This describes the format of each attribute of the result tuples. In the above example, the FORMAT token is followed by the length of the format data (3 bytes). The second token (CHAR) says the first returned attribute is a character field; it is followed by the number "20", which says the maximum length of the character field is 20 bytes. The third token says that the last (second) attribute of the returned data is a 4–byte integer field. FORMAT tokens are further explained in Section 5.3.

Each returned tuple begins with a tuple token, followed by the tuple in the specified format. In the above example, there are two attributes in the returned tuples. The first, since it is a CHAR attribute, begins with the actual length of the returned data, then the data itself (as a character string). The second is simply a 4–byte integer. The DONE token follows the last tuple. It specifies any error conditions and the number of tuples which satisfied the qualification. Variable length attributes are sent with a preceding byte–count. The host takes the data that is returned, formats it, and displays it for the user.

4.3. The Flow of Control

This section describes the flow of control for getting commands to the IDM and returning results back to the user program. In particular, the role of the operating system is discussed.

The IDM is an unusual device in that it performs its own scheduling and buffering of user commands. As a result, the host operating system has very few functions to perform. This section begins by describing how the IDM would appear to a user program in a typical implementation. Next, the manner in which the operating system interacts with the IDM is discussed. Finally, an example is given which illustrates the flow of control. The interaction between the operating system and the IDM is discussed further in Section 6.

4.3.1. User Program View

In a typical host implementation, programs in the host would issue system calls to their operating system in order to communicate with the IDM. A possible implementation of the system calls is:

```
writeidm (dbin, address, count)

readidm (dbin, address, count)

cancelidm (dbin)
```

"Writeidm" sends a command to the IDM. "Readidm" gets data from the IDM, and "cancelidm" cancels outstanding ———————————————————————

requests to the IDM. These are each discussed further in Section 6.

For "writeidm", the command to be sent is written in a contiguous area of memory in the user program. "Address" specifies the starting address of the command and "count" specifies the number of bytes in the query. For "readidm" the "address" is the address of the data area where the data is to be stored after it is sent by the IDM and count specifies the maximum number of bytes to transfer.

The "dbin" is supplied by the IDM and returned when a database is opened by a user program. It represents the "database instantiation number". It is needed for all three commands.

Note that the user program specifies where to put the results of a query and how much space is available for the results. It is permissible for the amount of space supplied by the user program to be less than the amount of space needed to hold the results of the query. In such a case, another "readidm" will get the next block of data.

The actual size of the result is encoded in the results. As illustrated previously in Figure 7, the results of a command are encoded with control tokens which specify the start of every tuple (TUPLE token) and the end of the results (the DONE token). If the DONE token is not encountered, then the results are larger than the available space and the next block of results can be retrieved by calling "readidm" again.

4.3.2. The Operating System Functions

The operating system is responsible for providing the user's identification, which is either a name of up to 20 characters (huname) or a number (huid), and for uniquely identifying the user program (pid). It must also forward the user program's request to the IDM. If the operating system does not securely provide an huid or huname to the IDM, a password maybe sent for the IDM to validate the user's identity. If either a password or a huname is sent, it should be sent only with the first open database command.

The "pid" is a four–byte number which identifies the instantiation of the user program on that host. It must uniquely identify which process is accessing the IDM on that host but it does not have to be unique across multiple hosts.

Note that the operating system need not care about the "dbin". The operating system either sends user commands to the IDM or **requests** the results of user queries. A request for a result may or may not be granted depending on whether the query has been processed yet.

Operating system requests:

```
write (dbin, pid, huid, address, count)

read (dbin, pid, address, count)
```

The IDM will acknowledge each request from the operating system. The response to a "write" depends on whether the IDM can process the command. If scheduling the commands would cause it to thrash, the IDM will refuse the "write" request. The response to a "read" depends on whether the query has been sufficiently processed.

The operating system has two options with both the "write" and "read" commands: it can wait for the request to be granted, or it can ask to be notified when the request can be granted.

4.4. Example of Flow Control

We will now illustrate the flow of control with an example. Let us assume that in this example the operating system has the IDM notify it when the results are ready. The user program will perform two interactions with the IDM. First, it opens a database and then it runs a query on the database.

PROGRAM	OPERATING SYSTEM	IDM
writeidm(0,addr,cnt)		
	write(dbin,pid,huid,addr,cnt)	
		receives open database command and possibly a password and/or huname
readidm(0,addr,cnt)		
	read(dbin,pid,addr,cnt)	
		transfers results (OS; dbin for database open
(take dbin from result of open database)		
writeidm(dbin,addr,cnt)		
	write(dbin,pid,huid,addr,cnt)	
		receives command (e.g., a retrieve)
readidm(dbin,addr,cnt)		

	read(dbin,pid,addr,cnt)	
		refuses request (results not available)
	suspends program	
		results become available; notifies host
	swaps program in read(dbin,pid,addr,cnt)	
		transfers results
	activate program	
process results		

Before a user program opens a database, there is no database instantiation number (dbin). When dbin=0 is sent with an *open database* command, a new user process, with its own dbin (not = 0), is created in the IDM. The new dbin is returned to the host for use with subsequent commands from the same user program. When a database is opened with a non–zero dbin, the current database is closed and the new one opened using the same dbin.

The example illustrates the two cases which can happen when a request for a result is made. In the first case, the results were available at the time of the request so they were returned immediately. In the second case the results were not available at the time they were requested. The operating system suspended the program and possibly the program was swapped out in the normal operation of the host system. The IDM then informed the host that a previously requested result had become available. The operating system determined which process was waiting for the new results. The process was then loaded into memory (if it has been swapped out) and then a new request to transfer results was issued. The new request is guaranteed to succeed and the program then continues.

The primary goals of the handshaking operations between the IDM and the operating system are to simplify the task of the operating system and to aviod the need to double buffer the results of a query. Avoiding double buffering is valuable in timesharing systems. When a program tries to read the results of a query, it is typically suspended until the results are available. When the IDM notifies the operating system that the query results are available, the operating system can reschedule the user program, move it back into ————————————————

memory if it was swapped out, and then tell the IDM where to put the results. This minimizes the amount of time the user program is "locked" in memory. Alternatively, in different environments it is desirable for the host program to wait for requested results. This option is discussed in Section 6.

4.5. Summary

The IDM can serve as a backend to a variety of hosts, from programmable terminals to mainframes. It is easy for the end user to communicate with the IDM through ad-hoc queries written in a general purpose query language, general purpose programs, or custom applications.

The IDM has been designed to have a minimum impact on the host processor and host operating system. A moderate amount of system support software is necessary to use the IDM. The IDM performs the database management tasks, freeing the system designer to concentrate on the user interface and on solving the user's problems.

5. End-user Interfaces

5.1. Introduction

This section describes different ways to implement end user interfaces. It is not intended to be an exhaustive list, nor does it provide recipes for building them. These are suggestions for designing such programs. Each OEM environment will require different features and different application programs.

5.2. The IDM Command Set

The IDM does not have a "machine language" in the usual sense. To "program" the IDM, a series of commands must be sent to it. An IDM command consists of a byte stream of at least two bytes. The first byte of every command is the "command token." The command token is like an op code and tells the IDM which command is to be performed (see Section 7 for a complete list of commands). The last byte of a command is always the ENDOFCOMMAND token which tells the IDM that the command is complete and that processing can begin. To tell the IDM to begin a transaction, for example, the following bytes would be sent:

"Token"	"Octal"
BEGINXACT	0324
ENDOFCOMMAND	0320

Most commands are modified by arguments. There are five types of arguments to a command.

Immediate
Range declarations
Query tree
Order declarations
Command-options

An immediate argument follows the command token. Such commands are noted in Appendix B. To create a database the command would be:

```
DBCREATE 4 mike
ENDOFCOMMAND
```

The argument "mike" is preceded by its length. The command is terminated by the ENDOFCOMMAND token.

Range, Query tree and Order arguments provide complete specification of retrievals, updates and other commands. Range declarations specify range variables used in the tree and must precede the tree. Range variables are numbered from 0 to 15 and are associated with a relation in the declaration. A range declaration begins with the RANGE token which is followed by the length of the declaration, and then the variable number and the relation name:

```
    RANGE 6 3 parts
```

states that the variable 3 will stand for the relation "parts". The query tree arguments, defined in detail in section 7, are binary trees. The tree is sent as a byte stream using a post order traversal[1]. This allows rebuilding the tree in the IDM without having to send physical pointers. Each node of the tree consists of a token and possibly some associated data. Tokens are divided into three classes: those which have no associated data, those which have fixed length data and those which have variable length data.

Order declarations are used with retrievals to specify an ordering on the data which is returned. An order declaration refers to a domain in the target list of the query tree. The order declarations must come after the query tree. Each declaration consists of an order token which determines if the domain is to be sorted in ascending or descending order followed by the number of the domain.

```
    ORDERA 3
    ORDERD 1
```

declares that the sort order on tuples returned should be first ascending on domain 3, and within identical values of domain 3, it should be descending on domain 1.

Command-options are used to indicate that format information should be returned, that certain error conditions (e.g., overflow) should be ignored, and other miscellaneous parameters of the command. Command-options are sent using the OPTIONS token followed by the count of the number of options and the options themselves. Each command-option has a single byte value except for the tape command option (see section 3.10). One options specifier may be sent per command.

5.3. Results of a Command

The results of a command may just indicate that the command was completed successfully or in error, or the command may return any amount of data. How the results are handled depends on the interface being designed. The general format is described here.

The simplest result is a DONE token followed by eight bytes of data. These are formatted into two 2-byte integers and a 4-byte integer. The 4-byte integer is the count of

[1] Starting at the root, for each sub-tree send the left child, then the right child and then the parent, recursively, i.e., the extreme left leaf is sent first and the root is sent last.

the number of tuples affected by this command. The first two-byte integer is a status word and the second is used for returning single values for certain commands. The DONE may be preceded by error messages. Each error message consists of the ERROR token, followed by the error number (see Appendix C) and possibly some data with its format information. For example:

```
ERROR 6 CHAR 5 parts
```

would mean that the relation "parts" was not found in the current database.

If the command was a retrieval then data may be returned. Each tuple returned is preceded by the TUPLE token. The data has no embedded format information except for the length of variable length fields. If the application program needs the format of the data being returned, the SENDFORMAT command-option should be sent with the command. In that case the format will be sent prior to the data. A format specification starts with the FORMAT token followed by the number of bytes in the specification and then the format tokens and possible lengths. The format tokens will be from the set:

```
BCD
BCDFLT
BINARY
CHAR
INT1
INT2
INT4
FLT4
FLT8
```

The first three types will be followed by the maximum size of an element of that domain. For example if a query is returning tuples with two integer fields and one character field that has a maximum length of 10 bytes, the following results would be returned:

```
FORMAT 4   INT1        INT2   CHAR   10
TUPLE         3      0    2   4      b      o      l      t
TUPLE         6      1   23   3      n      u      t

DONE   0      0      0        0      0      0      0      2
```

The interface program received 4 bytes of format indicating that every tuple would have a one-byte integer followed by a two-byte integer followed by a character field of at most ten bytes. Then two tuples were received, one with a 4-byte character field, "bolt" and one with a three-byte character field, "nut". Then the DONE was received indicating zero status and 2 tuples returned. Since the length of the format string is limited to 255 bytes it may be necessary to send two strings. Each will be preceded by the FORMAT token and length. A format specifier will never be broken between the token and length.

If the target list of a retrieve is sent with RESATTR nodes rather than RESDOM nodes, the "send names" command-option may be sent with the command, so that the names of the result attributes are retrieved immediately preceding the format information (if any). Each name is sent like a character constant: the CHAR token, followed by a 1-byte length, followed by the name. The "send names" command-option has no effect if the command it follows is not a retrieve command.

5.4. General Purpose Query Language

A general purpose query language interface allows users to pose arbitrary queries to the IDM and to get the results displayed. The interface program may also provide editing, file handling, and display formatting capabilities, but the inclusion and implementation of such features is mostly independent of the main function of the interface, i.e. parsing the query language and translating it into IDM commands. For the purposes of this discussion we will assume that IDL is being used as the query language. The full definition of the IDL command set is given in section 7. This section provides suggestions on how to build a parser for IDL. It is assumed that the reader is familiar with the basics of compiler construction. The use of a parser generator is recommended as the syntax is large but regular. The language can be recognized with an LALR(1) parser such as generated by YACC[2].

5.4.1. Lexical Analysis

Tokens in IDL are separated by blanks or operators. Blank, tab and end-of-line should be ignored except for delimiting tokens. The language has the following classes of tokens:

Keywords	- reserved and may not appear in other contexts
Names	- alphanumeric sequences
Strings	- surrounded by double quotes: ´"´

[2] Johnson, S. C., "YACC-yet another compiler compiler", CSTR 32 Bell Laboratories, Murray Hill, N.J. Also in "UNIX Programmers Manual, Seventh Edition", Vol. 2b. See also "LALR(1) Parser Constructor to Translate Computer Languages" by David M. Stern, DECUS program library #11-312, which runs under RSX-11 on DEC. PDP-11s, or "LR, Automatic Parser Generator and LR(1) Parser", by Charles Wetherell and Alfred Shannon, Preprint UCRL-82926, LLL Po box 808, Livermore, Ca. which runs in ANSI fortran. For a discussion of such generators see chapter 6 of "Principles of Compiler Design" by Aho and Ullman.

Operators	– one or two non–alphanumeric characters
Numbers	– one or more digits with optional decimal point
BCD	– one or more digits beginning with "#" with optional FORTRAN–style scientific notation
Parameters	– alphanumeric sequence beginning with "$"

During the parsing process it should not be necessary to add to the set of tokens. The only possible new token type introduced is the "range variable" and it is easier to recognize one as a name and then give an error if the name has not been declared in a "range" statement. With the exception of "range" statements, tokens need only be preserved for one statement and the storage for them reused.

5.4.2. Syntax

The syntax of IDL is keyword driven. All statements begin with one or two keywords and have a set syntax. The complete syntax is defined in Section 7. The syntax of most commands is broken into two parts: the *target list* and the *qualification*. The target list is usually a list of names, as in the *create index* command or a list of "name = expression" pairs as in the *retrieve* command. The target list of the *create* command is an exception in that the expression in the target list must be a type specifier.

Qualifications come in three varieties. One is a boolean expression tree which is used in query commands such as *retrieve* and *replace*. The second is used in maintenance commands such as *create* and usually begins with the key word *with*. This type is a comma separated list of specifiers of the form:

```
<with–node–option> = <constant>
```

The third is a list of names and is used in the *permit*, *deny* and *associate* commands.

5.4.3. Semantics

The main job of the semantic routines of the parser is to build up a query tree creating nodes of the proper type for each significant token. When the end of the statement is detected the command must be sent to the IDM. Typically, the parser will request that format information be returned, using a command–options argument, for a retrieve.

5.4.3.1. Target List Semantics

The target list specifies those domains which are retrieved or affected by the IDM command. Where the order is important, retrieval for example, the first domain is the left child of the root of the tree and the last is the parent of the target list end node (TLEND). In a *retrieve* command the names of the result domains are not significant ——————————————

and need not be sent to the IDM. In this case a RESDOM node may be sent rather than a RESATTR node, which requires a length. For display purposes on the front–end, the domain names are either specified by the user or can be defaulted to the attribute name if there is no expression involved. When defining a stored retrieval the names can be stored in the IDM using the RESATTR nodes and can be requested when the query is executed.

It is convenient for users to type:

```
retrieve (x.all) where x.name = "John"
```

The *all* stands for all domains in the relation associated with the variable "x". It only makes sense to use *all* in the target list of a retrieve.

The "order by" clause in IDL may reference result domain names in the target list, or expressions not in the target list. In the former case the proper expression must be found and an order argument sent with the command referring to that node in the target list. In the latter case a new ORDERDOM node must be added to the target list. Note that an ORDERDOM expression is only used for ordering the results and will not be returned by the IDM.

When data is returned on a *retrieve* it will be necessary to format and display the data. While parsing the target list an array of records can be used to store the domain names. The IDM can be requested to return the types of the domains being retrieved prior to sending any data. This information would also be put into these records and used when the data is returned for formatting.

5.4.3.2. Qualification Semantics

The qualification of an IDL command appears as the right sub–tree of the ROOT node. The minimum qualification is just a qualification end node (QLEND). If the qualification specifies with–node–options, a "with clause" will contain a list of options or modifications to the command. The parser must translate the option names into the one–byte ——————

option specifier and put it in the tree following the WITH node. The value of the option is the left child of the node while the next WITH node must be the right child. The boolean expression tree form represents the restrictions and joins in the command. The tree contains AND and OR nodes with relational clauses as children. In the permit command the user and group names go in the qualification as character constants which are the left children of QUALDOM nodes. Section 7 contains the full specification of all qualification trees.

5.4.3.3. Expression Semantics

All operator precedence is expressed in the query tree. IDL defines multiplication and division to have higher precedence than addition and subtraction, but this is transparent to the IDM, which processes expressions as they appear in the query tree.

5.4.4. Stored Commands

When executing a stored command no query tree is sent. This is to make transmitting and executing stored commands as efficient as possible. The EXEC token is sent followed by the name of the command. Parameters are sent as a stream of tokens. The parameter names may be specified, or the parameter values may be sent in order. The order of the parameters is the alphabetic order of their names except that totally numeric names are put in ascending numeric order. If the parameters are sent by name the name is preceded by a PARAM token and its length. The parameters are sent, preceded by a "type" token (INT1, INT2, INT4, BCD, BCDFLT, CHAR, FLT4, or FLT8) and possibly a length[3]. If the user types:

```
    execute get 5, "john"
```

the parser would send

```
    EXEC 3 get
    INT1 5
    CHAR 4 john
    ENDOFCOMMAND
```

The user could type, equivalently:

```
    execute get with name = "john", age = 5
```

and the parser would send:

[3] A trick to implement this form is to build a list of parameters as a tree which, when transmitted in post order, will send the parameters in the right order.

```
    EXEC 3 get
    PARAM 4 name
    CHAR 4 john
    PARAM 3 age
    INT1 5
    ENDOFCOMMAND
```

5.5. Embedded Query Language

The embedding of IDL in a general purpose programming language can take different forms. A minimal implementation will be discussed here, in which all queries are formulated at compile time and only parameters are provided at run time. It is possible to construct an embedding which allows queries to be constructed at run time as well. This would require invoking parsing routines at run time, i.e., combining what is discussed here with the features of a general query interface.

To embed the IDL in a programming language it is necessary to write a preprocessor which recognizes IDL statements and translates them into the host language[4]. The program then can be compiled normally, including some run time support routines. The simplest way for the preprocessor to recognize the IDL statements is to begin each line which contains one with a special symbol, dollar sign ´$´, for example (an "M" was used in Section 4). In this way the preprocessor need not parse the host language. This also makes it easy to modify the preprocessor to handle other host languages. Once the IDL statements are recognized they are parsed by the preprocessor and checked for syntax errors. Each IDM command is then sent to the IDM as a stored command using the define program command. The IDM will return a 4-byte number by which the command can be run. The source of the IDL statement is then deleted from the program and replaced by code which invokes the stored command.

The above works if the program only runs updates without any parameters. To process retrieves and handle parameterized commands it is necessary to parse the part of the host language that is involved with variable declarations. The embedding can require that variables used in IDL statements be declared on lines beginning with ´$´. When such lines are parsed the name and type of the variable is saved and the special symbol is deleted. When such variables are recognized in an IDL statement in the place of a constant in the grammar, the preprocesser replaces them with parameters in the stored command and generates code to send

[4] It is possible to put this function directly into the host language compiler, but this is often difficult unless you write your own compiler.

the run time value of the variable when the stored command is executed from the program.

To make retrievals useful in a programming language, it must be possible to get values out of the database and into programming variables. Since more than one value will generally be returned, the embedding should allow for the processing of the values individually. This can be done by making the retrieve command a looping construct in the host language and by putting programming variables in the target list. For example, IDL embedded in Pascal:

```
    $       var i, j : integer;

    $       range of e is employee
    $       retrieve (i = e.salary) where e.number = j
    $       begin
                    Pascal code
                      ...
    $       end
```

would retrieve all salaries into the variable ´i´, one at a time by generating the following Pascal code:

```
            var i, j : integer;
            idmparam : array[1..MAXPARM] of record
                    case t : idmtype  of
                            i2: (int2 : -32768..32767);
                            i4: (int : integer);
                            character : (chr : string_type)
                    end
            end;

            idmparam[1].int := j;
            idmparam[1].t := i4;
            idmexecute(2048, 1, idmparam);
            while idmnewtup do
            begin
                    i := idmint4;
                    Pascal code
                      ...
            end
```

The routines "idmexecute", "idmnewtup" and "idmint" must be included by the preprocessor. "Idmexecute" takes as arguments the query number to execute and the count of parameters to send with it and an array of parameters. "Idmnewtup" returns "true" if there is another tuple and "false" otherwise. "Idmint4" returns the value returned in the tuple. When there are no more values the program will fall out of the loop and continue with the rest of the code. While the programmer sees one tuple at a time, the underlying code should buffer a convenient amount of data from the IDM.

It is sometimes necessary to terminate the retrieval before all the tuples have been returned. The embedding should define a way to exit the implied loop of the retrieve. In Pascal this could be done by defining a procedure which the application programmer could call in the loop which would clear any buffered input and, if necessary, send a "cancel" command to the IDM. "Idmnewtup" would then get a DONE and return "false". When the DONE is returned the data should be placed in a global record so that the programmer can look at the values returned.

The preprocessor needs to insure that the values returned to the program are the same type as the variable to which they will be assigned. This can be done by adding an explicit conversion to expressions in the target list. In the example above the preprocessor would generate a target list which looked like:

```
                 .
                .
               .
          RESDOM
          /   \
     TLEND  CNVTI4
               \
               VAR
             (0 number)
```

If "e.number" was already a four-byte integer the IDM optimizations would delete the CNVTI4.

