ADABAS C for Natural Developers

by Jeffrey C. Gruber

This discussion of ADABAS C is intended for developers, not theorists, students or database administrators. To use Natural successfully, programmers need an understanding of how to access ADABAS successfully from the Natural programming environment.

If you want to know more about how ADABAS and Natural I/O statements work and how to retrieve ADABAS data more efficiently, you're in the right place. But this discussion provides guidelines, not absolutes, except

1) KNOW YOUR DATA

2) Understand what ADABAS does with your I/O requests

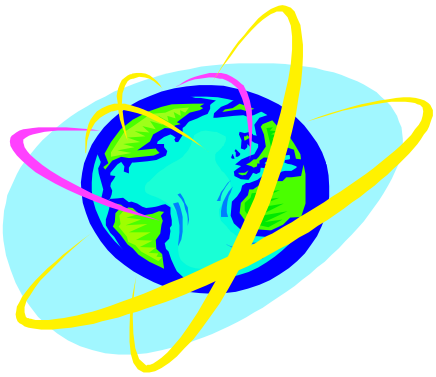
so that you can determine the best way to use your data.

Examples here are Natural 3.1.4 for OS/390 and ADABAS Version 7. When I refer to ADABAS in this discussion, I am referring to ADABAS C, the original database management system from Software AG, not ADABAS D, their SQL compliant database management system

Originally published 1998. Revised and updated 2001.

Introduction to ADABAS

What is ADABAS?



ADABAS is a database management system (DBMS) developed by Software AG of Germany in 1969. Today, corporations, schools and government agencies on six continents use ADABAS and have found it to be durable, robust, flexible and fast. The name ADABAS stands for Adaptable DAtaBASe. It is also very efficient in managing large databases and files.

ADABAS is a pseudo-relational DBMS. A *fully* relational DBMS creates access paths to the database each time application programs request data. This gives applications greater flexibility, but can be extremely costly in machine resources. A pseudo-relational DBMS stores these access paths to the database rather than creating them for each request. By defining selection criteria or indexes in advance, ADABAS can quickly locate and return the requested data. In ADABAS jargon, we call these stored indexes descriptors.

ADABAS is like any truly powerful tool; it must be used well to be useful. Use it poorly and it will only cause you trouble. Learn what ADABAS does, how it does it and how to use it and your Natural systems will be powerful, flexible and durable.

ADABAS Components

Each ADABAS database has four components.

– The ADABAS **Nucleus** is the engine that powers the DBMS and the brain that controls it.

– **Data Storage** is where ADABAS stores the raw data of the database. ADABAS divides Data Storage into data storage blocks.

- The **Working Storage** component is the scratch pad where ADABAS keeps control information.

– The **Associator** is our link to the raw data in Data Storage. ADABAS keeps descriptor information here. The Associator has two important components.

– The **Inverted List** is where ADABAS stores descriptor information.

– The **Address Converter**, which ADABAS uses to find data storage, blocks containing individual records.

Developers usually don't need to discuss the Nucleus, Work or Data Storage components of ADABAS, but now you can nod knowingly in design meetings when data analysts and DBAs mention them and things like RABNs.[1]

How ADABAS Stores Data

ADABAS stores the raw data as records in the Data Storage component. Each file in the database has a number of data storage blocks to hold it records. Within each data block, ADABAS allocates **free space** and **padding**. ADABAS uses these to provide space within the block for new records and records that enlarge after updating. If an updated record becomes too large to fit in a data block, ADABAS will move a record to another data block. This is a minor point from the developers' perspective, but can be important at times.

When we store a record, ADABAS assigns the record an Internal Sequence Number (**ISN**) that uniquely identifies a record within a file. Records have one or more fields.

When a field contains no data or is null, ADABAS will not store the null value. For example, we have defined a field in our employees file[2], called NOTES, as alphanumeric for a length of 100. If we store a record with spaces in NOTES, ADABAS will store only two bytes for this field. One byte to identify NOTES and the other to indicate that it is empty. Here's a simplified representation of some employee records as stored in the database.[3]

```
&750429&Rumplestiltskin&Rhonda&Typing,German&Payroll&&
&790152&Branson&Billy&Gift of Gab&Sales&&
&811249&Smithe&Wilma&Tech. Writing&Shipping&&
```

This compression technique is a major strength of ADABAS, because it allows us to save a lot of storage space. We do not need to take up space for fields that are rarely or sparsely populated. We only need the space to store the fields that actually contain data.

Since ADABAS compresses data for storage, it has to decompress the data when it retrieves records. To get the department value (Payroll) on the first record, ADABAS must decompress all the fields in front of DEPARTMENT. We need to keep ADABAS compression techniques in mind when creating record layouts and defining views to ADABAS files for Natural applications.

We can override the normal compression for a field by setting the compression option in the file definition. The compression options are normal, fixed and null suppression.

---

[1] RABN (usually pronounced rob-bin or rabe-bin) is short for Relative ADABAS Block Number and identifies a data storage block. Normally, developers don't care about RABNs.

[2] The DDM is on page 8.

[3] The ampersands (&) represent bytes of information ADABAS uses to identify each field and to indicate null fields.

With **normal suppression**, ADABAS will drop trailing spaces and leading