It is possible to issue several commands at once from a program. This requires, however, that the database be opened more than once at run time since each concurrent command needs a separate database instantiation number. Depending on how restrictive the embedding is, different implementations can be used. A simple implementation could prevent all concurrent accesses. This requires only opening the database once. The programmer could be required to explicitly open the database again if the program had concurrent access. A full implementation would have to detect the nesting at run time and open the database again. Alternatively the embedding could require that any concurrent accesses be nested statically. For example:

```
        ....
        { not good style }
$       retrieve (i = e.number)
$       begin
                if i = 10 then
$                       delete e where i = 10
                else
                        update(i);
$       end
```

Under the latter implementation the delete would be legal but the procedure "update" could not do any database calls.

5.5.1. Protection Within Programs

Like all stored commands, those coming from embedded commands can have access rights granted to them. Protection is done by the program name, so all commands stored under the same program name have the same protection. If the application needs different protection on different commands then a program statement can be used to change the name associated with some commands. For example, if a personnel application is being written which interactively retrieves information about employees, the user may request information which is protected. The host program does not need to look up the access rights of the person running the program. Instead different commands within the program are given different protections.

```
    $       program prog_1
    $       permit execute of prog_1 to all

            {a request for name}
    $       retrieve (name = e.name) where e.id = ident
    $       begin
                    { print name }
    $       end
            .....
            {more code...
                    the user asks to look at a salary}
    $       program prog_2
    $       permit execute of prog_2 to managers

    $       retrieve (sal = e.salary) where e.name = name
    $       ....
```

The permit commands are issued at preprocessing time rather than at run time. A null define program command may be issued to the IDM to establish the program name prior to storing any commands. If the program already exists this will cause an error. The protection statements may then be issued.

5.6. Subroutine Calls

Using a subroutine call interface is similar to embedding IDL, except that the preprocessing must be done by the applications programmer. A library of routines is created which communicates with the IDM. Routines similar to those in the previous section would be provided for the programmer to call directly. The stored commands would be stored by the programmer prior to running the program.

5.7. File System Support

The IDM provides a random access file system. The purposes of the system include:

1) A convenient way to store non-database information such as executable programs for intelligent terminal based systems.
2) The dump and load commands can store and retrieve information from an IDM file instead of from the host.
3) A complete file system can be based on IDM files instead of a separate host file system. For example, a hierarchical file system could be built with IDM relations used for file directories.

When used as a simple file system for storing programs, a host would need to provide a program which could copy a file from the host to the IDM and back. For example:

```
    writefile idm_filename host_filename
```

might be a program running on the host which verifies the existence of "host_filename", creates "idm_filename" nd then copies the file contents to the IDM. To load the file into a programmable terminal, a ROM- or PROM-based loader in the terminal would need to observe the IDM communication conventions to talk to the IDM to request a file.

5.8. Dump and Load Support

The IDM provides three types of "dump" and "load" commands:

1. copy in, copy out
2. dump, load
3. dump transaction, rollforward

Copy in/out is useful for initially loading a database and for bulk moving of data between the IDM and a host. The data format used by the copy commands is independent of the device and IDM so it can be interpreted by any host computer. Dump and load are used to quickly backup a database. Dump transact in conjunction with rollforward is a fast way to save and restore recent changes to the database.

5.8.1. Copy in, Copy out

Copy in and copy out provide a "bulk load" capability between the IDM and a host. A host program is needed to accept or send data and do any special formatting. For example, a user may have a file or tape on the host system in some special format. A "bulkload" program on the host could accept a format description of the input data and the name of a relation on the IDM to receive the data. The "bulkload" program would request from the IDM the format of the attributes in the relation, convert the input data to the proper data type, one domain at a time, and send the data to the IDM using the copy in command. Correspondingly, a user may want a relation on the IDM copied to the host and formatted in some particular way. The "bulkload" program could request the data from the IDM using either the copy out command or a retrieve command.

Copy in and copy out can also be used to transfer data between IDMs or different databases on the same IDM. When relations are copied using the copy out command, the names and formats of the relations are sent along with the tuples of the relation. The information could be sent back to an IDM using the copy in command.

5.8.2. Dump, Load, and Dump Transact

These commands are used for backup purposes. Whole databases can be dumped and reloaded. The information from a dump can be stored:

1. on the host
2. in an IDM file on another database
3. on IDM tape

The general strategy for backup is to occasionally dump the database and to frequently dump the transaction log. If a database must be restored from a dump, the most recent complete copy is loaded back in, and then each transaction dump must be loaded and rolled forward. Note that it is not sufficient to load a database and then apply only the most recent transaction dump. (In fact, the IDM detects this case and issues an error.)

Host support for dump and load depends on whether the data is being sent back to the host or kept in the IDM. If all data is dumped to IDM files, then support for dump and load can be put in the ad-hoc query language; otherwise, a host program is needed to send the dump/load command and store/retrieve the data. The main requirement of the host program is to supply the data in the same order as it was received.

5.9. Report Generators

A report generator provides a facility for formatting data from the IDM in a flexible way. A report generator could be combined with an interactive query facility or be a separate program. Using a report generator the user can specify what data to display, where it goes on the page, what order the data should be in, what to print on the top or bottom of each page, and many other things. The IDM can be used to project, restrict and sort the data before the generator processes it. The IDM can also be used to store and catalogue report descriptions. The relation "reports" could be used to store report names and queries. A "desc" relation can be used to store descriptions of the reports.

```
    reports(name, number, query, description)
    desc(number, record, type, line, start,
                    length, just, value)
```

A report would consist of several records with the same "number". The "type" would specify that this was part of a page heading, column heading, data field, or other type of specification. The "line" specifies the line on the page to which this record refers. "Start" tells where to start this field, "length" is the maximum length of the field and "just" tells how to justify the the actual string in the field (left, right, center). The value would be the actual string to print or the attribute or the relation being reported. When a report is defined, the generator could store a command on the IDM which would retrieve the data needed to do the report. The stored command number would be saved along with the report name, number and description (or title) in the "reports" relation. The description is translated from the report description language and entered in the "desc" relation.

When the report is to be generated the generator reads the "reports" and "desc" relations which tell how to format the report. The stored command is run (possibly with parameters from the user) and the generator receives and formats the data.

6. Communicating with the IDM

The following section describes the set of commands and protocol used for communication between the host and the IDM channel. The first subsections describe the general interface, independent of serial or parallel communication. A description of the commands and protocol is given and a model host interface is described to illustrate the communication. The model interface described assumes that the front end is a general purpose computer. In the stand alone system where the "host" is a programmable terminal, the interface is simplified and some of the information concerning the interface can be ignored. The last two subsections describe the details of the serial and parallel interface.

The communication between the host and the IDM channel can be described as the following set of interfaces:

```
   host          host                 host              IDM
   user  <---> operating <---> communication <---> channel
 program        system               device
```

The characteristics of the above host modules will, of course, vary greatly among different hosts. However, a model interface will illustrate the important issues of the communication and give insight into particular implementations.

The interface between the user program and the host operating system has been described in earlier sections. For the discussion here, it is sufficient to recognize that the user program wishes to write to and read from the IDM. The remaining two interfaces, i.e., the interface between the operating system and the host communication device, and between the device and the IDM channel are described in this section. The "host communication device" is a term given to the handler program and the hardware of the I/O device that is directly communicating with the IDM.

6.1. IDM Channel Communication Commands

The IDM channel supports a set of commands for communication between host and channel. These commands are called communication commands. The subset that is issued by the host consists primarily of commands to write to the IDM, i.e., send IDM commands, and to read from the IDM, i.e., read results. In addition, the subset includes communication commands to cancel IDM requests.

6.1.1. Host-to-IDM Channel Communication Commands

The IDM supports a protocol for sending IDM commands and requesting results which allows for flexibility in host operating systems or host interface programs. The operations of writing to the IDM and reading from the IDM are each supported by three separate communication commands. The host may issue any one of these commands to read from or write to the IDM.

The different READ and WRITE communication commands indicate the action that the host will take if the request cannot be immediately granted by the IDM. The host operating system may choose to: wait until the request can be granted, not wait until the request can be granted and send the request again at a later time, or not wait until the request can be granted but ask that the IDM "call" the host when the host should re-issue the request. For example, the general purpose computer host running a timesharing operating system might choose the third method of reading and writing. This would allow for efficient operation in the host. A host terminal program in a stand alone environment, however, would most likely choose to wait until a request can be granted. The second alternative method for reading and writing would be used by a host which does not wish to wait, but also does not wish to support calls from the IDM indicating that the request should be re-issued. In this case, the host re-issues the request after a host-dependent time period. Host characteristics will determine the appropriate READ and WRITE communication commands.

The operating system may use any one of the three WRITE communication commands described below to write to the IDM. It must also supply the following additional information (described in Section 4).

dbin	- data base instantiation number
pid	- process id
huid	- host user id
count	- number of bytes of data

The IDM responds YES to the WRITE command if it is currently able to accept the WRITE request and NO if it is not. If the response is YES, the IDM sends a count to the host. This count is less than or equal to the "count" sent by the host and indicates the number of bytes that may be written. The host then sends the data to the IDM. If the response is NO, the IDM sends an error code indicating why the request cannot be granted. Error conditions are discussed below.

WRITENW   WRITE command, No Wait
This is a request to write to the IDM. If space is available to write, the response from the IDM is YES, followed by a "~ount". The host will then

send "count" bytes. If space is not available, or an error has occurred, the response is NO and an error code is sent. If space was not available, the host may try to write at a later time.

WRITEW WRITE command, Wait
This is a request to write to the IDM. If space is not available to write, the host will wait. Unless an error has occurred, the response from the IDM is always YES, followed by a "count", and occurs when space becomes available. The host will then send "count" bytes. If an error has occurred, the response is NO, followed by an error code.

WRITECALL WRITE command, Call
This is a request to write to the IDM. If space is available to write, the response from the IDM is YES, followed by a "count". The host will then send "count" bytes. If space is not available or an error has occurred, the response is NO and an error code is sent. If space was not available, the host will not wait but requests that it be called when space is available. The IDM later calls the host (WAVAIL) indicating that space is available for writing.

The NO response to a WRITE request is accompanied by a code indicating one of the following situations:

1) insufficient space available for writing
This is a temporary condition and is handled as described in the communication commands above.

2) illegal communication command or negative count
This is a host system error.

3) too many commands on this "dbin"
This is a host user or host system error. It indicates that too many IDM commands have been issued on this "dbin". This may be caused by a user program which continually sends WRITE requests to the IDM. This error should result in a cancellation of the user program.

4) not accepting write requests on this "dbin"
This is a temporary condition and can occur during a physical or logical load. The host program is sending data faster than it can be loaded by the IDM. Eventually, the host program will be allowed to continue. If a WRITECALL was used, the IDM will notify the host (WCONTINUE) when this write may continue. Otherwise, the host program must

try again later.
Additional error conditions are described in section 6.3.4.

All results from the IDM must be requested by the host. This is done using one of the three READ communication commands described below. The operating system must provide the following additional information.

dbin	- data base instantiation number
pid	- process id
huid	- host user id
count	- maximum number of bytes read

The IDM responds YES if results are available. The IDM also sends a count indicating the actual number of bytes to read. This count is less than or equal to the "count" sent by the host. The IDM then sends the data. If the results are not available or an error has occurred, the response from the IDM is NO and an error code indicating why the request cannot be granted is sent.

READNW READ results, No Wait
This is a request to read results from the IDM. If the results are available, the IDM responds with YES and returns a "count". The IDM then sends "count" bytes of data. If the results are not yet available or an error has occurred, the response is NO and an error code is sent. If the results were not available, the host may request the results at a later time.

READW READ results, Wait
This is a request to read results from the IDM. If the results are not available, the host will wait. When the results are available, the IDM responds with YES and also returns a "count" indicating the number of bytes to be returned. Unless an error has occurred, the response from the IDM is always YES, and occurs when the results become available. After the "count", the IDM sends "count" bytes of data. If an error has occurred, the NO response is sent, followed by an error code.

READCALL READ results, Call
This is a request to read results from the IDM. If the results are available, the IDM responds with YES and also returns a "count" indicating the number of bytes to be returned. The IDM then sends "count" bytes of data. If the results were not yet available or an error has occurred, the response is NO, followed by an error code. If the results are not available, the host will not wait

but requests that it be called when results are available. The IDM later calls the host indicating that these results are now available.

The NO response to a READ request is accompanied by a code indicating one of the following conditions:

1) results are not yet available
   This is a temporary condition and indicates that results for this "dbin" are not available. Eventually, either results will be available (RESULTS) or an error condition will by reported. This condition is handled as described in the communication commands above.

2) illegal communication command or negative count
   This is a host system error.

3) illegal "dbin"
   This is a host user or host system error.

4) insufficient space to accept READCALL requests
   This is a temporary condition. The IDM will notify the host (RAVAIL) when READCALLs will be accepted.

Additional error conditions are described in section 6.3.4.

The remaining communication commands from the host to the IDM are:

WRCALL

This command is a WRITECALL request to the IDM with an implicit READCALL request to read the results. It is provided to increase the efficiency of the communication required to write an IDM command and read the results. Typically, the host program will write an IDM command and immediately read the results. This command allows the host to send both requests with one communication command. The response from the IDM is the response that would be received if the WRITECALL request were issued. The IDM then will notify the host when the results are available.

IDENTIFY

This command is used as the first communication with the IDM or to re-establish communication. The host must send the "hostid" (hid) to the IDM at this time. Additionally, the host characteristics must be sent to the IDM. These characteristics include: binary integer representation, character representation, BCD integer representation, and error detection options. The exact format for specifying this information is given in the next section. Unless an error has occurred, the IDM responds with YES followed by its "idmid". This is a unique IDM identification number to the host.

CANCEL

This command is used to cancel the current IDM command issued by a host program. The "dbin" and "pid" of the program that issued the command must be provided. The current command is aborted and results are flushed. In order to ensure that all buffers containing data for this dbin are flushed within the IDM and within the host system, the CANCEL command is followed by an acknowledgement which must be read by the host. The host issues a CANCEL command to the IDM, and the IDM immediately responds with a YES response (unless an error has occurred). Then, the host must issue a read command on the same dbin, and the IDM will respond to it with a NO response, accompanied by a special error code indicating the CANCEL acknowledge. This protocol is illustrated in section 6.3.5.

CANCELP

This command is used to cancel all pending IDM commands issued by a particular program on the host. The host must supply the "pid" of the program. All pending transactions will be aborted and all results will be flushed. This command is used when a program on the host has abnormally terminated. The IDM sends a YES response, or a NO response with an error code.

CANCELH

This command is used to cancel all pending IDM commands issued by the host. All pending transactions will be aborted and all results will be flushed. This command is used following a host system failure. When the host returns on-line, it must issue the IDENTIFY command and then issue the CANCELH command. The IDM sends a YES response, or a NO response with an error code. In the serial interface, the host is identified by the serial port over which it communicates. In the parallel interface, the host is identified by the address it uses on the interface bus.

HELLO

This command is used primarily to indicate to the

IDM that the host is still on-line. The host must communicate with the IDM at least every "TIMEOUT" minutes. "TIMEOUT" is a host defined time period, which is specified as a host characteristic in the IDENTIFY command. If the host is not sending any communications to the IDM, then it must send a HELLO command at least every "TIMEOUT" minutes. If "TIMEOUT" minutes pass and no communication is received by the IDM, the host is assumed to be down. The host can optionally turn off this facility. The HELLO command can also be used by the host to determine whether the IDM is on-line. The IDM sends a YES response, or a NO response with an error code.

6.1.2. IDM Channel-to-Host Communication Commands

The set of communication commands which the IDM uses to communicate with the host is described below. If the host chooses not to use READCALL and WRITECALL, the host will not receive these asynchronous communication commands from the IDM; in that case, any communication from the IDM would be an immediate response to commands sent by the host.

RESULTS - Requested Results Available
This command informs the host that requested results are available. This command is only issued as a response to a READCALL or WRCALL from the host. The IDM supplies the "pid" and "dbin" of the requester. The host must then send the corresponding READCALL request to read the results.

RAVAIL - Read Available
This command informs the host that the IDM now can accept READCALL or WRCALL requests. It is only issued following a READCALL request in which the IDM responded NO and indicated that READCALL requests could not be accepted.

WCONTINUE - Write Continue
This command informs the host that a particular write request may now continue. It is only issued following a WRITECALL or WRCALL in which the IDM responded that it could not accept a write on this "dbin". The IDM supplies the "dbin" and "pid" of the request and the host may then send the write request.

WAVAIL - Write Available

This command informs the host that the IDM is now available for writing. The command is only issued following a WRITECALL or WRCALL request in which the IDM responded that there was no space available for writing. This command is not associated with any particular "dbin" or "pid". The host then may send a WRITE request.

6.2. Operating System to Host Device

In order to send communication commands to the IDM and receive them from the IDM, the host must define an interface between the host operating system (or host program) and the host communication device. In our model interface, the operating system will receive a program read or write request, append the necessary parameter information, and deliver the request to the host communication device. The operating system in our example uses READCALL and WRITECALL requests. The device then performs the communication with the IDM. Also, the device is responsible for receiving communication commands from the IDM and delivering the information to the operating system. The details of the device-to-IDM communication are given in the next section.

To perform the communication between the operating system and the host device, the model host device contains two independent sets of registers: the SEND/RECEIVE registers and the ASYNC RECEIVE registers. The SEND/RECEIVE registers are used to issue communication commands from the host and to receive the responses. The ASYNC RECEIVE registers are used to receive communication commands issued by the IDM. Each set of registers consists of a status/command register and registers for the parameters required by the communication command. Alternatively, these parameters could be kept in a parameter block and a single register containing the address of the block could accompany the status register. This is an implementation detail determined by the characteristics of the host machine and the former design has been chosen for this example. The registers are defined as follows:

SEND/RECEIVE Registers

```
SR_STATUS - host word size - status bits for sending
                             communication command and
                             receiving response
        The status bits are:
          communication command - 8 bits
          READY bit - device ready
          INTRENABLE bit - interrupt enable bit
          GO bit - transfer information to IDM
          DONE bit - request completed
          RETURN bit - response from the IDM (YES / NO)
          RECERROR bit - recoverable error occurred
          NRECERROR bit - non-recoverable error occurred


S_DBIN - 16 bits - data base instantiation number
S_PID  - 32 bits - process id
S_HUID  - 32 bits - host user id
S_ADDR - host word size - address of bytes to send or
                          address of location for results
S_COUNT  - host word size - number of bytes to send or
                            receive
R_COUNT  - host word size - number of bytes actually
                            returned
R_ERROR  - 16 bits - error code if RETURN bit is NO
```

ASYNC RECEIVE Registers

```
AR_STATUS - host word size - status bits for receiving a
                             communication command
                             from the IDM
          The status bits are:
              communication command - 4 bits
              ATTN bit - attention bit
              INTRENABLE bit - interrupt enable

AR_DBIN - 16 bits - data base instantiation
                    number of requester
AR_PID  - 32 bits - process id of requester
```

The SEND/RECEIVE registers are used as follows. The READY bit in the SR_STATUS register indicates that the device is ready to accept a communication command from the operating system. At this time, if the operating system has a command to send, the device handler loads the device SEND registers with the request information. The device handler then sets the GO bit. The GO bit initiates the transfer of the information.

The device communicates the information to the IDM, receives a response, and takes the appropriate action, such as reading bytes or sending bytes. On a WRITE request, the device uses the address in the S_ADDR register and the count in the S_COUNT register to send the data to the IDM. On a READ request, the device uses the S_ADDR register to store the data returned from the IDM. After this is completed, the DONE bit is set indicating the completion of the request. At this time, the RETURN bit indicates the response from the IDM (YES/NO), the RECERROR and NRECERROR bits indicate whether a recoverable or non-recoverable transmission error occurred, and the R_ERROR register contains an error code if the RETURN bit is NO. In the case of a READ request, the R_COUNT register contains the number of bytes returned.

The ASYNC RECEIVE registers are used to accept the communication commands from the IDM. When a communication command is received by the device, it loads the ASYNC RECEIVE registers with the information received from the IDM and sets the ATTN bit to interrupt and alert the operating system. The operating system then reschedules the requests which were waiting on this call from the IDM.

6.2.1. An Example

We can now describe the flow of control from the user program to the host communication device for our model interface. Consider a user program that wishes to send an IDM command to the IDM. The operating system receives this request:

```
writeIDM(dbin,addr,cnt)
```

This indicates the "dbin" of this user program request and the address and count of the command to be sent. The operating system must now communicate this request to the host device, supplying the "pid" and "huid" of the requesting program. The operating system in our example always chooses to use WRITECALL or READCALL. Therefore, if space is not available or results are not available, the operating system does not wait on the particular request and expects to be called by the IDM when the request can be completed.

The operating system places the new user request on a queue of requests, adding the "pid" and "huid" of the user program. It then checks the communication device. If the device is inactive, the operating system must invoke the handler routine to process this request. Otherwise, the request is handled at interrupt time when it reaches the head of the queue. The handler places the request information in the SEND registers, loads the SR_STATUS register with the WRITECALL command and sets the GO bit.

The device then handles the communication with the IDM. If the WRITE request was granted, the device sends the bytes to the IDM. When the communication is completed, the RESULT register is set to indicate a YES response, the ERROR registers are set appropriately, and finally the DONE bit is set. The operating system can now resume the requesting program and the device handler processes the next request.

If the response from the IDM had been NO, the operating system could then suspend the requesting program, and the device handler would process the next request. At a later time, the IDM would send a WAVAIL communication command to the host. The device would receive the request and inform the operating system. The operating system would re-issue the WRITE request by placing the request in the queue once again.

A user program requesting to READ results,

```
readIDM(dbin,addr,cnt)
```

would be processed in an analogous manner.

To summarize, the operating system handles requests to write or read the IDM as follows:

```
                    OPERATING SYSTEM

0. receive readIDM or writeIDM system call
1. add the "pid" and "uid" of the user program
   to the request
2. place the request on the queue
3. check the READY bit of the device
        if the device is idle:
                invoke the handler routine to process
                this request
4. done
```

When the device completes a request, it sets the DONE bit and interrupts. The interrupt handler proceeds as follows:

SEND/RECEIVE INTERRUPT HANDLER

```
0. disable interrupts
1. check the RETURN bit to determine the IDM response
       if response is NO:
               record error information from the
                R_ERROR register for user program
                waiting on this request
2. wakeup user program waiting on this request
3. remove request from the queue
4. check queue for next request
       if not empty:
               load the SEND registers with request
                information
               set the device GO bit
5. enable interrupts
6. done
```

The operating system would invoke the SEND/RECEIVE INTERRUPT HANDLER at step 4.

When an asynchronous communication command arrives from the IDM, the device sets the ATTN bit and interrupts. The interrupt handler which is invoked by this interrupt proceeds as follows:

ASYNC RECEIVE INTERRUPT HANDLER

```
0. disable interrupts
1. read the AR_STATUS register for the communication
    command sent
2. wakeup user program waiting on this command
3. place request back on the queue
4. enable interrupts
5. done
```

6.3. Host Device to IDM Channel

The function of the host communication device is to secure the sending and receiving of communication commands and data to and from the IDM. The communication commands and appropriate responses are used to define a protocol for the communication between the host device and the IDM channel. All communication between host and IDM is done using packets. A packet, which is the unit of communication, is a series of bytes that is always acknowledged by the receiver. Two kinds of packets are used: communication packets and data packets. Communication packets are fixed length and contain communication information. Data packets are variable length and contain actual data.

6.3.1. Communication Packets and Data Packets

A communication packet from the host to the IDM is a 16-byte packet. It contains a communication command and the associated parameters required by the IDM. The first byte of every communication packet is a special HEADER byte, which is followed by a communication command. The last two bytes of every communication packet are used for a "check", which is an error detection quantity. Packet error detection is described in the next section. The octal codes for the communication commands and the HEADER byte are given below.

Communication Command Codes
(OCTAL)

Host to IDM		IDM to Host	
IDENTIFY	000		
WRITENW	001	WAVAIL	001
WRITEW	002	RAVAIL	002
WRITECALL	003	WCONTINUE	003
READNW	004	RESULTS	006
READW	005		
READCALL	006		
CANCEL	007		
CANCELP	010		
CANCELH	011		
HELLO	012		
WRCALL	013		

HEADER byte -- OCTAL 347

With the exception of the IDENTIFY packet, a communication packet from the host to the IDM consists of:

Host to IDM
Communication
Packet

HEADER	1 byte
command	1
dbin	2
pid	4
huid	4
count	2
check	2

The parameters that are supplied in the communication packet, i.e. the "dbin", "pid", "huid", "count", and "check" are expected to be sent in the 2-byte or 4-byte integer format of the host. The IDM channel will convert the data types to its own internal data type.

The IDM will ignore any parameter in a communication packet which does not pertain to the particular communication command. For example, CANCEL, CANCELP, CANCELH, and HELLO commands are all sent in a 16-byte communication packet with the appropriate parameters included.

A communication packet from the IDM to the host is a 10-byte packet containing an "IDM to host" communication command. This packet is used to send the RESULTS, WCONTINUE, WAVAIL or RAVAIL commands. The IDM-to-Host communication packet contains:

```
IDM to Host
Communication
  Packet

-------------
|  HEADER   |  1 byte
-------------
|  command  |  1
-------------
|   dbin    |  2
-------------
|   pid     |  4
|           |
-------------
|  check    |  2
-------------
```

The RESULTS and WCONTINUE communication commands make use of the "dbin" and "pid" in this communication packet, while the other commands do not.

A data packet from the host or from the IDM is variable length and contains a maximum of 2048 bytes. Every data packet is followed by a 2-byte check. Therefore, the maximum data packet is 2K + 2 bytes. The dbp and a serial channel communicate by default in packets of 256 bytes; and the dbp and a parallel channel communicate by default in packets of 2 Kbytes. In case of a parallel channel, it is possible to override this default. The other allowed sizes are: 1 Kbytes, 512 bytes and 256 bytes. See appendix A for information on how to set the dbp to channel packet size through the configure relation.

6.3.2. The IDENTIFY Packet

The IDENTIFY packet must be sent as the first communication with the IDM. It is also sent following a loss of communication or a resynchronization; resynchronization is discussed in the Parallel Interface and Serial Interface sections. The IDENTIFY packet consists of a different set of parameters than the other communication packets. These parameters contain the host-dependent characteristics. The IDM uses this information to convert different data representations supported by various hosts to the representation of data used by the IDM.

The following list of host characteristics are recognized by the IDM. Each characteristic is represented by 1 byte in the IDENTIFY packet, and the octal code for each byte is given below.

HOST CHARACTERISTICS	OCTAL CODE
int2order - order of bytes for 2-byte integer	
least significant byte followed by most significant byte	000
most significant byte followed by least significant byte	001
int4order - order of int2's for 4-byte integer	
least significant int2's followed by most significant int2's	000
most significant int2's followed by least significant int2's	001
The int2's in these 4-byte integers are assumed to be in the order specified by "int2order"	
charrep - character representation	
ASCII code	000
EBCDIC code	001
errordet - error detection	
2's complement checksum (2 byte)	000
none (2-byte check ignored)	377

timeout - a 1-byte integer in minutes
The host agrees to send a communication command at least every "timeout" minutes. If "timeout" minutes passes and no communication from the host is received, the host is assumed to be down. A "timeout" value of 0 turns off this feature. The channel cancels all commands issued by a "timedout" host just as if the host had sent a CANCELH.

hostid - a 2-byte integer used for identification of the host. This integer is assumed to be in the host 2-byte integer format.

IDENTIFY PACKET

HOST to IDM

HEADER	1 byte
IDENTIFY cmd	1
	1
int2order	1
int4order	1
	1
	1
charrep	1
errordet	1
	1
timeout	1
	1
hostid	2
check	2

The IDENTIFY packet is used to: initially establish communication, re-establish communication following a host or IDM failure, and resynchronize following a loss of synchronization. When the host is re-establishing communication following a host failure, it must also cancel all pending IDM requests by issuing a CANCELH command. When the host and IDM are attempting to resynchronize, it is necessary for the host to know whether an IDM failure has occurred. If this has happened, all pending host requests have been cancelled. Therefore, the response to the IDENTIFY packet includes the "idmid" and an indication of whether the IDM has just come back on-line.

6.3.3. Packet Error Detection

Each communication packet and data packet includes an additional 2-byte quantity for error detection, and is acknowledged by the receiver. When either the host or the IDM sends a packet, it must wait for a positive acknowledgement from the receiver. If a negative acknowledgement is returned, then the communication or data packet must be resent.

The IDM supports a 2-byte checksum for error detection of packets. The host specifies, in the IDENTIFY packet, whether it will support a checksum or no error detection. In either case, the 2-byte error detection quantity is present at the end of every packet and every packet must be acknowledged. The IDENTIFY packet must include the error detection quantity that the host has chosen to provide.

If the host chooses not to support error detection, then the 2-byte quantity from the host and from the IDM is ignored. In this case, all packets are always positively acknowledged. The checksum provided by the IDM is simply a 2-byte wide sum of all bytes in the packet. The host includes the checksum by placing it in the 2-byte "check" of the packet in the host 2-byte integer format.

6.3.4. The Acknowledgement Byte and Response Packet

All communication packets and data packets are acknowledged by both the IDM and the host. This is done using a 1-byte acknowledgement. Since a communication packet from the host also requires a YES or NO response, the acknowledgement byte from the IDM to the host includes this response. The acknowledgement bytes are defined as follows:

Host to IDM	Octal	Binary
ACK	000	00000000
NACK	370	11111000

IDM to Host	Octal	Binary
ACK-YES	000	00000000
ACK-NO	037	00011111
NACK	370	11111000

	Octal	Binary
HEADER byte	347	11100111

ACK is used to indicate that the packet was correctly received. If error detection is used, then ACK means that the error detection quantity has been computed and agrees with the error detection quantity appended to the packet. In response to a communication packet, ACK-YES from the IDM indicates that the packet was correctly received and that the request may proceed. ACK-NO from the IDM indicates that the communication packet was correctly received, however the request may not proceed at this time. NACK, a negative acknowledgement, indicates that the packet was not correctly received and informs the sender to resend the packet. ACK-YES and NACK are the only responses given to a data packet from the host since data packets only require an acknowledgement.

Since the correct transmission of the acknowledgement byte is critical, the codes for the acknowledgement byte from the IDM to the host have been chosen to allow for 1-bit or 2-bit error correction. Any 1-bit or 2-bit error in this acknowledgement code can be corrected, since such a code is still uniquely close to one of the legal acknowledgement codes. "Uniquely close" means that the code would differ in bit pattern from only the intended code by 1 or 2 bits (depending on a 1 or 2 bit error).

The error correcting of this acknowledgement byte can be implemented in one of two different ways. The first implementation is to compare the received code to the legal codes by using an exclusive-OR. Then the result of each comparison is bit-counted. A bit count of 0, 1, or 2 indicates the correct code assuming that there has been, at worst, a 2 bit error. Alternatively, the 256 possible codes that may be received can be kept in a table indicating either the corresponding transmitted code, or an error situation.

The response to a data packet is simply the 1-byte acknowledgement. However, in response to a communication packet from the host, each acknowledgement byte from the IDM is followed by a "response" packet. The response packet contains 2 bytes of information in the host 2-byte integer format and is followed by a 2-byte check. This check includes only the 2 bytes of information in the response packet and does not include the preceding acknowledgment byte. The response packet, like all other packets, must be acknowledged by the host. If the host returns a NACK to the response packet, the acknowledgement byte and the response packet will be resent. The information in the response packet depends upon the acknowledgement byte that was sent, but is always 4 bytes in length. Therefore, the IDM response to a communication packet from the host is always a 5-byte quantity: the acknowledgement byte followed by the response packet. A response packet is one of the following:

RESPONSE PACKET FROM THE IDM

```
---------------     -------------     -------------
| ACK-YES     | 1   | ACK-NO    | 1   |   NACK    | 1
---------------     -------------     -------------

EOR
---------------     -------------     -------------
| |  count    | 2   |error code | 2   | bad check | 2
| |           |     |           |     |           |
---------------     -------------     -------------
|   check     | 2   |  check    | 2   |  check    | 2
|             |     |           |     |           |
---------------     -------------     -------------
```

The ACK-YES response is always followed by a "count" response packet. In response to a READ or WRITE request, this count indicates the actual number of bytes that may be read or written. In other words, the count specifies the size of the data packet that will be sent or received by the IDM. This count does not include the 2-byte check appended to every packet. In response to a READ request, the most significant bit of the 2-byte count is set when the IDM is sending the end of the results. This bit is called the EOR or End Of Results bit.

The count returned by the IDM in the "count" response packet may differ from the count sent by the host for both READ and WRITE requests. The IDM will read and write data packets of sizes up to 2K-bytes. Therefore, a READ or WRITE request with a count greater than 2K will be acknowledged by a response packet with a count of 2K or less.

On a WRITE request, provided a count of 2K or less, the IDM will either grant the write request for the count indicated by the host, or will indicate that there is insufficient space for writing. If a host program wishes to send more than 2K bytes, this must be done using multiple WRITE requests. For example, if the host program wishes to send 10K bytes to the IDM, the device might communicate this data to the IDM using 5 WRITE requests, each with a "count" equal to 2K. In any case, when the IDM responds with YES to the write request, it is always followed by a count response packet. Therefore, the device may request to send 10K bytes, and the IDM will return a count indicating how many bytes may be written, such that "count" <= 2K.

On a READ request, the IDM will return the results that are available at the time of the request. Therefore, the IDM may send a count less than that requested by the host. In this case, the device sends multiple READ requests. The IDM indicates the end of the data by setting the EOR bit in the response packet for the final data packet. The host may read results until the EOR count is received.

In the ACK-YES response to an IDENTIFY command, the 2-byte "count" is used for the IDENTIFY command response; i.e. the "idmid" and the single bit IDM failure indicator.

IDENTIFY PACKET RESPONSE

```
-------------------
|    ACK-YES      | 1
-------------------
-------------------
|FI|      | idmid | 2
-------------------
msb             lsb
-------------------
|     check       | 2
-------------------
```

FI - failure indicator - is placed in the most significant bit of the most significant byte. When set, it indicates that the IDM has just returned on-line.

idmid - is placed in the least significant byte of the 2-byte response

In the case of a response to some other IDM command, such as a CANCEL command, the contents of the count response packet can be ignored.

The ACK-NO response is always followed by an "error" response packet. This packet contains a 2-byte error code indicating the error condition that has occurred. The error conditions that may arise were discussed previously, and the corresponding error codes are given below. Negative error codes indicate host system or user errors. The following errors apply to revision 37 or higher of the serial channel and revision 15 or higher of the parallel channel.

Customers with older channels should refer to version 1.3 of the Software Reference Manual.

ERROR CODES

Communication Command causing the error	Error	Code in host 2-byte integer format
READNW READCALL	results not available	1
WRITENW WRITECALL WRCALL	insufficient space for writing	2
READCALL WRCALL	not accepting READCALL requests	3
WRITENW WRITECALL WRCALL	not accepting write requests on this "dbin"	4
ANY	illegal HEADER byte	-1
ANY	illegal communication command	-2
ANY READ ANY WRITE	count <= 0	-3
ANY READ	illegal "dbin"	-4
IDENTIFY	illegal code in IDENTIFY packet	-5
ANY WRITE	Read output for this dbin. This means that multiple commands have been sent without reading any results. Results must be read before issuing any new write commands.	-7
ANY READ	CANCEL acknowledge	-8
ANY	no IDENTIFY sent	-11
ANY WRITE	Must read CANCEL acknowledge before issuing another command. A cancel was issued by the host and the CANCEL acknowledge was not read before another command was issued.	-12
ANY WRITE	Must read IDM error message before any	-13

	additional writes. This can happen on a load database, load transaction, copy in or file write. If an error happens during one of these commands, the host program will get this error. It should then cancel the command it is running and read the IDM error message.	
ANY READ	Results cancelled by channel. This message is returned when the host specifies a timeout period in the configure relation and fails to request results in that amount of time.	-14

The NACK response to a communication packet from the host is always followed by a "bad-check" response packet. The 2-byte quantity in this packet contains the "bad" check-sum or CRC that was computed by the IDM. This may be useful to the host for diagnostic purposes.

6.3.5. Examples

This section illustrates the protocol described in the preceding sections. Each example has been chosen to illustrate a different protocol sequence. An arrow (----->) indicates one communication between the host and the IDM, i.e. all information is sent before any other communication can proceed. Asterisks (* * *) indicate that there may or may not be additional communication during that time.

A. Startup protocol

The IDENTIFY packet is used as the first communication with the IDM. At startup, the protocol is:

```
         HOST                                  IDM
   -----------------
   |  IDENTIFY     |
   | communication |      ------->
   |   packet      |
   -----------------
                                        -----------------
                                        |  ACK   YES    |
                                        -----------------
                          <-------      |response packet|
                                        |  FI + idmid   |
   -----------------                    -----------------
   |    ACK        |      ------->
   -----------------
```

Following this protocol, host requests can be received by the IDM.

B. Re-establishing communication after host failure

Following a loss of communication or following a re-synchronization, the IDENTIFY packet must be issued by the host. Additionally, if the host has gone down and is now re-establishing communication with IDM, the CANCELH ~ommand must be used to cancel all pending IDM commands. Therefore, the sequence to re-establish communication in this case is:

```
         HOST                                    IDM
   ----------------
   |   IDENTIFY    |
   | communication |       ------->
   |    packet     |
   ----------------
                                        ----------------
                                        |  ACK   YES    |
                                        ----------------
                         <-------       |response packet|
                                        |  FI + idmid   |
                                        ----------------
   ----------------
   |     ACK       |       ------->
   ----------------

                          *   *   *

   ----------------
   |   CANCELH     |
   | communication |       ------->
   |    packet     |
   ----------------
                                        ----------------
                                        |   ACK    YES  |
                                        ----------------
                         <-------       |response packet|
                                        |               |
   -----------------                    ----------------
   |      ACK      |       ------->
   -----------------
```

C. A WRITE request

Assume that the device has received a WRITECALL request from the operating system via the SEND registers. The device must now send the request to the IDM in the form of a communication packet. Suppose the "count" of this request is 100 bytes and that the IDM has space to accept the 100 bytes. The protocol would proceed as follows:

```
            HOST                                  IDM

      ----------------
     | WRITECALL      |
     | communication  |        ------->
     | packet         |
     | count = 100    |
      ----------------                     ----------------
                                          | ACK    YES     |
                               <-------    ----------------
                                          |response packet|
                                          | count = 100    |
                                           ----------------
      ----------------
     |      ACK       |
      ----------------
     |                |
     |  data packet   |
     |   100 bytes    |        ------->
     |    + check     |
     |                |
     |                |
      ----------------
                                           ---------------
                               <-------   | ACK    YES    |
                                           ---------------
```

At this point, the transfer is complete and the device sets the status register appropriately.

D. A READ request - Results not available

Suppose the next request that the device receives is a READCALL request. This is the request for the results of the IDM command that was sent to the IDM in the previous example. If the results are not yet available, the communication protocol would proceed as follows:

```
            HOST                             IDM
        -----------------
        | READCALL      |
        | communication |    ------->
        | packet        |
        -----------------
                                        -----------------
                                        |  ACK    NO    |
                             <-------   -----------------
                                        |response packet|
                                        |  results not  |
                                        |  available    |
        -----------------               -----------------
        |    ACK        |    ------->
        -----------------

                               *  *  *

                                        -----------------
                                        | RESULTS       |
                              <-------  | communication |
                                        | packet        |
        -----------------               -----------------
        |    ACK        |    ------->
        -----------------
                               *  *  *
        -----------------
        | READCALL      |
        | communication |     ------->
        | packet        |
        -----------------
                                        ------------------
                                        | ACK     YES    |
                              <-------  ------------------
                                        | response packet|
                                        |  with count    |
        ----------------                ------------------
        |     ACK      |      ------->
        ----------------                ------------------
                                        |   "results"    |
                                        |    data        |
                              <-------  |    packet      |
                                        |   + check      |
                                        |                |
        -----------------               ------------------
        |     ACK       |     ------->
        -----------------
```

E. A READ request - host requests more data than available

To illustrate another communication sequence, suppose that the host requests to read 4K bytes of results using the READNW command. The results are available but the total number of bytes is 3K. The communication protocol would proceed as follows:

```
        HOST                                     IDM
   ----------------
   |   READNW     |
   | communication|
   |   packet     |        ------->
   | count = 4K   |
   ----------------
                                          ----------------
                                          | ACK   YES    |
                           <-------       ----------------
                                          |response packet|
                                          |  count = 2K  |
   --------------                         ----------------
   |    ACK     |          ------->
   --------------                         ----------------
                                          |  "results"   |
                                          |    data      |
                           <-------       |   packet     |
                                          |   2K bytes   |
                                          |   + check    |
   ----------------                       ----------------
   |    ACK       |        ------->
   ----------------
                           *   *   *

   ----------------
   |   READNW     |
   | communication|
   |   packet     |        ------->
   |  count = 2K  |
   ----------------
                                          ----------------
                                          | ACK   YES    |
                                          ----------------
                           <-------       |response packet|
                                          |   count = 1K |
                                          |  + EOR bit set|
   ---------------                        ----------------
   |     ACK     |         ------->
   ---------------                        ----------------
                                          |  "results"   |
                                          |    data      |
                           <-------       |   packet     |
                                          |  1K bytes    |
                                          |  + check     |
   --------------                         ----------------
   |    ACK     |          ------->
   --------------
```

F. Negative acknowledgements

The following protocol illustrates the action to be taken when a negative acknowledgement is received.

```
            HOST                                     IDM
        -----------------
        |   WRITEW      |
        | communication |     ------->
        |   packet      |
        -----------------
                                      -----------------
                                      |    NACK       |
                                      -----------------
                            <-------  |response packet|
                                      |  bad check    |
        -----------------             -----------------
        |     ACK       |     ------->
        -----------------
                              *  *  *

        -----------------
        |    WRITEW     |
        | communication |     ------->
        |   packet      |
        -----------------
                                      -----------------
                                      | ACK    YES    |
                                      -----------------
                            <-------  |response packet|
                                      |  with count   |
        ----------------              -----------------
        |    NACK      |      ------->
        ----------------
                                      -----------------
                                      | ACK    YES    |
                            <-------  -----------------
                                      |response packet|
                                      |   with count  |
        ----------------              -----------------
        |     ACK      |
        ----------------      ------->
        |     data     |
        |    packet    |
        |   + check    |
        ----------------
                                      -----------------
                            <-------  |    NACK       |
        ----------------              -----------------
        |     data     |      ------->
        |    packet    |
        |   + check    |
        ----------------
                                      -----------------
                            <-------  | ACK   YES     |
                                      -----------------
```

G. Cancel Protocol

The protocol for issuing a CANCEL on a particualar "dbin" proceeds as follows.

```
        ----------------
        |    CANCEL    |
        |    dbin x    |
        | communication|     ------->
        |    packet    |
        ----------------
                                        ----------------
                                        |  ACK   YES   |
                                        ----------------
                             <-------   |response packet|
                                        |  info = 0    |
        ---------------                 ----------------
        |    ACK      |      ------->
        ---------------

                              *  *  *

        ----------------
        |READW,READNW or|                              |
        |   READCALL    |                              |
        |   dbin x      |
        | communication |    ------->
        |   packet      |
        ----------------
                                        ----------------
                                        |  ACK   NO    |
                                        ----------------
                             <-------   |response packet|
                                        | with cancel   |
                                        |acknowledgement|
        ---------------                 ----------------
        |    ACK      |      ------->
        ---------------
```

6.4. The Parallel Interface

6.4.1. The IEEE-488 Bus

Parallel I/O to the IDM is achieved through the standard IEEE-488 bus (GPIB). The characteristics and features of this standard are described in "IEEE Standard Digital Interface for Programmable Instrumentation"[1] This bus can be used to connect a single host or several hosts to the IDM. To simplify communication over the bus, to allow for efficient sharing of the IDM among hosts, and to avoid the locking of the bus due to a single host failure, the IDM is the SYSTEM CONTROLLER and CONTROLLER-IN-CHARGE of the bus at all times. It is assumed that all devices on the bus are IDM hosts.

The IEEE-488 bus standard document provides capability identification codes for identifying the interface functions and subsets implemented by a particular device (Appendix C of the IEEE-488 Standard). An IDM host that chooses to communicate using the parallel interface must have talker, listener and serial poll capabilities. The specific requirements that an IDM host must satisfy and the IDM interface characteristics are given in terms of the allowable subsets defined in the standard document.

[1] "IEEE Standard Digital Interface for Programmable Instrumentation", IEEE Std 488-1978, Published by The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, New York 10017.

	HOST REQUIREMENTS	IDM CHARACTERISTICS
Source Handshake	SH1 complete capability	SH1 complete capability
Acceptor Handshake Function	AH1 complete capability	AH1 complete capability
Talker Function	T2 basic talker serial poll	T3 basic talker talk only mode
Talker Extended Function	TE0 no capability	TE0 no capability
Listener Function	L2 basic listener	L1 basic listener listen only mode
Listener Extended Function	LE0 no capability	LE0 no capability
Service Request Function	SR1 complete capability	SR0 no capability
Remote Local Function	RL0 no capability	RL0 no capability
Parallel Poll Function	PP0 no capability	PP0 no capability
Device Clear Function	DC0 no capability	DC0 no capability
Device Trigger Function	DT0 no capability	DT0 no capability
Controller Function	C0 no capability	C1 system controller C2 send IFC and

	take charge
C4	respond to service request
C27	send interface messages and take control synchronously

6.4.2. The Protocol

The host bus address is specified as described in the Operator Manual, or the default configuration may be used. The default configuration assumes that there is a single host on the bus at bus address 1.

When a host wishes to send a communication packet to the IDM, the host device issues SRQ (service request). The IDM will SERIAL POLL hosts to determine the requester(s). After the SERIAL POLL sequence, the IDM addresses each requester, in turn, to TALK. When the host is addressed to TALK, it sends a communication packet. The IDM receives the packet and addresses the host to LISTEN in order to send the acknowledgement byte and response packet. Continuing the protocol, the IDM then addresses the host to TALK so that the host can send the acknowledgement byte. This is the shortest protocol sequence that is allowed on the bus, i.e. no other communication can occur before the acknowledgement byte is received. The IDENTIFY packet is an example of this sequence. In order to send another communication packet, the host issues SRQ, and the protocol is repeated. Each time that the host is addressed to TALK, the IDM later will issue the UNTALK command to unaddress the host. After a host is addressed to LISTEN, the UNLISTEN command will be issued.

At startup, the IDM issues IFC (Interface Clear) to clear the bus. Following this, the action of the host is to issue SRQ (service request). A SERIAL POLL then is performed by the IDM and all addresses are polled. The host must use the same device address for talking and listening. Following the serial poll, the host is addressed to TALK, and it must send the IDENTIFY communication packet. The IDM will address the host to LISTEN and send the acknowledgement byte and response packet. The host is addressed to talk to send its acknowledgement byte and the startup protocol is complete.

When the IDM wishes to send a communication packet to the host, the host will be addressed to LISTEN. This will only occur outside of a host-to-IDM communication sequence, i.e. a sequence initiated by the host. Therefore, when the host is addressed to LISTEN, it can determine which kind of packet to receive. The IDM then sends the communication packet, addresses the host to TALK, and expects the return of the acknowledgement byte.

If the host receives a NACK to a communication packet, it must acknowledge the NACK response packet and then restart the communication sequence by issuing a service request. If a NACK is received to a data packet, the host does not issue service request, but simply retransmits the data packet.

To illustrate the use of the bus protocol with the IDM protocol, an example from the previous section has been repeated below. The example assumes that the host is at bus address 1. The bus protocol is given in angular brackets. The actual octal bus commands are specified in parentheses.

```
       HOST                              IDM
=========================================================

  < service request >

                         < serial poll enable >  (030)
                         < talk host 1 >         (101)

  < send status byte >
                         < untalk >              (137)
                         < serial poll disable > (031)

                         < talk host 1 >         (101)

    -----------------
    | WRITE         |
    | communication |    ------->
    | packet        |
    -----------------

                         < untalk >             (137)
                         < listen host 1 >      (041)

                                    -----------------
                                    | ACK    YES    |
                         <-------   -----------------
                                    |response packet|
                                    |               |
                                    -----------------

                         < unlisten >           (077)
                         < talk host 1 >        (101)

    -----------------
    |     ACK       |
    -----------------
    |               |
    |  data packet  |
    |               |    ------->
    |               |
    -----------------

                         < untalk >             (137)
                         < listen host 1 >      (041)

                                    ----------------
                         <-------   | ACK    YES   |
                                    ----------------

                         < unlisten >           (077)
```

6.4.3. Resynchronization

If transmission errors occur over the communication line causing the loss of synchronization in the communication protocol, IFC (Interface Clear) will be issued by the IDM. Following every resynchronization, the startup protocol is performed, and the host must send an IDENTIFY packet. This action may take place as a result of repeated negative acknowledgements (NACKs) from the host, when an acknowledgement byte is not recognizable, or when synchronization is lost for any reason.

Loss of synchronization may occur during a host-IDM communication sequence, or outside of such a sequence. The host must always respond to IFC by performing the startup protocol. IFC indicates to the host that: synchronization was lost between the IDM and the host, synchronization was lost between the IDM and another host on the bus, or the IDM is returning after a failure. Therefore, upon receiving IFC, the host must:

```
1. perform startup protocol
2. check response
        if IDM failure occurred:
                cancel all user programs waiting
                 on IDM requests
        if the host was in a host-IDM
          communication sequence:
                re-issue interrupted request
```

6.5. The Serial Interface

6.5.1. The RS-232-C Interface

Serial I/O to the IDM is achieved through a standard RS-232-C Interface. The characteristics of this standard are described in "EIA Standard RS-232-C"[2] Communication is asynchronous using a full duplex character oriented protocol. The data transmitted consists of 8 bit binary with no parity and one stop bit. The baud rates supported by the IDM are:

[2] "EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange", Electronic Industries Association, August 1969.

```
  150
  300
  600
 1200
 1800
 2400
 4800
 9600
19200
```

The host specifies the configuration of the serial ports as described in the Operator Manual, or uses the default configuration which assumes all serial ports at 9600 baud.

6.5.2. The Protocol

The communication protocol described in the previous section can be easily achieved for the SERIAL interface. Although full duplex communication is possible over the RS-232-C line, the communication protocol defined in the previous section enforces an actual protocol that is similar to half duplex main channel protocol. The IDM and the host are not sending data at the same time. Due to the full duplex capability, however, there exists a race condition in the serial communication protocol. The HEADER byte, which is sent with every communication packet, is used to detect this race condition.

The condition can arise only if the IDM is sending a communication packet (e.g. the IDM has been requested to call the host when results are available). If the host and the IDM both send communication packets at approximately the same time, they will each receive communication packets rather than acknowledgement bytes. This condition is easily detected. The host always reads one byte after sending a communication command. That byte will be one of the values: ACK-YES, ACK-NO, NACK, or HEADER. The byte is error corrected as discussed in the previous section. If the byte is HEADER, then the race condition has occurred. At this point the host will read and disregard the communication packet from the IDM and will await the acknowledgement byte that it was expecting previously. The IDM also detects the condition, sends the appropriate acknowledgement and, at a later time, resends the communication packet that was disregarded by the host.

6.5.3. Resynchronization

If synchronization is lost during communication or the host wishes to abort the current communication sequence with the IDM, the host may issue BREAK (defined by the RS-232

Standard). This will reset the protocol between the IDM and the host. Following this BREAK, the IDM is ready to accept a communication packet from the host.

To aid in the resynchronization process, when the IDM is expecting to read a communication packet from the host, it will ignore all characters until it reads a HEADER byte. If it is not possible for the host to send a BREAK, it can attempt to resynchronize by simply sending a communication packet to the IDM. The IDM will always attempt to resynchronize when it receives characters at a time when characters from the host are not expected. However, without using BREAK, resynchronization by sending a communication packet to the IDM is not always possible, since the IDM may be expecting data from the host.

Resynchronization is used when the program actually communicating with the IDM abnormally terminates, or when it is desirable to stop a communication protocol with the IDM. An example of the latter case is when the host has issued a READW command and is waiting for the results to be returned. If the host wishes to abort this request, it issues a BREAK and then the CANCEL communication packet to flush the results.

7. IDM Command Set

This section presents the IDM command set. The IDM does not have a "machine language" in the normal sense. To "program" the IDM it is necessary to send high level commands to it and interpret the results. Every command starts with a command "token" or op-code. Other tokens are used in the definition of parameters to the command. Throughout this document tokens are given in capital letters and are fully defined in Appendix B. Numbers to the immediate right of the token are one-byte length specifiers. This is sent after the token and is followed by that many bytes of data associated with that token.

As mentioned in Section 5, commands take several different kinds of parameters. The most complex of these are the query trees. A discussion of the general syntax of the trees is given in Section 5. In this section we show the complete command list and associated query trees for the IDM. The trees are always sent to the IDM using a post-order traversal.

In the query tree examples with each command, the information below the token in parentheses is sent after the token. Some of the error messages which could be received are given with explanations after each command. The complete list of error messages is given in Appendix C.

Because it is difficult to understand the commands without giving query-language examples we define the query language IDL. IDM parse tree syntax is summarized in Appendix E.

For each command we provide four things: the IDL syntax in a form similar to BNF, the octal command token value, a description of the command, and examples showing both the IDL form and the resulting IDM command. To facilitate presenting the IDL syntax we first define the meta-symbols which will be used in the command definitions.

7.1. IDL meta-symbol definitions

The following symbols are used in examples of IDM commands:

´(´, ´)´	- Parentheses are necessary, and must appear literally in the command.
´[´, ´]´	- Anything included in square braces is optional.
´|´	- A vertical bar indicates that a choice of words is presented.
´{´, ´}´	- Curly braces indicate that the word may appear

0 or more times.

´/*´, ´*/´ - Words between these symbols are explanatory comments.

´<´, ´>´ - Words in angle braces are meta-symbols, explained below.

All other words are key words and must appear literally.

Definitions of meta-symbols:

7.2. Target List

<target list>: <target list element>
{, <target list element> }

The target list is the list of objects that are affected by the command. In the case of a retrieve:

retrieve (e.name)

the target list is the attribute "e.name". In the query tree the target list is on the left of the ROOT node. It is ended by a TLEND (target list end) node. The query which would be sent with this command would be:

```
              ROOT
              (0 0)
             /     \
        RESDOM    QLEND
        /     \
   TLEND      VAR 5
             (0 name)
```

The tree would be sent using a post order traversal. Items in parentheses below a node are data associated with that node and are sent after the token value and length.

The target list consists of RESDOM, RESATTR and ORDERDOM nodes along the left edge of the tree with expression subtrees as their right descendants. In a retrieve, the order of the target list elements is important because it determines the order in which data will be returned. The first target list domain is the left child of the root. The last domain is the parent of the TLEND node. RESDOM nodes point to result domains in the target list. RESATTR nodes are used to send the name of a target list domain, in an append, for example. ORDERDOM nodes are used in retrieve commands to specify a target list element which is not to be returned to the host but is used in sorting the data.

<target list element>:

<name> = <expression> | attribute | <variable> . all

Target lists can be simple, as in the case above, where the target list was composed of a single relation variable and an attribute name. Arbitrary expressions, as defined below, can also be included in the target list. For example:

retrieve (e.name, wages = e.salary * e.hours)

```
                        ROOT
                        ( 0 0 )
                        /     \
                       /    QLEND
                  RESDOM
                   /     \
                  /     VAR 5
             RESDOM    (3 name)
             /     \
        TLEND       \
                   MUL
                   /  \
                  /    \
             VAR 7     VAR 6
           (3 salary) (3 hours)
```

In the above query, the target list is composed of the attribute "e.name" and the expression "e.salary * e.hours". The expression must be given a name in order to be displayed to the user; the name assigned in the above query is "wages". A RESATTR node must be sent instead of a RESDOM node if the IDM is to return the names of the columns in response to command–option 2 in the retrieve command:

```
                      ROOT
                     ( 0 0 )
                     /     \
                    /    QLEND
               RESDOM 5
                (name)
               /      \
              /      VAR 5
       RESATTR 5    (3 name)
        (wages)
        /      \
   TLEND        MUL
               /   \
              /     \
          VAR 7    VAR 6
       (3 salary) (3 hours)
```

The construct, <variable> . all, is used to reference all attributes of a relation. This is represented by an ATTRALL node. The form is only valid in the target list and is expanded by the IDM to the equivalent tree using RESATTR and VAR nodes.

7.3. QUALIFICATION

```
<qualification>:     ( <qualification> )
               |   not <qualification>
               |   <qualification> and <qualification>
               |   <qualification or <qualification>
               |   <clause>
```

The qualification is the part of the database command that determines which objects are affected by the command. The command:

```
delete emp where emp.salary > 24000
```

deletes from the relation associated with the variable "emp" all employees whose salary is over 24000. The clause "emp.salary > 24000" is the qualification. The above command has no target list. The following tree would be used in this command:

```
              ROOT
              (0 0)
             /    \
            /      \
        TLEND      GT
                  / \
                 /   \
                /     \
              VAR 7   INT2
          (0 salary)  24000
```

A qualification is a boolean expression of relational clauses.

```
<clause>:         <expression> <relop> <expression>
```

A relational clause may only appear in a qualification. The operands may be any expression.

```
<relop>:            =        /* equal */
    |          !=       /* not equal */
    |          <        /* less than */
    |          <=       /* less than or equal */
    |          >        /* greater than */
    |          >=       /* greater than or equal */
```

Relational operators are supported by the IDM for all data types. If expressions are characters the comparisons are made on the basis of ASCII sort order. Blanks at the end of character strings are ignored for comparison purposes. Zeros at the end of binary strings are ignored for comparison purposes. Floating point binary numbers are compared as the first byte signed and the remaining bytes unsigned. The operands may be any expression.

7.4. Expression

```
<expression>:          <aggregate>
    |          <attribute>
    |          <constant>
    |          <expression> <arithop> <expression>
    |          - <expression>
    |          ( <expression> )
    |          <constant function>
    |          <unary function> ( <expression> )
    |          <with length function> (<intl>,
                       <expression> )
    |          <binary function> ( <expression>,
                       <expression> )
    |          <ternary function> ( <intl>, <intl>,
                       <expression> )
```

The IDM supports expressions in both the target list and qualification for most commands. We will explain all of the terms used above, then give examples of the use of expressions.

```
<arithop>:          +    /* addition       */
                  | -    /* subtraction    */
                  | *    /* multiplication */
                  | /    /* division       */
```

Arithmetic operators are supported by the IDM only for integer and BCD expressions. The IDM allows different integer types in the same expression. An operation is always done in the type of its longest integer operand.

```
<aggregate>:          <aggop> ( <expression>
     [ by <expression> { , <expression> } ]
     [ where <qualification> ] )
```

An aggregate is an arithmetic function on the data specified which returns one or a set of values. The "aggop" defines which function to use.

```
<aggop>: sum [unique] | count [unique] |
             avg  [unique] | once [unique] |
             any | max | min
```

The aggregate, once, cannot be used in an aggregate function. "Once" returns one and only one value. If no value exists, or if more than one value exists, an error message is generated. The aggregate operators of sum, count, and avg will perform the sum, count, and avg functions on either all the data specified, or, if "unique" is specified, only on unique values. Sum and avg can only be used with integer or BCD expressions; the others can be used on expressions of any type. An example of an aggregate in the target list is:

```
    retrieve (first = min(e.name))
```

```
                    ROOT
                    (0 0)
                   /     \
                  /       QLEND
             RESDOM
             /    \
            /      \
       TLEND      AGHEAD
                  /     \
                 /       \
             AOPMIN   QLEND
               |
             VAR 5
             (2 name)
```

The above example puts the alphabetically first employee's name in the position named "first". Notice that the

This page has been intentionally left blank.

aggregate has a qualification which must be present in the tree even if empty. An example of an aggregate in the qualification is:

```
delete emp where emp.sal > avg (emp.sal)
```

This example deletes from the relation associated with the variable "emp" all those employees who make more than the average salary.

An example of an aggregate function with a "by clause" is:

```
retrieve (dept_total = sum (e.salary by e.deptno))
```

which retrieves the sum of the salaries, grouped by department number. The tree for aggregate functions has a new node, the BYHEAD, which is the head of the "by list". The by list is structured exactly like a target list for a retrieval.

```
              ROOT
              ( 0 0 )
               /   \
              /  QLEND
         RESDOM
          /       \
TLEND              \
              ARGHEAD
               /       \
              /         \
          BYHEAD      QLEND
           /     \
          /       \
    RESDOM     AOPSUM
     /       \      |
TLEND    VAR 7     VAR 7
        (3 deptno) (3 salary)
```

When used with the "by list" aggregates are one of the most powerful tools in IDL. See the previous section, "Introduction to IDL", for more examples.

<constant function>: userid | dba | host | gettime | getdate| databasename

Constant functions take no arguments. They are useful for getting at IDM system data.

"userid" returns the IDM user id for the current user (a two–byte integer).

"dba" returns the IDM user id for the database administrator for the current open database (a two–byte integer).

"host" returns the IDM host id from which the current command came; it is a 2–byte integer sent from the host in the IDENTIFY packet. See Chapter 6 for details of the IDENTIFY packet.

"gettime" is the number of 60ths of a second since midnight. The value will typically be wrong after the IDM has been brought on–line. The "settime" command allows any host to reinitialize its value.

"getdate" counts the number of days. It can be initialized to any value by the "setdate" command. When the time reported by "gettime" reaches the number of 60ths of a second in 24 hours, it is reset and the date reported by "getdate" is incremented by 1.

"databasename" is the character string name of the currently opened database.

<**unary function**>: int1 | int2 | int4 | rel_name | rel_id | abs | binary

These functions take one argument, which may be any expression except in the case of "abs". "abs" is invalid for character and binary data types.

The functions int1, int2, and int4 will convert BCD, character, or one–, two–, or four–byte integers or fixed length strings of the proper length, to the specified result. For instance,

```
replace emp
    (wages = int4 (emp.salary) * emp.hours)
```

will cause the IDM to first convert emp.salary to a 4–byte integer, then multiply it by the employee's hours, then store the result in wages. If the operand is of type character it must contain only digits or be of zero length.

"rel_name" expects a relid of an object and returns its name. If the number is invalid, a zero length string is returned.

"rel_id" expects a name of an object and returns the relid of the object. The object name argument may be followed by a colon followed by a user name to specify an object not owned by the user submitting the command. If the name is invalid a zero is returned.

"abs" returns the absolute value of its argument.

"binary" converts its argument to the "binary" type. It does not change the value of the argument in any way.

```
<with length function>:          [fixed] bcd | [fixed] bcdflt
        | [fixed] string
```

The syntax for "bcd" is:

    bcd(precision, expression)

The function converts "expression" to a BCD integer with a maximum of "precision" digits. For example,

    bcd(5, "123")

returns a BCD integer whose value is 123.

    bcd(3, "12345")

returns OVERFLOW status.

    bcd(4, "1234.56")

returns the truncated integer 1234.

The syntax for "bcdflt" is:

    bcdflt(precision, expression)

The function converts "expression" to a floating-point BCD number with a maximum of "precision" significant digits, rounding the value if necessary. For example:

    bcdflt(4, "123.45")

returns a floating point BCD number whose value is 123.4.

    bcdflt(5, "1234567.89")

returns a floating-point BCD number whose value is 1234600.

If the precision parameter is 0, the number of digits required to store the converted expression is used. In addition, the functions "bcd" and "bcdflt" can be preceeded by "fixed" to indicate that the "precision" parameter specifies the exact number of digits to be generated. The number is padded with zeros to the right of the decimal point to accommodate the field. It does not affect the arithmetic routines.

The conversion function "string" will take 1-, 2-, or 4-byte integers, binary, or BCD strings and convert them to a character string. The length must be a one-byte integer constant. The length for "string" is expressed in bytes.

If preceeded by "fixed" the result is blank-padded to that length; otherwise the length specifies the maximum length of the result. Trailing blanks are deleted or added as required when non-zero length is specified. If the length is zero the conversion will be done into the minimum amount of space needed.

For all three functions, the following lengths (in bytes) will be generated by zero-length conversions.

Conversion Table		
Operand type	string	bcd or bcdflt
int 1	4	3
int 2	6	4
int 4	11	7
bcd or bcdflt	(2 * length) - 3	no-op
char	no-op	(length / 2) + 2
binary	length	length

<binary function>:   mod | concat | att_name

The "mod" function takes two arguments and gives the remainder of the first divided by the second. It can only be used on integer or BCD expressions. An example is:

```
replace emp
  (num_children = mod(emp.num_children, 12))
```

which will take the number of children an employee has (as specified in the relation denoted by "emp"), divide that number by 12, and store the remainder in num_children. mod(n, 0) is defined to be equal to "n".

```
                      ROOT
                      (0 0)
                     /    \
                    /    QLEND
                   /
             RESATTR 12
           (num_children)
           /            \
       TLEND             \
                         MOD
                        /   \
                       /     \
                  VAR 13    INT1
            (1 num_children)  12
```

Note the result variable number in the ROOT node.

"concat" takes two character strings, strips all trailing blanks from the first string (all but one, if the string is all blank), strips all trailing blanks from the second string, and appends the second to the first. "concat" performs the same functions for binary strings, except trailing zero bytes are stripped instead of trailing blanks.

For instance:

```
retrieve ( name = concat ( emp.first, emp.last ) )
                        ROOT
                        ( 0 0 )
                       /      \
                RESATTR 4   QLEND
                 (name)
                 /     \
                /       \
           TLEND      CONCAT
                       /  \
                      /    \
                     /      \
                 VAR 6     VAR 5
               (14 first) (14 last)
```

would return an employee's first and last names concatenated in the domain named "name".

"att_name" is useful in writing queries to the system catalogues. Its first argument is a relation id and its second is an attribute id. It returns the name of the attribute or a null string if the relation id, attribute id pair is invalid.

<**ternary function**>: substring | bcdfixed

The syntax for "substring" is substring (begin, length, expression). "substring" takes characters (or binary bytes) beginning at position "begin" of a character string "expression" and copies them to a result domain. The number of characters (bytes) to copy is denoted by "length". "begin" and "length" are one–byte integer constants. They follow the SUBSTR token in the IDM command. For instance,

```
replace emp (dept_name = substring (3,2,dept.name) )
```

will take the characters beginning at position 3 in dept.name, and copy them for a length of 2 characters, placing them in the attribute dept_name. Substring is valid on character and binary expressions. If the substring extends past the end of the original string it is blank or null padded (respectively).

The syntax for "bcdfixed" is:

```
bcdfixed (precision, fraction, expression)
```

The function converts "expression" to a floating–point BCD number with a maximum of "precision" digits, and a maximum of "fraction" significant fractional digits, rounding the ——

value if necessary. For example,

```
bcdfixed (5, 2, "768.534")
```

returns a floating–point BCD number whose value is 768.53:

$7.6853 * 10^2$

```
bcdfixed (4, 3, "123.45")
```

returns OVERFLOW error status.

```
bcdfixed (8, 2, "35.478")
```

returns a floating–point BCD number whose value is 35.48:

$3.54800000 * 10^1$

If the precision parameter is 0, the number of digits required to store the converted expression is used.

<attribute>: <variable> . <name> | <name>

An attribute is either the pair (relation variable, attribute name) separated by a period or just the attribute name. The former is used in IDL statements and is represented as a VAR node in a querytree. The latter form is used in SQL statements and is represented as a ATTR node. The ATTR node is followed by the range of range variables to which this attribute might belong and the name of the attribute. It is an error for the named attribute to be in more than one of the relations. There are cases in SQL where the name of the attribute is not known. This is represented by a NVAR node that gives the range variable and attribute number.

<variable>: <name>

A variable is associated with a relation using a range statement. The variable is represented in the tree by a VAR node. The VAR node has a "variable number" which ranges from 0 to 15, and a particular domain fo the relation.

<constant>: <string> | <binary string>
| <pattern matching string>
| <one–byte integer>
| <two–byte integer>
| <four–byte integer>
| <four–byte float>
| <eight–byte float>
| <bcd>
| <bcdflt>
| <parameter name>

A "string" is a sequence of alphabetic or numeric characters. In IDL a "string" is a sequence of alphabetic or numeric characters delimited with quotation marks ("), though the delimiters are not actually sent to the IDM. For instance, to find the salaries of all employees named Jones, the command is:

```
    retrieve (e.salary) where e.name = "Jones"
```

This page has been intentionally left blank.

```
                          ROOT
                         (0 0)
                        /     \
                       /       \
              RESDOM            \
              /    \             EQ
             /      \           /  \
        TLEND        VAR 7     /    \
                 (1 salary) VAR 5 CHAR 5
                           (1 name) (Jones)
```

A pattern matching string is like a string except that certain characters are used to match patterns.

```
    * - matches zero or more characters.
    ? - matches any one character.
    [ - begin a group of characters any one of
                  which may be matched.
    ] - end the group of characters.
    \ - escape any of the above.
```

Pattern matching strings may only appear in the qualification of a command. The host program must translate the special characters into the following values depending on whether the host is ASCII or EBCDIC based:

	ASCII	EBCDIC
* -	0200	0334
? -	0201	0335
[-	0202	0336
] -	0203	0337
\ -	processed only in the host.	

<name>:   /* any alphanumeric sequence
    beginning with an alphabetic character */

Names are limited to 12 characters.

<parameter name>: $<name> | $<integer>

Parameters are used in defining stored queries.

<object name>:   <name> [: <user>]

A relation, file, view or stored command may be specified either by just its name or by its name and owner. If no owner is specified then the object must be owned by the current user. If that object does not exist then the name refers to an object owned by the database administrator. The <object name> meta-symbol will be used for any of the above objects to signify that it may be given in that format. Often the object must be of a specific type, most often a relation.

```
<with-node-options>:          <with-node-option> = <constant>
      { , <with-node-option> = <constant> }
```

With-node-options are used in place of a qualification in database maintenance statements such as create and destroy. Each with-node-option is represented by a one-byte value (see Appendix B) which is passed in the query tree in a WITH node. The value of the with-node-option is given to the left of the WITH node. Examples are provided with the appropriate commands.

```
<format>: i1
        | i2
        | i4
        | f4
        | f8
        | c1...c255
        | uc1...uc255
        | bin1..bin255
        | ubin1...ubin255
        | bcd1...bcd31
        | ubcd1...ubcd31
        | bcdflt1...bcdflt31
        | ubcdflt1...ubcdflt31
```

Formats are specified in the create command; the available formats are 1- ,2- ,and 4-byte integer, 4- and 8-byte floating point (floating point numbers are stored, compared, and retrieved only; the IDM does no floating point arithmetic) and three kinds of variable length attributes. Character attributes are either compressed (e.g., c10 signifies a compressed character attribute that is a maximum of 10 characters long) or uncompressed (uc19 signifies that the attribute is to be always stored as 19 characters, even if they are all blanks). Character compression is performed by deleting trailing blanks.

A binary attribute is a binary string that is stored simply as it is received from the host system. "Uncompressed" binary strings (e.g. ubin5 means an uncompressed binary string, 5 bytes long) are zero filled to the length specified when the data is received from the host. "Compressed" binary strings (e.g. bin200 means a binary string with a maximum length of 200 bytes) have trailing zero bytes deleted.

Integer and floating-point BCD attributes also are either compressed (variable-length) or uncompressed (fixed-length). The length specified is the number of digits, so the actual length is (n/2)+2, where n is the number of digits and any remainder is dropped on the division by 2 (i.e., the number of digits is always odd; if an even number of digits is specified, it is tacitly incremented by one). Compressed BCD attributes use less storage because leading and trailing zeros are dropped. Leading and trailing zeros are left alone in uncompressed BCD attributes.

<attlist>: <name> {, <name>}

The attlist is the list of attributes affected by a command.

<protect mode>: read | write | execute | all

Protection commands use the protect mode to define the type of protection desired.

7.5. Commands

7.5.1. ABORT TRANSACTION

IDL SYNTAX

abort transaction

OCTAL COMMAND CODE: 322

DESCRIPTION

Abort transaction causes the current IDM transaction to be aborted. All logical effects of the transaction will be undone. If the user did not previously send a *begin transaction* this command results in error.

ERROR MESSAGES:

illegal command

A user has not sent previously a *begin transaction* command.

7.5.2. APPEND

IDL SYNTAX:

```
append [ to ] <object name> ( <target list> )
        [ where <qualification> ]
```

OCTAL COMMAND CODE: 303

DESCRIPTION:

The append command adds zero or more tuples to a relation or a view. The attributes are named, and the value specified for each attribute. An attribute for which the value is not specified is assigned the default value: blanks for character attributes, zero for numeric attributes. Command-options to turn off overflow and divide by zero checking as well as ignore duplicates may be given. Command-options to send format and attribute name information are ignored. (See the retrieve command for the setting of command-options.)

EXAMPLES:

```
range of pr is newparts
append to parts (name = pr.part, quan = 10)
```

The above command adds tuples to the "parts" relation, taking the "name" attribute from the relation bound to the variable "pr", and assigning a value of "10" to the quantity attribute of each tuple added.

```
          APPEND
          RANGE 9 2 newparts
          RANGE 6 3 parts
                                        ROOT
                                       (3 0)
                                       /   \
                                      /   QLEND
                                RESATTR 4
                                 (name)
                                /      \
                               /        \
                        RESATTR 4      VAR 5
                         (quan)       (2 part)
                        /      \
                    TLEND       \
                              INT1
                              (10)
          ENDOFCOMMAND
```

For SQL "insert" statements, RESDOM nodes may be sent instead of RESATTR statements:

```
    insert into rel
                  values: ´a´, 1, ´mike´
```

```
APPEND
RANGE 4 0 rel
                             ROOT
                            (0  0)
                            /    \
                      RESDOM     QLEND
                      /    \
                     /   CHAR 1 ´a´
                 RESDOM
                 /    \
                /    INT1 1
           RESDOM
           /    \
      TLEND   CHAR 4 ´mike´

ENDOFCOMMAND
```

One RESDOM must be sent for each attribute in the relation or an error will be generated.

ERROR MESSAGES:

<u>out of space</u>
No m re tuples can be added because the database is out of free space. The database should be extended, or relations within the database destroyed.

quota exceeded
When the relation was created a quota was given: the addition of this tuple would cause the relation to exceed the quota.

not found
The named relation or attribute was not found.

wrong type specified for attribute
If a conversion from ASCII to integer or numeric (or vice-versa) must take place, it should be explicitly stated in the append command:

```
append to rell (name = string(10,p.number))
                  where p.number > 10
```

The above command adds to the relation "rell" all parts where the number is above 10; furthermore it converts the integer "number" to a ten-character value and stores it in the attribute "name" of the "rell" relation.

tuple too large
One of the appended tuples is more than the maximum size of a tuple (2000 bytes). The append is therefore not performed.

This page has been intentionally left blank.

7.5.3. ASSOCIATE

IDL SYNTAX

```
    associate <object name> | <attribute> [ [ with ]
        <string> [ , <string> ] ]
```

OCTAL COMMAND CODE: 333

DESCRIPTION:

The associate command is used to add or replace information in the description catalogue. If <object name> is specified then description refers to the entry of the catalog for that relation, file, view or stored command. If an attribute is specified then the description refers to that attribute. The user must be the owner of the relation, file, view or stored command. The first string corresponds to the "text" field in the catalogue. The second corresponds to the "key" field. If the second string is provided it will be truncated to 2 characters and is treated as a key along with the relation and attribute id´s. If neither string is present, all tuples in the "descriptions" relation applying to <object name> or <attribute> are deleted and nothing is added. If the associate command refers to a record which (on all three keys) is already in the catalogue the description is replaced; otherwise it is appended.

EXAMPLES:

To associate a description with the "parts" relation:

```
    associate parts "Relation listing all parts"
```

```
        ASSOCIATE
        RANGE 6 4 parts
                    ROOT
                    (4 0)
                   /     \
                  /       \
            TLEND    QUALDOM
                      /     \
                     /       \
               CHAR 26    QLEND
            (Relation listing all parts)
        ENDOFCOMMAND
```

To add the fact that "number" has an index this command would be used.

```
    range of p is parts
    associate p.number
                 "Has a clustered index on number", "I1"
```

```
ASSOCIATE
RANGE 6 3 parts
                ROOT
                (3 0)
               /   \
              /     \
             /       \
       RESDOM          QUALDOM
      /     \          /     \
     TLEND   \        CHAR 31  \
          VAR 7  (Has ...)  QUALDOM
        (3 number)           /     \
                            /       \
                        CHAR 2     QLEND
                        (I1)
ENDOFCOMMAND
```

"I1" is a user assigned key, in this case it means that there is an index on this domain. The command can also be used with views:

```
associate myparts
      "range of p is parts create view myparts
            retrieve (p.all) where p.pnum < 20", "V"
```

This provides a human-readable definition of the view.

7.5.4. AUDIT, AUDIT INTO

IDL SYNTAX:

```
     audit [[ into ] <object name>  ] ( <target list> )
          [ where <qualification> ]
```

OCTAL COMMAND CODE: 331, 332

DESCRIPTION:

audit creates an audit report from the transaction log or from a copy of it (i.e. the output of a dump transaction). The transaction log is not in a form readable by users. This command returns a formatted output of the log in the order in which modifications to the database took place. For an audit command, the output is returned to the host, while for an audit into command, the output is stored in a relation specified by the user. The tuples being returned may have different formats and will be preceeded by a FORMAT record if it changes. The qualification and target list are limited to refer to the following attributes:

time	- time of the update
date	- date of the update
user	- user who did it
relid	- relation involved
number	- internal transaction number
type	- type of update
value	- data which was changed

"value" may not appear in the qualification. If the audit report is to be generated from a log (as opposed to the transaction log), the log must belong to the currently open database and must correspond to that database. For an audit into command, the value field may appear in the target list only if a restriction on relid is imposed which would limit relid to a single value. One and only one variable can correspond to a log in the command.

EXAMPLES:

```
          range of t is transact
          audit (relation = rel_name(t.relid),
                                       t.type, t.date)
                        where t.date > getdate - 2
```

Here we want to see everything done since two days before today's date.

```
      AUDIT
      RANGE 9 0 transact
                                  ROOT
                                  (0 0)
                                 /     \
                                /       \
                          RESDOM         GT
                          /    \        /  \
                         /      \   VAR 5   \
                     RESDOM   RELNAME (0 date) SUB
                      /  \        |           / \
                     /    \     VAR 6        /   \
                RESDOM    VAR 5 (0 relid) GETDATE INT1
                 /   \   (0 type)                   2
                /     \
            TLEND    VAR 5
                   (0 date)
      ENDOFCOMMAND
```

The data retrieved will be in various formats; an example return would look like:

```
FORMAT  8    CHAR     12    INT1 INT2 INT2 CHAR  20  INT2
TUPLE   5    parts  APPEND  10    3    nut 1000
TUPLE   5    parts  DELETE  12    4   bolt  10
FORMAT  5    CHAR     12    INT1 INT2 INT2
TUPLE   6   supply REP_OLD  12   100
TUPLE   6   supply REPLACE  12    50
```

This might be displayed to the user as:

```
    Relation "parts":
                        |number |name        |weight   |
         action         ------------------------------
             append     |     10|         nut|     1000|
             delete     |     12|        bolt|       10|
                        ------------------------------
    Relation "supply":
                       |pnum       |snum     |
          action       ---------------------
             old       |         12|      100|
             new       |         12|       50|
                       ---------------------
```

```
range of l is log5
audit into inv_audit (l.type, l.date, l.value)
        where l.relid = rel_id("inventory")
```

```
AUDIT_INTO
RANGE 5 0 log5
RANGE 10 2 inv_audit

                                      ROOT
                                     ( 2 0 )
                                     /     \
                                    /       \
                            RESDOM           EQ
                            /     \          /    \
                           /       \        /      \
                   RESDOM       VAR 5   VAR 6   CNVTRID
                   /     \    (0 type) (0 relid)    |
                  /       \                      CHAR 9
            RESDOM        VAR 5                (inventory)
            /     \      (0 data)
           /       \
      TLEND       VAR 6
                 (0 value)
ENDOFCOMMAND
```

This command will store in the relation "inv_audit" all the changes that were made to the relation "inventory".

ERROR MESSAGES:

Incorrect number of logs

One and only one variable can correspond to a log in the command.

Incorrect use of value or type attribute

The value field can only appear in the target list and no functions can be applied on it. If the value of a REPLACE record is requested, the audit may not be restricted to exclude REP_BEGIN or REP_OLD. If the REP_OLD is included REP_BEGIN must also be included. For an *audit into* command, the value field can be retrieved only if relid is restricted to a single value.

Bad log

The log does not match the currently open database.

7.5.5. BEGIN TRANSACTION

IDL SYNTAX:

begin transaction

OCTAL COMMAND CODE: 0324

DESCRIPTION:

The *begin transaction* command is given whenever multiple IDM commands are to be treated as a single transaction. Once a *begin transaction* command is issued, the only legal commands are:

abort
append
begin
delete
end
replace
reopen
retrieve
sync
execute (of stored commands and of programs containing those commands)

A transaction is aborted by issuing the *abort* command or in the event of a deadlock. A transaction is completed when an *end transaction* command is received. If multiple *begin transaction* commands are issued, the transaction is completed only when the same number of *end transaction* commands are received.

7.5.6. CLOSE

IDL SYNTAX:
    close

OCTAL COMMAND CODE: 023

DESCRIPTION:
The user issues a _close_ when she is finished processing within the database.

EXAMPLES:

```
close

EXITIDM
ENDOFCOMMAND
```

The above command caused the IDM to mark the fact that the user has completed processing within the current database. A new database may be opened, or the user can log off the system. Since the command must be associated with a DBIN (an open database) this command takes no parameters.

If only one dbin has a database open and a _close_ command is received, the database is automatically checkpointed.

A _close_ command terminates the present dbin. When another database is opened, another dbin will be created in the IDM.

ERROR MESSAGES:

No database is open.
If close is issued on DBIN 0.

7.5.7. CLOSE FILE

SYNTAX:

```
     closefile(filenum)
```

OCTAL COMMAND CODE: 360

DESCRIPTION:

The command close file is used in the IDM-provided random access file system to signify that the user has terminated processing that file. No more reads or writes to that file may be done without an "open file" command. The file number is what is returned by the "open file" command. Close file causes all changed data blocks of the file to be written to disk storage.

EXAMPLES:

```
closefile(1)

FILECLOSE 1
ENDOFCOMMAND
```

The above command closes the file opened as "1" in the currently active database.

7.5.8. COPY

IDL SYNTAX:

```
    copy [in | out] <database name> [ ( <object name>
          { , <object name> } ) ]
```

OCTAL COMMAND CODE: 354, 355

DESCRIPTION

Copy is the logical dump facility of the IDM. Copy out copies either all relations or the named relations to the host or to IDM tape in a format readable by copy in. If no relations are specified, all "user-relations" owned by the current user are copied. ("User-relations" are relations created by the user but not "system relations" or transaction logs.) The user must have read (resp. write) permission on all domains of a relation to use copy out (in) on it. The format of the output which copy out produces for each relation is: the relation name, followed by the attribute names, followed by a format description, followed by a set of TUPLE-value pairs, followed by a DONE token. Copy out uses the four-byte integer count which follows the DONE token to indicate the number of relations copied so far. It stops processing after the first error.

The general format of a copy in command is:

1. COPYIN
2. (copyin tree)
3. ENDOFCOMMAND
4. (information for first relation)
5. (information for second relation)

.
.
.

N. ENDOFCOMMAND

where the information for each relation is in the same format as produced by copy out. If the data is from tape, the ENDOFCOMMAND token following the last tuple information is not required. Copy in does not have to use the results of a copy out command as input; instead, a host program may be used to generate the appropriate input. If relation names are not specified in the command tree of the copy in command, then all of the relations in the following data are copied in. If relation names are specified, then only the specified relations are copied in. If a relation to be copied in exists in the database, the format information is checked and the data is appended, otherwise the relation is created. All of the relations created will be owned by the user.

To copy to and from IDM tape, see section 3.10.

EXAMPLES:

To archive some data the following command could be used.

```
open archive
copy out mydb(olddata, olderdata)


     DBOPEN 7 archive
     ENDOFCOMMAND

     COPYOUT 4 mydb
                       ROOT
                       (0 0)
                       /   \
                      /     \
                 RESDOM      QLEND
                  /  \
                 /    \
             RESDOM  CHAR 7
              /   \ (olddata)
             /     \
         TLEND  CHAR 9
              (olderdata)
     ENDOFCOMMAND
```

If the format of "olddata" and "olderdata" were:

```
olddata(num=i1, name=c20, field = uc2)
olderdata(num=i1, name=c20, field = uc2)
```

the output from the <u>copyout</u> would be:

```
CHAR 7 olddata
CHAR 3 num
CHAR 4 name
CHAR 5 field
FORMAT 5 INT1 CHAR 20 CHAR 2
TUPLE (tuple value in above format)
TUPLE (   "    )
  .
  .
  .
DONE (status = CONTINUE,
      number of relations copied so far,
      count of tuples)
CHAR 9 olderdata
CHAR 3 num
CHAR 4 name
CHAR 5 field
FORMAT 5 INT1 CHAR 20 CHAR 2
TUPLE (tuple value in above format)
TUPLE (   "    )
```

```
          .
          .
          .
        DONE (status != CONTINUE,
              number of relations copied,
              count of tuples)
```

To create or append to a relation olddata, the following command could be used.

```
open archive
copy in mynewdb(olddata)
```

```
DBOPEN 7 archive
ENDOFCOMMAND

COPYIN 7 mynewdb
                        ROOT
                        (0 0)
                       /    \
                      /      \
              RESDOM          QLEND
              /    \
             /      \
         TLEND   CHAR 7
               (olddata)
ENDOFCOMMAND

     Data for one or more relations including
     the olddata relation goes here.  It should
     be in exactly the same format as the output
     from a copy out command, such as the output
     from the above command.

ENDOFCOMMAND
```

ERROR MESSAGES:

<u>bad tree node</u>
  there are bad tokens in data

<u>database not found</u>
  The specified database was not found.

<u>not found</u>
  In copyout: specified relation was not found in the database. In copyin: specified relation was not found in the following data.

<u>bad format in copyin</u>
  attribute names and formats do not agree in number or data format does not agree with relation format.

7.5.9. CREATE

IDL SYNTAX:

```
    create <object name> ( <name> = <format> {
         , <name> = <format> } ) [ with <with-node-
         options> ]
```

OCTAL COMMAND CODE: 306

DESCRIPTION:

The create command sets up an empty relation in the database currently open. The create command contains several optional parameters:

Create with-node-options

Option	Number	Meaning
quota	(1)	specifies the maximum size of the relation, excluding index blocks. The default is no quota.
logging	(4)	specifies whether the transaction log is to be updated whenever this relation is updated.

The numbers specified above are the "with-node-option numbers" that appear in the translated command.

When the relation is initially created it is empty, and no indices exist for it. A clustered index should be created for it as soon as it has grown to several blocks of data.

EXAMPLES:

```
        create parts (name = c20, cost = bcd8, quan = i2)
                with logging, quota = 50
```

The above command sets up the relation "parts" with attributes "name" (a 20 character field), "cost" (an 8-digit BCD field), and "quan" (a two-byte integer field).

```
CREATE
RANGE 6 15 parts
                             ROOT
                            (15 0)
                            /     \
                           /       \
                     RESATTR 4     WITH
                      (name)       (4)
                   /        \      /   \
                 /        TYPE   QLEND  \
            RESATTR 4  (CHAR 20)       WITH
               (cost)                   (1)
             /      \                  /   \
            /       TYPE            INT1    \
        RESATTR 4  (BCD 8)          (50)    QLEND
          (quan)
        /      \
       /      TYPE
     TLEND   (INT2 2)
(partial tree)
ENDOFCOMMAND
```

The relation will be allowed to grow to a total of 50 data blocks, after which an error will be returned to the user if further additions take place.

The parameter "logging" tells the IDM to log all changes to the relation in the database transaction log.

ERROR MESSAGES:

<u>out of space on disk</u>
The relation could not be created because the database was full. A recommended action is to destroy unused relations, thus picking up extra space.

<u>tuple too long</u>
The smallest possible tuple (all variable width domains being zero length) would not fit on a 2-KB block. Tuples are limited to fit within one block, including overhead.

<u>permission denied</u>
Not all users have permission to create relations; this permission is given by the DBA.

<u>illegal command</u>
It is illegal to create relations inside a transaction or stored command.

7.5.10. CREATE DATABASE

IDL SYNTAX:

```
create database <name> [ with <with-node-options> ]
```

OCTAL COMMAND CODE: 313

DESCRIPTION:

The create database command sets up a database that is empty except for the system relations.

The number of 2-K blocks assigned to the database is specified with the create database command. Since database allocations are only made in whole numbers of zones, the number of blocks specified will be rounded up to the first whole number of zones, the allocation made, and the number of blocks actually allocated returned to the user. A zone is a group of cylinders: the precise number of cylinders per zone varies from disk to disk.

The database will not be allowed to grow beyond the size specified. An error will be returned to the user if the database attempts to grow beyond this size. If the database size is to be extended, the "extend" command should be used.

Create database with-node-options

demand	number of blocks to allocate	(5)
disk	disk to allocate to	(3)
ascii	create a database using ASCII character set	(13)
ebcdic	create a database using EBCDIC character set	(14)

"demand" specifies the desired size of the database. create database will attempt to allocate approximately the amount of space specified by "demand". "disk" specifies the disk(s) on which the database should be created. If the disk option follows the demand option "disk = <name>" can be abbreviated as "on <name>" "ascii" and "ebcdic" can be used to explicitly specify the character set that will be used to store character data in the database. The usual default is to store characters in ASCII. (See section 3.12)

Each option implies the default value of the other unless otherwis specified (demand defaults to one zone; disk defaults to any disk). Implied or specified, the demand option precedes the disk option to form a "demand-disk" option group.

This command is only executable from the system database. You must have create database permission in the system database to create a database.

EXAMPLES:

Disk space is allocated by whole zones. If the demand is equal to 150 blocks and the zone size on the disk is 180 blocks, the IDM wil allocate 180 blocks. The IDM allocates all of the space requested unless there is not enough free space on the disk. If there is not enough free space to meet the demand, the IDM will allocate as much of the demand as possible. If no demand is specified, the database will be created with one zone in size. A zone is between 128 and 254 blocks depending on the physical characteristics of the disk drive. The zone size of a disk can be found in the "bpz" attribute of the "disks" relation in the "system" database.

```
    create database burt with demand = 7500 on "disk1"
    or
    create database burt with demand = 7500, disk = "disk1"
```

The above command creates the database "burt" with a size limit of 7500 blocks. It will reside on the disk named "disk1". If there is no space on "disk1" the database will not be created.

```
    create database burt with disk = "disk1"
```

will create a database with the minimum demand of one zone on "disk1". If there iinsufficient space on "disk1", the database will not be created.

```
    create database burt with disk = "disk1", demand = 7500
```

This command will allocate one zone (default demand) from "disk1", then allocate the number of zones containing 7500 blocks on any drive at all.

This page has been intentionally left blank.

database will not be created.

```
DBCREATE
RANGE 5 0 burt
             ROOT
             (0 0)
            /    \
           /   WITH
     TLEND      (5)
               /   \
            INT2  WITH
           (7500)  (3)
                   / \
                  /   \
                 /     \
             CHAR 5   QLEND
             (disk1)
ENDOFCOMMAND
```

```
    create database db with demand = 1000
```

This will create a database with 1000 blocks on any available disk(s).

```
    create database test
           with demand = 2500 on "diska",
                demand = 2500 on "diskb"
```

This will create a database on the two disks "diska" and "diskb". If neither disk has any free space, the database will not be created.

If no demand is specified, the database will be created with one zone in size. A zone is between 128 and 254 blocks depending on the physical characteristics of the disk drive. The zone size of a disk can be found in the "bpz" attribute of the "disks" relation in the "system" database.

ERROR MESSAGES:

<u>permission denied</u>
A user must have permission from the system DBA to create a database and must have the system database open.

<u>already exists</u>
Database names must be unique.

<u>illegal command</u>
It is illegal to create a database inside a transaction

or stored command. It is also necessary to open the system database before creating a database.

7.5.11. CREATE FILE

IDL SYNTAX:

```
     create file <name> [ with <with-node-options>]
```

OCTAL COMMAND CODE: 336

DESCRIPTION:

The create file command is used in conjunction with the IDM's random access file system. It causes a file to be given the name in the create command. Files are always associated with databases, and are in the database the user had open when the file was created.

IDM random access files have no structure. They are simply byte streams that are addressable at any point. The "read file" and "write file" commands both specify the byte offset into the file that is to be read or written. Protection for files is specified the same as for relations, with separate "deny" and "permit" commands.

With-node-options for the "create file" command are the same as the "create" command, except that the logging option is not allowed. This means that a file is not guaranteed to have have correct data if a crash occurs. When a file is closed the changed data is forced to disk. This guarantees the file data is correct if a disk failure occurs after closing a file.

In order to manipulate the file, an "open file" command must be given after the "create file".

EXAMPLES:

```
        create file mine
```

The name of the file is sent as if it were a result variable.

```
FILECREATE
RANGE 5 0 mine
             ROOT
             (0 0)
             /   \
            /     \
       TLEND     QLEND

ENDOFCOMMAND
```

ERROR MESSAGES:

<u>out of space</u>
    no room for the allocation in the database

7.5.12. CREATE INDEX

IDL SYNTAX:

```
    create [ unique ] [ nonclustered | clustered ] index
         [ on ] <object name> ( <name> { , <name> } )
         [ with <with-node-options> ]
```

OCTAL COMMAND CODE: 310

DESCRIPTION:

Indices are used to provide direct access to data. A "clustered" index is one for which the data is physically in order. A nonclustered index is one that is created for an attribute or group of attributes for which the data is not clustered. The default is to create a clustered index.

Only one clustered index is allowed per relation (although it may specify up to 15 attributes). Up to 250 nonclustered indices are allowed per relation. If a nonclustered index is created that already exists, an error will be returned. (The old index should first be destroyed). If a clustered and a nonclustered index are created on the same attribute(s) the IDM will use the clustered index. When a clustered index is created all indices on that relation are destroyed including a clustered index on the same attribute if one existed. (The other indices are not destroyed if the clustered index is recreated using the with-node-option "recreate").

When a clustered index is created the data is sorted according to the attributes specified, then a B-tree index is created. When the relation is written in sorted form on the disk, the "fillfactor" parameter is used to specify how full to make the blocks. Fillfactors range from 1 (1% of the block is to be filled) to 100 (the block is to be completely filled.) Relations that are known to have a high potential for growth should have a small fillfactor specified so the data can be kept physically clustered for as long as possible. If a relation has become scattered (blocks that contain data that should be in sort order are spread over several cylinders) the DBA will know that fact because the I/O time will become large with respect to the average read time. Then the clustered index should be created again (the old one is automatically destroyed) and a new fillfactor specified.

A "unique" index is one for which the attribute values are unique; that is, no two of the attributes (or attribute combinations) are the same. Unique indices are provided by the IDM as a convenience for the users because there are data values, such as employee number, which for a given application must be unique. By creating a unique index for such attributes, the user makes the IDM enforce the fact that the data values must be unique. If at any time two attributes are assigned the same value--through a replace, copy or append command--if the attribute has a unique index on it, an error will be reported and the update will be aborted. If the unique index was created using the with-node-option "delete_dups", or if these updates are done while command option 6 is set, these updates delete the duplicate without generating an error. The user is notified of the deletion by an information only (not an error) message returned as the DUP bit in the DONE structure.

When a unique index is created, if there are existing tuples with duplicate values on the indexed attributes, the index will not be created and an error will be returned. However, if the "delete_dups" with-node-option is sent, the index will be created, one or more of the tuples will be deleted so that the index is unique and the DUP bit in the DONE structure will be set. Command option 6 has no effect at the time the index is created.

The "recreate" with-node-option may be used if the index already exists and empty data or index pages must be deallocated. With this option, the data will not be resorted. (This option may also be used on system relations in user data bases except for the relation and indices relations).

Create index with-node-options

delete_dups	as if not set, i.e. abort the update if duplicate unique key is detected	(7)
fillfactor	amount to fill disk blocks	(9)
skip	skip indicates how many blank blocks to leave between the data blocks	(10)
recreate	if index already exists, do not resort data	(12)

The options default to:

delete_dups	abort update if duplicate unique key
fillfactor	100% full
skip	0
recreate	false: sort or resort the data

<u>Root node bits for create index</u>

unique (bit 0)	if set, keys are maintained unique
clustered (bit 1)	if set, data is maintained in order on keys

The maximum size ("width") N of an index must satisfy the following inequalities:

This page has been intentionally left blank.

clustered index	N <= 252 bytes
nonclustered index	N <= 248 bytes

where N is the total width of all indexed attributes; if an attribute is compressed then add 1 byte per attribute. See Section 3.9, "Index Sizes", for a description of an algorithm for calculating the sizes of newly-created indices.

EXAMPLES:

```
create clustered index on parts (name, number)
        with fillfactor = 40
```

The above command causes the "parts" relation to be sorted on (name, number), written on the disk in blocks 40% full, and a B-tree index created for the (name, number) pairs. Now when an access is made that specifies (name) or (name, number), the access is direct; that is, only the index and the exact blocks needed are read, not the entire relation.

```
create unique nonclustered index on parts (number)
```

The "parts" relation already has a clustered index (from the above example); the nonclustered index on "number" is created in this example to facilitate access to the "parts" relation when "number" alone is specified. It is a "unique" index, so the IDM enforces that no two part numbers may ever be the same.

```
INDCREATE
RANGE 6 0 parts
                          ROOT
                         (0 01)
                         /    \
                        /      \
                  RESDOM     QLEND
                   /   \
                  /     \
            TLEND      VAR 7
                      (0 number)
ENDOFCOMMAND
```

ERROR MESSAGES:

not owner
: Only the user who created the relation can create an index on it.

index exists
: A nonclustered index of exactly the same characteristics exists.

<u>out of space</u>
The space for the index is counted in the space for the database.

<u>index too large</u>
The size of an index exceeds the maximum size allowed.

<u>system relation</u>
New indices cannot be created on system relations.

<u>index does not exist</u>
When using the with-node-option "recreate", the index must already exist.

7.5.13. CREATE VIEW

IDL SYNTAX:

```
     create view <object name> ( <target list> )
          [ where <qualification> ]
```

OCTAL COMMAND CODE: 317

DESCRIPTION:

The create view command is used to set up a virtual relation, which is one that is never a physical entity, but is composed of parts of one or more relations (called the base relations), or other views. Views may be protected and destroyed as other relations; they may be updated if the update can unambiguously be applied to one of the base relations (see Section 3.7).

EXAMPLES:

```
         range of p is parts
         range of pr is products
         create view mine (p.name, p.cost, pr.quan)
                 where pr.name = "TV"
                 and pr.part = p.name
```

The above view is called "mine" and consists of the names and costs of all parts that are used to make TV´s, as well as the number of parts required ("quan") from the "products" relation.

```
VIEW
RANGE 6 0 parts
RANGE 9 1 products
RANGE 5 2 mine
                             ROOT
                             (2 0)
                            /        \
                          /           \
                     RESATTR 4         AND
                      (name)            /    \
                   /        \          /      \
                 /        VAR 5     EQ        EQ
            RESATTR 4   (0 name)    /\        |  \
               (cost)              /  \       |    \
             /      \          VAR 5  CHAR 2  VAR 5 VAR 5
            /       VAR 5     (1 name)  (TV) (1 part)  (0 name)
        RESATTR 4  (0 cost)
          (quan)
        /      \
       TLEND    VAR 5
               (1 quan)

ENDOFCOMMAND
```

ERROR MESSAGES:

permission denied
    The user does not have read permission on the relations used to create the view.

7.5.14. DEFINE, DEFINE PROGRAM

IDL SYNTAX:

```
    define < name> <command> { <command> } end define

    /* only for embedded language systems */
    define program <object name> <command> { <command> }
        end define
```

OCTAL COMMAND CODE:

```
        DEFINE:   341
        DEFINEP:  356
```

DESCRIPTION:

The define statement defines a stored command. The <command> may consist of any of:

retrieve
append
replace
delete
begin transaction
end transaction

The command may have parameters in any place a constant would normally be acceptable. If multiple commands are in the define there should be only one ENDOFCOMMAND. The define program command is for use within programs and returns a 4-byte number in the count field of the done block with which to refer to the stored command. The name provided is the program name in which the queries are being generated. Each define program with that program name will be associated with, and physically clustered near, other define program commands in the same program name. A null define program command may be issued to verify and insert the name in the relation catalogue prior to defining any commands. It is an error to issue a null define program command for a program which already exists. It is up to the host system to destroy a stored program prior to prior to recompiling the host program which contains a given stored program.

A stored command or program created by the DBA is accessible to all users who have permission to use it. A non DBA user can define a stored command or program with the same name as one defined by the DBA. If a non owner of that command wants to use it, he must invoke it as "command name":"owner name". For the ones created by the DBA, the "owner name" does not have to be mentioned.

Command-options sent with definitions are in effect when the command is defined. If command-option 15 is set, the command-options will be used at the time the command is actually executed. Additional command-options may be sent to the IDM at execute time, and apply to all commands inside the stored command or program.

A define or define program command will be logged if all the relation and/or views used by the command are also logged.

EXAMPLES:

```
define additem
range of i is items
append to items (name = $name, number = $num)
retrieve (count =
        count(i.number where i.number = $num))
end define
```

```
DEFINE 7 additem
APPEND
RANGE 6 2 items
                     ROOT
                     (2 0)
                    /    \
                   /   QLEND
             RESATTR 4
              (name)
              /     \
         RESATTR 6   \
          (number)  PARAM 4
        /       \    (name)
     TLEND    PARAM 3
               (num)
RETRIEVE
RANGE 6 2 items
                     ROOT
                     (0 0)
                    /    \
                   /    QLEND
               RESATTR 5
                (count)
             /        \
            /       AGGHEAD
       TLEND      /      \
                 /        \
            AOPCNT         EQ
              |           / \
            VAR 7        /   \
         (2 number)  VAR 7   PARAM 3
                  (2 number)  (num)
ENDOFCOMMAND
```

ERROR MESSAGES:

<u>already exists</u>
A relation, file, view or stored command has the name given. All named objects must be unique for each user.

<u>Stored command or program too big</u>.
The internal representation of a stored command occupies more than approximately 8KB, or there are more than 65000 commands associated with a stored program.

7.5.15. DELETE

IDL SYNTAX:
```
     delete <variable> [ where <qualification> ]
```

OCTAL COMMAND CODE: 304

DESCRIPTION:
The delete command is used to remove one or more tuples from a relation. The user must have write permission for the relation.

EXAMPLES:

```
          range of p is parts
          range of pr is products
          delete p where p.name = pr.part and pr.name = "TV"
```

The above command deletes all parts where the part is used in making a TV.

```
          DELETE
          RANGE 6 5 parts
          RANGE 9 2 products
                              ROOT
                              (5 0)
                             /     \
                            /       \
                      TLEND          AND
                                    /    \
                                   /      \
                                 EQ        EQ
                                /  \      /  \
                               / VAR 5  VAR 5  \
                          VAR 5 (2 part) (2 name) CHAR 2
                        (5 name)                   (TV)
          ENDOFCOMMAND
```

```
          delete p
```

The above command deletes every part in the "parts" relation.

ERROR MESSAGES:

permission denied
The user does not have write permission for the relation.

7.5.16. DENY

IDL SYNTAX:

```
     deny  <protect mode>  [on | of] <object name>
          [ ( <attlist> ) ] [ to <user> {, <user> }]
```

OCTAL COMMAND CODE: 316

DESCRIPTION:

The deny command is used to deny access permission to users. The <object name> specified may be a relation, file, view, or stored command. The <user>'s may be user names or group names. A group is any entry in the users catalogue which has the uid field equal to the gid field. If there are no users specified, the protection applies to everyone. The <protect mode>'s are listed below. Read and write apply to relations, views and files. "All" denies both read and write. Execute applies only to stored commands. A deny command which contradicts previous permit commands will take precedence. Only the owner of an object or the DBA may deny permissions. The DBA may also deny rights to use the create, create index, tape read, tape write, and create database commands.

The <protect mode> is passed in the second byte of the root node of the query tree. The following modes are acceptable:

mode	octal code	
read	1	
write	2	
all	3	(read and write)
read tape	4	
write tape	10	
all tape	14	(read and write)
execute	340	
create	306	
create index	310	
create database	313	

The way to protect ranges of values (access to employee records with salaries less than 1000, for instance) is to deny access to the relation and then create a view which restricts the relation. Similarly updates can be selectively allowed by defining the exact type of update in a stored command. Views and stored commands (created by the owner of the relations affected) do not check permissions, on the affected relations, when they are referenced; however, views and stored commands defined by other than the owner of the affected (base) relations do check permissions on the base relations when they are referenced.

This page has been intentionally left blank.

EXAMPLES:

```
permit read of parts
deny read of parts to george, harvey, mary
```

The above commands say that the "parts" relation cannot be read by the users george, harvey, or mary but everyone else may.

```
deny write of parts(descript) to clerks
```

The entire group "clerks" is denied write permission on the description field of the "parts" relation. Anyone in the "clerks" group who previously had permission no longer does. "Clerks" must be defined as a group in the <u>users</u> relation.

```
DENY
RANGE 6 5 parts
                       ROOT
                       (5 2)
                       /   \
                      /     \
               RESDOM        QUALDOM
               /    \        /     \
              /      \    CHAR 6    QLEND
         TLEND    CHAR 8  (clerks)
                (descript)
ENDOFCOMMAND
```

A deny to <u>all</u> deletes previous permissions:

```
permit read of parts to bob
deny read of parts
```

Bob does not have permission to read "parts". The statements in the opposite order would give read permission to bob only. When a relation is created protection defaults to:

```
deny all of relation
permit all of relation to owner
permit read on tape to bob
```

ERROR MESSAGES:

<u>user not found</u>
The user name is put in the "users" system relation by the DBA. This error message means that the user specified is not in the "users" relation for this database.

<u>not owner</u>
Only the owner of an object or the dba may grant permissions on it.

7.5.17. DESTROY

IDL SYNTAX:

```
destroy <object name> {, <object name> }
destroy ( <target list> ) [ where <qualification> ]
```

OCTAL COMMAND CODE: 307

DESCRIPTION:

The destroy command is used to eliminate relations, files, views and stored commands. The entire object is removed from the system, and its space is freed for use within the current database. Only the owner or the database administrator can destroy an object. If there are views or stored commands which depend on the relation or the view they must be destroyed first.

EXAMPLES:

The destroy command has two forms. The first simply specifies the object name:

```
destroy parts, products
```

The above command destroyed the "parts" and "products" relations.

```
DESTROY
                        ROOT
                        (0 0)
                        /   \
                       /     \
                 RESDOM    QLEND
                  /    \
                 /      \
          RESDOM    CHAR 5
          /    \    (parts)
         /      \
    TLEND   CHAR 8
           (products)
ENDOFCOMMAND
```

In the second form the relation names are specified through the use of a target list:

```
range of r is relation
destroy (r.name) where (r.owner = userid)
```

```
DESTROY
RANGE 9 0 relation
                              ROOT
                             (0 0)
                             /     \
                            /       \
                      RESDOM          EQ
                      /    \         /  \
                     /      \       /    \
                TLEND    VAR 5    VAR 6     USERID
                        (0 name) (0 owner)
ENDOFCOMMAND
```

The above command makes use of the system relation named "relation". The "relation" relation contains information about each relation in the database, including the relation names and owners. The "destroy" above destroyed all relations owned by the user.

This form of the destroy command will return the number of relations actually destroyed in the "count" field of the DONE token.

The DBA of a database is allowed to destroy anything. The following example will destroy all relations, views, files and commands starting with "temp" regardless of their owner:

```
range of r is relation
range of u is users
destroy (name=concat(r.name, concat(":",u.name)))
        where r.name = "temp*"
        and r.owner = u.id
```

ERROR MESSAGES:

not owner
: Only the owner or DBA may destroy an object.

has dependencies
: There are dependent objects which must be destroyed first.

is open
: An object which is being accessed may not be destroyed, and someone is currently using the object.

7.5.18. DESTROY DATABASE

IDL SYNTAX:
    destroy database <name> { , <name> }

OCTAL COMMAND CODE: 314

DESCRIPTION:
    The database specified is completely eliminated from the system; the space is freed for use by other databases. All relations and files in the database are destroyed.

EXAMPLES:

```
        destroy database inventory



                DBDESTROY
                          ROOT
                         (0 0)
                          /   \
                         /   QLEND
                     RESDOM
                     /    \
                 TLEND CHAR 9
                        (inventory)
                ENDOFCOMMAND
```

The above command results in the destruction of the database "inventory". All disk space allocated to it is made free space.

ERROR MESSAGES:

not owner
    Only the DBA of the database or the system database can destroy it.

is open
    Someone is using the database.

7.5.19. DESTROY INDEX

IDL SYNTAX:

```
    destroy [ nonclustered | clustered ] index
         [ on ] <object name> ( <name> { , <name> } )
```

OCTAL COMMAND CODE: 0311

DESCRIPTION:

The destroy index command is used to remove an index from the system. This is done for two reasons: first, because the index is unnecessary, and its space is needed for other relati^ns; second, because the index is not used often enough to justify having to pay the overhead of updating it whenever the tuple attributes that it indexes are updated. The index is identified by being either clustered or non-clustered and by the keys which it indexes.

EXAMPLES:

```
        destroy clustered index on parts (name, number)
```

The above command destroyed the index for (name, number). Initially the relation remains sorted on (name, number). New data will be appended at the end of the relation.

```
INDDESTROY
RANGE 6 15 parts
                   ROOT
                  (15 2)
                 /     \
                /    QLEND
            RESDOM
             /   \
            /     \
       RESDOM    VAR 5
        /   \   (15 name)
       /     \
   TLEND    VAR 7
         (15 number)
ENDOFCOMMAND
```

```
        destroy nonclustered index on parts (number)
```

The above command destroyed the nonclustered index on the "number" attribute of the "parts" relation.

```
INDDESTROY
RANGE 6 15 parts
                        ROOT
                       (15 0)
                       /    \
                      /    QLEND
                 RESDOM
                  /    \
                 /      \
            TLEND      VAR 7
                     (15 number)
ENDOFCOMMAND
```

ERROR MESSAGES:

<u>not owner</u>

Only the owner of the relation or the DBA for the database can destroy indices for that relation.

7.5.20. DUMP DATABASE

IDL SYNTAX:

```
    dump database <name> [ to  [ file ] <name> ] with tran-
         saction [ to  [ log ] <object name> ]
```

OCTAL COMMAND CODE: 344

DESCRIPTION:

The dump database command is the "physical" dump command. It is accompanied by a mandatory transaction log dump. The transaction log for the database is first dumped to the front end system, to a log in the currently open database, or to tape; then the database is dumped to the front end system, to a file in the currently open database, or to tape. The file <name> and the log <object name> should not already exist in the open database. They will be created automatically. If both the database and the transaction log are to be dumped to the front end system, the transaction log will be dumped first followed by an EOR bit, then the database will be dumped followed by an EOR bit.

The dump database command does not allow the database to be active during the dump. If the database being dumped is already open, the dump command is not allowed. During the dump, the database is locked and noone can open it. If the database or the transaction log are to be dumped to tape, the tape command-option should be used. See section 3.10 for a description. The database may be reloaded to any disk.

Dump database with-node-options (implied)

file	dump the database to a file (6)
log	dump the transaction log to a log (11)

The protocol between the host computer and the IDM is the following:

HOST	IDM
DUMPDB command tree ENDOFCOMMAND	
	DONE token translog + EOR (the above is sent only if dump of transaction log is to go to the host)
ENDOFCOMMAND (this acknowledgment of receipt is sent by the host only if the dump of the transaction log is to go to the host)	
	DONE token (the DONE token is always sent)
	database + EOR (this is sent only if the dump of the database is to go to the host)

The EOR bit is sent in the count field of the last packet that is sent to the host (see section 6.3.4).

EXAMPLES:

dump database inventory with transaction

```
DUMPDB 9 inventory
           ROOT
           (0 0)
          /    \
         /      \
    TLEND        QLEND
ENDOFCOMMAND
```

The above command causes the transaction log to be read, truncated, and sent to the front-end system. The entire database (without the transaction log) is then read and sent to the front-end system.

```
    open backup
    dump database inventory to Jan10 with transaction
```

```
DBOPEN 6 backup
ENDOFCOMMAND
DUMPDB 9 inventory
             ROOT
             (0 0)
            /    \
           /      \
       TLEND    WITH
                (6)
               /    \
              /      \
           CHAR 5   QLEND
           (Jan10)
ENDOFCOMMAND
```

The transaction log for "inventory" is sent to the front-end system. The database "inventory" is then written to the file "Jan10" in the database "backup".

```
    open backup
    dump database inventory with transaction to trlog17
```

```
DBOPEN 6 backup
ENDOFCOMMAND
DUMPDB 9 inventory
             ROOT
             (0 0)
            /    \
           /      \
      TLEND        WITH
                   (11)
                  /    \
                 /      \
             CHAR 7     QLEND
           (trlog17)
ENDOFCOMMAND
```

The transaction log is written to the new log "trlog17" in database "backup" and the database is written to the front-end system.

```
    open backup
    dump database inventory to Jan10 with transaction
            to trlog17
```

```
          DBOPEN 6 backup
          ENDOFCOMMAND
          DUMPDB 9 inventory
                            ROOT
                           (0 0)
                          /     \
                         /       \
                    TLEND         WITH
                                  (11)
                                 /    \
                                /      \
                       CHAR 7          WITH
                     (trlog17)         (6)
                                      /    \
                                     /      \
                              CHAR 5          QLEND
                             (Jan10)
          ENDOFCOMMAND
```

The transaction log for "inventory" is dumped to the log "trlog17" in "backup" and the database "inventory" is dumped to the file "Jan10" in "backup".

ERROR MESSAGES:

out of space
: There must be enough space in the database/file specified to hold the entire database.

does not exist
: The database does not exist.

db not accessible
: The database is already being dumped.

file exists
: The file already exists in the open database.

log exists
: The log already exists in the open database.

no free pages
: No free pages in the working database (if the database or transaction are being dumped to the working database).

permission denied
: The user issuing the dump command must be dba of the database (s)he wants dumped.

7.5.21. DUMP DISK

Section removed.

7.5.22. DUMP TRANSACTION

IDL SYNTAX:

dump transaction <database name> <object name> [to <object name>]

OCTAL COMMAND CODE: 353

DESCRIPTION:

The dump transaction command causes most of the contents of the transaction log for the named database to be written to another log in the currently open database, to the front-end system, or to tape and truncated. The complete contents are not always written because there may be transactions in progress. If the <object name> is the name of a log (i.e. the result of a previous dump transaction or load transaction command), the entire log is dumped to another log, to the front-end system, or to tape. If a new log is specified, it should not exist in the currently open database. The new log will be created automatically. If the log is to be dumped to tape, the tape command-option should be used. See section 3.10 for a description.

The dump transaction command DOES allow a database to be active during the dump. The dump will proceed if the database is active and users may open the database while the transaction dump is in progress.

The transaction log should be periodically dumped for two reasons: first, to free up the space it is taking up in the database; second, to use the dumped log as a part of a backup strategy.

Dump transaction with-node-options (implied)

log	dump the transaction log to a log (11)

The protocol between the host computer and the IDM is the following:

HOST	IDM
DUMPXACT command tree ENDOFCOMMAND	
	DONE token (the DONE token is always sent)
	translog + EOR (this is sent only if dump of transaction log is to go to the host)
ENDOFCOMMAND (this acknowledgment of receipt is sent by the host only if the dump of the transaction log is to go to the host)	

The EOR bit is sent in the count field of the last packet that is sent to the host (see sect. 6.3.4).

EXAMPLES:

```
     dump transaction personnel transact

              DUMPXACT 9 personnel
              RANGE 9 0 transact
                           ROOT
                          (0 0)
                          /   \
                         /     \
                    TLEND       QLEND
              ENDOFCOMMAND
```

The transaction log is emptied of all but the active transactions, and written to the front end system.

```
     open backup
     dump transaction personnel transact to Maytrans
```

```
          DBOPEN 6 backup
          ENDOFCOMMAND
          DUMPXACT 9 personnel
          RANGE 9 0 transact
                       ROOT
                       (0 0)
                      /     \
                     /       \
               TLEND          WITH
                              (11)
                             /    \
                            /      \
                     CHAR 8         QLEND
                   (Maytrans)
          ENDOFCOMMAND
```

The transaction log is emptied of all but the active transactions, and written to the log "Maytrans".

```
     open backup
     dump transaction personnel trlog5 to Maytrans
```

```
          DBOPEN 6 backup
          ENDOFCOMMAND
          DUMPXACT 9 personnel
          RANGE 7 0 trlog5
                       ROOT
                       (0 0)
                      /     \
                     /       \
               TLEND          WITH
                              (11)
                             /    \
                            /      \
                     CHAR 8         QLEND
                   (Maytrans)
          ENDOFCOMMAND
```

The log "trlog5" in the database "personnel" is copied to the log "Maytrans" in the database "backup".

ERROR MESSAGES:

out of space
    There must be enough space in the database/file specified to hold the entire log.

does not exist
    The database does not exist.

log exists

The log already exists in the open database.

7.5.23. END TRANSACTION

IDL SYNTAX:

```
     end transaction
```

OCTAL COMMAND CODE: 325

DESCRIPTION:

The end transaction command is given whenever a set of commands that began with a "begin transaction" is complete, and the user wishes to make the results of the transaction known to the rest of the system.

7.5.24. EXECUTE, EXECUTE PROGRAM

IDL SYNTAX:

```
[execute] <query name> [ [ with ]  [ ( ]
    [ name = ] <constant> { , [ name = ] <constant> }  [ ) ] ]
```

OCTAL COMMAND CODE: 340, 370

DESCRIPTION:

The *execute* command executes a stored command. When unambiguous, the *execute* key word may be omitted. The parameters are passed as a linear list. If the names are given, the values are substituted for the named parameters. If the parameters are not named then the parameters are substituted for the name list which is sorted in alphabetic order. If more than one command is executed by the *execute* command each command will generate a DONE block. All but the last DONE block will have the CONTINUE bit set in its status field. The *execute program* command (EXECP) is used within programs which contain embedded IDL statements. The EXECP command is followed by the 4–byte number returned by the *define program* command.

PCHAR parameters may be sent only if they do not appear in the target list of a *retrieve* or *append* command inside the definition.

When parameters are not named, attention should be paid to the way parameter names are sorted. They are sorted as ASCII character strings, so that, for example, "2" sorts after "10". If the user wants to use ten parameters, they should be referred to as "01", "02', ..., "09', and "10'.

When argument names are used in the *execute* sequence, as in the first example below, the actual arguments are matched with the corresponding formal arguments in the definition, regardless of how the names of the formal arguments are sorted, and regardless of the order of the arguments in the *execute* sequence.

EXAMPLES:

```
execute update with name = "mike", amount = 44
```

Here the parameters are explicitly identified.

```
EXEC 6 update
PARAM 4 name
CHAR 4 mike
PARAM 5 amount
INT1 44
ENDOFCOMMAND

help "relation"
```

```
EXEC 4 help
CHAR 8 relation
OPTION 2 1 2
ENDOFCOMMAND
```

The second example requests that both format and name information is returned before the data.

From a program a command might be:

```
EXECP 2112001
CHAR 6 foobar
INT1 7
ENDOFCOMMAND
```

ERROR MESSAGES:

not found
The command was not found in the current database.

parameter not found
A parameter was sent which was not in the stored command.

too many parameters

7.5.25. EXTEND

Section Deleted.

7.5.26. EXTEND DATABASE

IDL SYNTAX:

```
    extend database <name> with <with-node-options>
```

OCTAL COMMAND CODE: 0327

DESCRIPTION:

The extend database command is used to increase or decrease the allocation for a database. The <with-node-options> accepted are as in the create database command, except that the demand option may be negative. The rules for positive allocation are the same as for the create database command. The extend database command can only be executed from the system database.

To decrease the allocation for a database, a negative value is given with the demand option. Since deallocation is only done for whole zones, the number of blocks specified will be rounded up to the next multiple of the number of blocks per zone. The number of blocks per zone varies between disks.

Only entirely freeable zones will be removed from the database. A freeable zone contains no pages which are either used or demanded. If there are not enough freeable zones in the database to satisfy the number of blocks demanded, all freeable zones will be deallocated. In any case, the actual number of blocks which are deallocated will be returned to the user.

If a disk option is also given with a negative demand option, the deallocation will only be done from freeable zones on the specified disk(s). If no disk option is given, the deallocation will be from zones which belong to the database on any disks.

EXAMPLES:

```
        extend database test with demand = -2000
```

This command will remove 2000 blocks from the database "test". If there are not enough freeable zones in "test", all freeable zones will be removed from the database.

```
        extend database accounts
               with demand = -3500 on "diska" on "diskb",
                    demand = -1500 on "diskc"
```

This command will remove from the "accounts" database a total of 3500 blocks between "diska" and "diskb", and 1500 blocks on "diskc".

It is also possible to include extend and deextends in the same command. The negative demands (deextends) will be processed first.

ERROR MESSAGES:

illegal command
: The user must be in the system database to execute this command.

permission denied
: Only the owner of this database can extend the database.

Database is open and cannot be locked
: When decreasing the size of a database, no other user may have the database open.

7.5.27. LOAD DATABASE

IDL SYNTAX:

```
     load database <name> [ from [file] <name> ]
```

OCTAL COMMAND CODE: 345

DESCRIPTION:

The load database command causes a physical load of the database. The database must already exist. It is then overwritten by the contents of the specified file or the data stream arriving from the host or from tape. The input must have been obtained by an earlier dump of the database.

The load database command will not compact data. To reorganize a database and reclaim space, copy in and copy out should be used. Note, however, that copy in/out will not handle files or stored commands.

If the IDM fails during a load database, the database is marked as inaccessible. The database must be destroyed and recreated. A message identifying this condition will be printed by recover and when someone tries to open the database.

Load database requires that the host computer correctly supply data unaltered. The IDM checks input from the host for reasonable consistency but it is possible that an incorrect host program could supply "load database" with bad data which will result in the IDM code failing or the data being loaded incorrectly.

If the database is to be loaded from tape, the tape command-option should be used. See section 3.10 for a description.

Load database with-node-options (implied)

file	load the database from a file (6)

The protocol between the host computer and the IDM is the following:

```
        HOST                              IDM

        LOADDB command
        tree
        ENDOFCOMMAND
        (above is always sent)

        database
        (above is sent only if
        input is to be received
        from the host)

                                          DONE token
```

EXAMPLES:

```
        load database inventory

                LOADDB 9 inventory
                          ROOT
                          (0 0)
                          /   \
                         /     \
                     TLEND      QLEND
                ENDOFCOMMAND
```

This command loads the database "inventory" from a file on the front-end system. The file on the host must be the result of an earlier dump of "inventory".

```
        open backup
        load database inventory from May6

                DBOPEN 6 backup
                ENDOFCOMMAND
                LOADDB 9 inventory
                          ROOT
                          (0 0)
                          /   \
                         /     \
                     TLEND    WITH
                               (6)
                              /   \
                             /     \
                        CHAR 4   QLEND
                        (May6)
                ENDOFCOMMAND
```

This command presumes that the database "inventory" was previously dumped to the "backup" database, file "May6", with a dump database command. If the load is being done to recover from a loss of data, the transaction logs that were created after the "May6" dump should be applied with "rollforward" commands.

ERROR MESSAGES:

wrong format
The named file or data from the host is not a dump of a database.

does not exist
The database does not currently exist.

out of space
The space currently occupied by the database is not sufficient to hold the earlier version of the database. The new database must have at least as many blocks as its older version whether or not the blocks were used on the older version.

corrupt input file
The input file is not in the proper format.

7.5.28. LOAD DISK

Section removed.

7.5.29. LOAD TRANSACTION

IDL SYNTAX:

```
     load transaction <database name>  <object name>  [ from
          <object name> ]
```

DESCRIPTION

This command is used to load or move a copy of the transaction log. The <database name> is the name of the database to which the log should be loaded. The log is loaded to the named relation, either from a relation in the currently opened database, from the host, or from tape. The source must be the result of a dump transaction or dump database commands (dump database always forces a dump transaction).

Load transaction requires that the host computer correctly supply data unaltered. The IDM checks input from the host for reasonable consistency but it is possible that an incorrect host program could supply "load transaction" with bad data which will result in the IDM code failing or the data being loaded incorrectly.

If the log is to be loaded from tape, the tape command-option should be used. See section 3.10 for a description.

Load transaction with-node-options (implied)

log	load the transaction log from a log (11)

The protocol between the host computer and the IDM is the following:

HOST	IDM
LOADXACT command tree ENDOFCOMMAND (above is always sent)	
transaction log (above is sent only if input is to be received from the host)	
	DONE token

EXAMPLES:

```
        load transaction inventory May

                LOADXACT 9 inventory
                RANGE 4 0 May
                            ROOT
                            (0 0)
                           /     \
                          /       \
                     TLEND         QLEND
                ENDOFCOMMAND
```

This command will load a transaction log from the front-end system and will store it in the log "May" in the "inventory" database. The log "May" should not exist beforehand and will be created by the command.

```
        open backup
        load transaction inventory June from invjune

                DBOPEN 6 backup
                ENDOFCOMMAND
                LOADXACT 9 inventory
                RANGE 5 0 June
                            ROOT
                            (0 0)
                           /    \
                          /      \
                     TLEND        WITH
                                  (11)
                                 /    \
                                /      \
                          CHAR 7     QLEND
                        (invjune)
                ENDOFCOMMAND
```

ERROR MESSAGES:

out of space
: There must be enough space in the named database/log to hold the input.

does not exist
: The named database does not exist.

log exists
: The named log already exists.

illegal source log
: The object being copied is not a transaction log.

This page has been intentionally left blank.

7.5.29A. NEW PASSWORD

IDL SYNTAX

newpassword <new password>

OCTAL COMMAND CODE: 346

DESCRIPTION:

The new password command is used to change the password of a user in the login relation of the system database. It allows users to change their passwords without being allowed access to the system database. This command can only be used if there is an existing login tuple for the user. The new password command must be sent from DBIN 0. It must be preceded by the old password for the user. If the host computer sends user names (hunames) rather than numbers (huids) to identify users on the host, the command must also be preceded by the huname of the user.

EXAMPLES:

From a host which sends host user ids (huids) to identify users the host system would send to the IDM using DBIN 0:

```
PASSWORD 11 oldpassword
NEWPASSWORD 11 newpassword
ENDOFCOMMAND
```

From a host which sends host user names (hunames) to identify users the host system would send to the IDM using DBIN 0:

```
HUNAME 4 jack
PASSWORD 11 anypassword
NEWPASSWORD 14 betterpassword
ENDOFCOMMAND
```

ERROR MESSAGES:

permission denied

The user was not found in the login relation of the system database, or did not send the correct old password before the new password.

unknown command

The command was sent to a dbin other than 0.

7.5.30. OPEN

IDL SYNTAX

```
     open <database name>
```

OCTAL TOKEN NUMBER: 342

DESCRIPTION:

The _open_ command is used to open a database for activity. In addition the _open_ returns a database instantiation number (DBIN) in the done block. The DBIN is used to identify the messages between the user and the IDM. The user sends the first message on DBIN 0 and it must be an _open_ command. An _open_ command on a non-zero DBIN closes the current database and opens a new one. The DBIN is transparent to the IDL user (i.e., a parser may be written which hides dbin from the user).

EXAMPLES:

At the beginning of an interactive session a user would type:

```
open mydata
```

The host system would write to the IDM using DBIN 0:

```
DBOPEN 6 mydata
ENDOFCOMMAND
```

The IDM would return a DONE block with a DBIN for future commands:

```
DONE statword dbin unused
```

If the user in the same session typed:

```
open demo
```

The host system would send the command on the current DBIN. This would close "mydata" and open "demo".

ERROR MESSAGES:

not found

The database does not exist

database is locked.

A _physical dump_, load, _rollforward_, _create_ or _destroy_ is in progress on the database.

7.5.31. OPEN FILE

SYNTAX

```
filenumber = openfile(<mode>, <file name>)
```

DESCRIPTION

The IDM file facility is not defined as part of IDL. Open file should only be used from a host program. The <mode> determines if the file is to be opened for read, write or both. The mode is sent after the FILEOPEN token and argument byte count. The file is looked up in the current database. The user must have the proper permissions to do the open. The filenumber is returned as a two byte integer in the DONE token and is used to identify the file in read file and write file commands. There is a limit of 20 open files per DBIN.

	Open file modes
read	0
write	1
read and write	2

EXAMPLES

```
fn = openfile("read", "textfile")

FILEOPEN 9 0 textfile
ENDOFCOMMAND
```

ERROR MESSAGES:

permission denied
You cannot read or write this file.

not a file
The named object is not a file.

illegal mode
The mode of the open was not 0, 1 or 2.

too many open files
There is a limit of 20 open files per DBIN.

7.5.33. PERMIT

IDL SYNTAX:

```
permit <protect mode> [on | of] <object name>
     [ ( <attlist> ) ] [ to <user> {, <user> } ]
```

OCTAL COMMAND CODE: 315

DESCRIPTION:

The permit command is a protection control command. It allows access to a specific user or a group. The user names and groups are recorded in the "users" relation by the DBA. If no names are specified the permission applies to everyone. All conflicting deny tuples are removed. The <object name> may be a relation, view, file or stored command. Read, write or all may be specified for the <protect mode> of relations, views, files and IDM tape drives. Execute must be specified for stored commands. Relations, views, files and stored commands default to no access allowed by anyone except the owner. To allow others to access a relation, the permit command must be used.

The database administrator may also grant permission to use the create, create index and create database commands. These default to denying permission. The "create" permission applies to relations, views, files and transaction logs.

EXAMPLES:

```
permit read of parts to george
```

The "parts" relation can be read by the user "george".

```
deny execute of getsum
permit execute of getsum to managers
permit execute of getsum to dave
```

Dave and all users in the group "managers" are allowed to execute the stored query "getsum". To allow all others to create relations the dba must issue: |

```
permit create
```

To permit dave to read from an IDM tape drive:

```
permit read on tape to dave
```

ERROR MESSAGES:

unknown user

The "users" relation for the currently open database must include the user (or group) specified.

not owner

Only the owner or DBA may grant permissions on an object.

SEE ALSO: DENY 7.5.16.

This page has been intentionally left blank.

7.5.32A PLAN

IDL SYNTAX

```
range of <variable> is <object name> [with
     [dindex=<index id>,] dorder=<order number>]
```

OCTAL COMMAND CODE: 373

DESCRIPTION

The plan command associates a user plan with the variable number of a relation. In IDL the user plan is specified on the RANGE statement but the host system sends the information to the IDM on a separate PLAN statement.

EXAMPLES:

```
range of p is products with dindex=1, dorder=1
range of s is suppliers with dorder=3
range of r is requests
```

The host system would generate RANGE statements of the form:

```
RANGE 9 0 products
RANGE 10 1 suppliers
RANGE 9 2 requests
```

and would follow them with the PLAN statements:

```
PLAN 0 1 1
PLAN 1 -1 3
```

The first PLAN statement tells the IDM to use the first nonclustered index as an access path to the relation products and to process it first. The second PLAN statement tells the IDM to do a relation scan of the relati^n suppliers and to process it secondly. Notice that no plan was specified for the relation requests. The IDM will decide what access path to use to access requests.

ERROR MESSAGES:

There are no error messages form the IDM for the plan statements, because the IDM never sees the plan statements independent of an IDM command. The IDM will send error messages when the plan declarations are actually sent if they are in error.

Range number is too big

Range variable numbers must be 0 - 15

Bad order number supplied for a variable

-1 was sent as the order number. It can be any number other than -1.

Bad index id supplied for a variable
An index id was supplied that either does not exist or is not "useful" for that variable. "Useful" means that the attributes that are referenced in simple clauses or join clauses, match the key attributes in the index.

This page has been intentionally left blank

7.5.33. RANGE

IDL SYNTAX:

range of <variable> is <object name> [with minlocks]

OCTAL COMMAND CODE: 343

DESCRIPTION:

The *range* command associates a variable name with the name of a relation or view. Most IDM commands require the range variable, not the actual relation name. Typically, the host program will keep the range table in a least recently used order and remember the range statements between IDL commands. The IDM, however, requires the statements on every command in which a variable is used.

The host system participates in the assignment of range variables. Actually, the IDM never sees the relation variable itself, but a range variable number assigned by the host.

The optional "with minlocks" specifies that the IDM should use the minimum amount of locking on this variable. This provides degree 2 consistency rather than the default degree 3 consistency. With degree 2 consistency, page locks are released as the process finishes with the page rather than at the end of the transaction. Degree 2 locking will only be used for read locking and only if the relation has not had degree 3 locking in an earlier step in the transaction. There is a substantial processing expense in using degree 2 locking.

The RANGE token is followed by the length of the relation name plus 1 as a single byte, then the range number and the name of the relation. If "minlocks" is specified, then octal 100 is or'ed with the range number.

EXAMPLES:

```
range of products is products
```

Now the name "products" can be used in retrieve statements; this is sometimes useful for beginning users.

```
range of p is products:bill
```

Now the variable p is associated with the relation "products" and specifies that bill is the owner of this relation, since a database object is specified by the name of the object and by owner's name. Several users may own completely differently relations with the same name in the same database. If the owner's name is not ————————————————————

given then the object is pressumed to be owned by a current user or by the DBA. If current user does not own the object and does not specify the owner's name of the object to which he refers then the IDM will send an error message: "The <object name> not found" after an attempt to execute a query.

```
range of t is temp
range of t is parts
```

The variable "t" is associated with the last relation in a range statement involving it, so it is bound to the relation "parts" at the end of this command sequence. Note that this is a property of a possible IDL implementation. If the range declarations above are actually sent to the IDM, the "duplicate range number" error (see below) is returned by the IDM to the originating host.

ERROR MESSAGES:

There are no error messages from the IDM for the range statements, because the IDM never sees the range statement independent of an IDM command. The IDM will send error messages when the range declarations are actually sent if they are in error.

<u>Range number is too big</u>
Range variable numbers must be 0 - 15

<u>Duplicate range number</u>
A number was used twice in the range declarations for a single command.

7.5.34. READ

SYNTAX:

read(filenum, count, offset, addr)

OCTAL COMMAND CODE: 361

DESCRIPTION:

The *read* command is used by the IDM random access file system, and is not part of IDL. Relations may not be read with the *read* command; only files created with a *create file* can be the objects of the *read* and *write* commands. To read a file, it must first be opened with the *open file* command. This command returns a file number, which must be furnished in any *read* commands for the file. The *read* command takes three parameters, all mandatory: the file number, the byte-count offset within the file, and the number of bytes to read. The first byte of the file is addressed with an offset of 0. The "addr" is only used by the host interface routine to put the returned data in the right place.

The read command returns the number of bytes actually read. If this number is less than the number specified in "count", the end of the file has been reached. The actual interface to the IDM file system is determined by the host. The communication between the host program and the IDM is shown in the example below.

EXAMPLE:

Assuming a file has been previously opened for reading using the FILEOPEN command, the following sequence will read 300 bytes starting at byte offset 1200.

```
FILEREAD 4
INT2 1200
INT2 300
ENDOFCOMMAND
```

The IDM will respond with a DONE token followed by the actual data.

```
DONE status, unused, count
DATA
```

Note that the offset and count for the FILEREAD command can be expressed as INT1, INT2 or INT4. The value "4" from FILEREAD was the "filenumber" returned from a previous FILEOPEN command. After the sending the FILEREAD command to the IDM, the host program reads the DONE token which will have the actual number of bytes being returned. Any errors (bad file number, etc.) would have been returned before the DONE token. In such a case, the count in the DONE token would be zero and no data would follow the DONE token. Note that the IDM does not send anything after the last byte of DATA. The DONE token status word will have the DONE_CONTINUE and DONE_COUNT bits set if the FILEREAD is correct, otherwise, the DONE_ERROR will be set. In either case, the count in the DONE token will contain the actual number of DATA bytes being returned.

ERROR MESSAGES:

file not opened with proper mode
The file must have been opened with "read" or "read/write" mode.

7.5.35. RECONFIGURE

IDL SYNTAX:

```
     reconfigure
```

OCTAL COMMAND CODE: 334

DESCRIPTION:

The reconfigure command will cause the configuration of the IDM to be updated according to the information in the "configure" relation.

EXAMPLES:

```
reconfigure

CONFIGURE
ENDOFCOMMAND
```

7.5.36. REOPEN

IDL SYNTAX
not applicable

OCTAL TOKEN NUMBER: 330

DESCRIPTION:

The reopen command is used to allow several processes to share the same transaction id in the IDM. It is not part of IDL. It should be used whenever an update needs to be nested inside one or more retrieves. Reopen returns a database instantiation number (DBIN) of the child process in the done block. The process issueing the reopens is called the parent process. The child DBIN is used to identify the nested command between the USER and the IDM, similar to the DBIN returned by OPEN. The DBIN is transparent to the IDL user (i.e., a parser may be written which hides dbin from the user). A reopen should be issued for each level of nesting required by the application.

There are some restrictions which apply to its usage.

(1) The command can only be issued inside of a transaction and no updates must have occurred for the dbin issuing the command (called the parent) between the BEGIN TRANSACTION and the first DBREOPEN.

(2) Only 7 reopens can be issued inside one transaction.

(3) All the children dbins must be released (via EXITIDM) before the end of the transaction.

(4) In an abort or deadlock situation, the output for each outstanding command for each dbin in the "family" must be CANCELed and all the children dbins released. In this situation the children dbins are placed in a special state which only allows CANCEL and EXITIDM commands to be processed. All others will be returned with an ILLEGAL COMMAND error. If the children dbins are not removed with a EXITIDM they will remain waiting for input until the IDM is reset or a cancel process or cancel host command is received.

(5) Only the following commands will be accepted by a child dbin: RETRIEVE, REPLACE, APPEND, DELETE, SYNC, and DBCLOSE. BEGIN TRANSACTION and END TRANSACTION are accepted but they are noops. Stored commands can be executed using the child dbin also.

(6) None of the updates will be seen until the end of the transaction. The END TRANSACTION triggers the parent to process all the updates written in the log for itself and its children dbins. There is a possibility that some updates will not happen. If two dbins modify

the same tuple, only 1 will happen because the updates are done at the end and the first replace can move the tuple so that the second won't find it.

EXAMPLES:

After opening the desired database and starting a transaction the user should issue as many reopens as are needed to nest the updates. The host system would write to the IDM using the DBIN from the open database:

```
OPEN database          (returns parent dbin)
BEGIN TRANSACTION      (parent)
        DBREOPEN     (returns child1 dbin)
        DBREOPEN     (returns child2 dbin)
           .
           .
           .

        RETRIEVE     (using parent)
           RETRIEVE   (using child1)
             REPLACE    (using child2)
             if deadlock (or some other error)
                CANCEL (using parent)
                CANCEL (using child1)
                CANCEL (using child2)
                EXITIDM (using child1)
                EXITIDM (using child2)
                other exception processing
           .
           .
           .

        EXITIDM      (using child1)
        EXITIDM      (using child2)
END TRANSACTION
```

The IDM would return a DONE block with a child DBIN for future commands:

```
DONE statword dbin unused
```

That DBIN is then sent with all commands at the desired level of nesting in the transaction.

ERROR MESSAGES:

<u>too many reopens in this transaction</u>

The limit is 7.

<u>children still active at end of transaction</u>

The END TRANSACTION is not processed. The transaction will remain active until the children are released and an ABORT or END TRANSACTION is issued. The program can also be canceled which will abort the transaction.

parent already has updates - reopen denied.
The dbin issuing the DBREOPEN has already issued update commands in this transaction.

7.5.37. REPLACE

IDL SYNTAX:
```
    replace <variable> ( <target list> )
         [ where <qualification> ]
```

OCTAL COMMAND CODE: 305

DESCRIPTION:

The replace command replaces one or more attributes in zero or more tuples of a relation. The variable is outside the target list since only one relation may be affected by a single replace command. The replace may access more than one relation in calculating what is to be updated and how it is to change.

EXAMPLES:
```
        range of p is parts
        range of pr is products
        replace p (cost = p.cost + p.cost / 10)
                where p.name = pr.part and pr.name = "TV"
```

The cost of each part that is included in making a "TV" is raised by 10%.

```
        replace p (name = "electronic") where p.name = "t*"
```

The "name" attributes for all tuples in the relation "parts" for which the "name" fields begin with a "t" are changed to the value "electronic".

```
          REPLACE
          RANGE 6 10 parts
                              ROOT
                             (10 0)
                              /   \
                             /     \
                      RESATTR 4     \
                       (name)        \
                       /    \         EQ
                      /   CHAR 10    /  \
                 TLEND  (electronic) /    \
                                 VAR 6    PCHAR 2
                              (10 parts)  (t\200)

          ENDOFCOMMAND
```

ERROR MESSAGES:

permission denied:
    You do not have write permission for this relation

wrong type attribute

Character and numeric attribute types must be explicitly converted. The command "append" has examples of type conversion.

WARNING:
Care should be taken when replacing values in relation a with values from relation b that only one value in relation b qualify for each replacement in a. A syntactically correct <u>replace</u> may be ambiguous. In that case, the IDM will choose one among the qualifying values.

7.5.38. RETRIEVE, RETRIEVE INTO

IDL SYNTAX:
```
    retrieve [unique] [[ into ] <object name>  ]
    ( <target list> )
         [ order [ by ] <order_spec> [: a | d ]
         { , <order_spec> [: a | d ] } ]
         [ where <qualification> ]
```

OCTAL COMMAND CODE: 301, 302

DESCRIPTION:

The retrieve command (301) causes data to be sent to the host. The retrieve into (302) sets up a new relation, and puts the result of the retrieve into the new relation. The "retrieve" command can reference up to 15 relations; all of them must be in the same database. The "order by" clause specifies the sort order of the returned data. The <order_spec> may either be the name of an object in the target list or a new expression.

Rootnode bytes for retrieve/retrieve into

retrieve	ROOT 0 0
retrieve unique	ROOT 0 1
retrieve into	ROOT rno 0
retrieve unique into	ROOT rno 1

where rno is the range number of the result variable.

Retrieve command-options

send format before data	1
send domain names before format	2
ignore over/under flow	3
ignore divide by zero	4

normally arithmetic exceptions cause an abort.

EXAMPLES:

```
range of p is parts
retrieve (p.name, p.cost)
        order by cost:descending
        where p.cost > avg (p.cost)

  RETRIEVE
  RANGE 6 0 parts
                       ROOT
                       (0 0)
                      /    \
                     /      \
                    /        \
               RESDOM        GT
               /   \        /  \
              /     \    VAR 5  \
         RESDOM  VAR 5  (0 cost) \
         /   \  (0 name)       AGHEAD
        /     \                /    \
       TLEND VAR 5         AOPAVG    \
           (0 cost)            |   QLEND
                           VAR 5
                         (0 cost)
  ORDERD 2
  OPTIONS 1 1
  ENDOFCOMMAND
```

The above command causes the IDM to first calculate the average of the "cost" field of each tuple in the relation "parts". Then the IDM accumulates the "name" and "cost" attributes of the tuples which contained a "cost" greater than the average. These are sorted by the value in the "cost" attribute, largest value first, and sent to the host, preceded by the format of "name" and "cost" as requested by command-option 1.

Sometimes, it may be nesessary to create a new relation and to put the results of the retrieve into the new relation.

```
retrieve into exp_parts(p.name, p.cost)
        order by cost:descending
        where p.cost > avg (p.cost)
```

```
RETRIEVE
RANGE 6 0 parts
RANGE 10 1 exp_parts
                     ROOT
                     (1 0)
                    /    \
                   /      \
                  /        \
           RESDOM           GT
           /   \           /  \
          /     \     VAR 5    \
     RESDOM  VAR 5   (0 cost)   \
     /   \  (0 name)           AGHEAD
    /     \                    /    \
  TLEND VAR 5            AOPAVG      \
       (0 cost)             |       QLEND
                          VAR 5
                        (0 cost)
ORDERD 2
OPTIONS 1 1
ENDOFCOMMAND
```

```
retrieve(o.customer, o.part)
        order by o.date_ordered
```

Here we have an example of an "order by" clause which references a new expression.

```
RETRIEVE
RANGE 9 0 orderlog
                   ROOT
                   (0 0)
                  /    \
                 /      \
            RESDOM     QLEND
            /    \
           /      \
       RESDOM     VAR 9
       /   \     (0 customer)
      /     \
  ORDERDOM VAR 5
   /     \  (0 part)
  /       \
TLEND   VAR 13
     (0 date_ordered)
ORDERA 3
ENDOFCOMMAND
```

ERROR MESSAGES:

permission denied
: You must have read permission on all domains in the query.

not found
: The named attribute or relation was not found.

7.5.39. ROLLFORWARD

IDL SYNTAX:

```
    rollforward <database> from  <object name>  [ to <date>
         [ <time> ] ]
```

OCTAL COMMAND CODE: 352

DESCRIPTION:

The rollforward command is used to update a database after a load database command. The relation specified must have been created with a load transaction command or be the "transact" catalogue. The <date> and <time> specify that only transactions committed by that time will be redone. If the date and time are not specified then the whole log is used. Only the DBA may run this command.

A rollforward cannot be restarted or interrupted. If, for example, power is lost during a rollforward, the database must be reloaded from the last dump and the previous rollforward commands reissued.

EXAMPLES:

```
        open backup
        rollforward inventory from transl to May 30, 8:00


              DBOPEN 6 backup
              ENDOFCOMMAND
              ROLLFORWARD 9 inventory
              RANGE 7 15 transl
                           ROOT
                          (15 0)
                         /     \
                        /       \
                  TLEND         QUALDOM
                               /      \
                              /        \
                        INT4        QUALDOM
                        2344        /     \
                                   /       \
                             INT4        QLEND
                            1728000
              ENDOFCOMMAND
```

Note that the host system must convert the date to internal date count and time to the number of 60´ths of a second since midnight.

ERROR MESSAGES:

Wrong format
    The named relation or disk was not in transaction log format.

log out of sequence
    The log does not match the system's recorded time of the last dump or rollforward.

7.5.41. SETDATE/SETTIME

IDL SYNTAX:

```
     setdate <4-byte integer> | settime <4-byte integer>
```

OCTAL COMMAND CODE: 371, 372

DESCRIPTION:

The SETDATE and SETTIME commands are not part of IDL. System time and date are set to the time and date of the last checkpoint in the system database when the IDM is brought up. SETDATE and SETTIME allow the dba in the system database to specify the time and date. A checkpoint in the system database automatically happens any time the SETDATE or SETTIME command is issued. The IDM treats "time" as the number of 60th of a second since midnight. When "time" reaches 5,184,000 (60 * 60 * 60 * 24), it wraps around to zero and "date" is incremented. The time and date are automatically stored with each ENDTRANSACTION record in the transaction and batch logs. They can be accessed using AUDIT. They can also be retrieved or tested as an INT4 using the GETTIME and GETDATE functions. The time and date are sent to the IDM as 4-byte integers.

EXAMPLE:

The following sets the time of the IDM to 432000.

```
    SETTIME
    432000
    ENDOFCOMMAND
```

The IDM will respond with a normal DONE token.

7.5.41. SYNC

IDL SYNTAX:

```
sync
```

OCTAL COMMAND CODE: 33

DESCRIPTION:

The sync command causes a checkpoint on the currently open database. If no databases are open before the sync command is sent (that is, the sync command is sent to dbin 0), sync will cause a checkpoint on all active databases.

EXAMPLE:

```
open inventory
sync
```

The above command will checkpoint the database inventory.

```
SYNC
ENDOFCOMMAND
```

This page has been intentionally left blank

7.5.41A TRUNCATE

IDL SYNTAX:

truncate <object name> {, <object name>}

OCTAL COMMAND CODE: 312

DESCRIPTION:

The *truncate* command is used to quickly delete all tuples from a relation. The individual tuples are not recorded in the transaction log, so it is not possible to see which tuples were deleted with the *audit* command. A truncate command may be executed within a stored command or program, but since a truncate command cannot be backed out, it is not possible to truncate a relation inside a transaction.

EXAMPLE:

truncate monthtotal

The above command will delete all tuples from the "monthtotal" relation. All pages but one per index allocated to the relation will be made free.

```
TRUNCATE
                          ROOT
                         ( 0 0 )
                        /       \
                  RESDOM         QLEND
                 /      \
            TLEND        CHAR 10
                        (monthtotal)
ENDOFCOMMAND
```

IDM ERROR MESSAGES:

not owner
Only the owner of the relation, or the dba, can truncate the relation.

not relation
Only relations can be truncated. Other objects are not allowed.

illegal command
The *truncate* command cannot be executed inside a transaction.

relation is open
The user must be the only one accessing the relation.

system relation
System relations cannot be truncated.

7.5.42. WRITE

SYNTAX:

write (fnum, count, offset, addr)

OCTAL COMMAND CODE: 362

DESCRIPTION:

The *write* command is used by the IDM random access file system and is not part of IDL. A relation may not be the object of the *write* command; only files created with a *create file* can be the objects of the *read* and *write* commands. To write a file, it must first be opened with the *open file* command. This command returns a file number which must be furnished in any *write* commands for the file.

The *write* command takes three parameters, none of them optional: the file number, byte–count offset within the file and the number of bytes to write.

The *write* command returns the number of bytes actually written. If this number is less than the number specified in "count", there was an error in processing the *write*. If an error occurs in the FILEWRITE command itself, the IDM channel will disallow writing on the given dbin until a read is performed.

Writes to a file do not have to be contiguous. The length of a file is determined by the maximum offset ever written. "Holes" in a file read as zeros and occupy no space if they are at least 2K in length and fall on a 2K boundary. For example, creating a file and then writing one byte at offset 100000 will create a file of length 100001. Bytes 0–99999 will read as zero.

EXAMPLE:

The following writes 2000 bytes of data to a previously opened file starting at offset 100000.

```
FILEWRITE 2
INT4 100000
INT2 2000
ENDOFCOMMAND
DATA
```

The IDM will respond with a normal DONE token. The "count" value in the DONE token will contain the actual number of bytes written. Note that the offset and count values can be supplied as INT1, INT2 or INT4 tokens.

ERROR MESSAGES:

file opened with wrong mode
A file must have been opened for write or read/write.

bad file number
The file number does not correspond to an open file for this DBIN

7.5.43. WRITE EOF

SYNTAX

```
     writeeof(file, count, offset, addr)
```

OCTAL COMMAND CODE: 363

DESCRIPTION

Writeeof is like the *write* command except that the file *is truncated* to the *length* "offset + count".

EXAMPLES:

To truncate a file:

```
          writeeof(file, 0, 0, 0)

          FILEWRITE 2
          INT1 0
          INT1 0
          ENDOFCOMMAND
```

Britton Lee, Inc.

14600 WINCHESTER BLVD.
LOS GATOS, CALIFORNIA 95030
(408) 378-7000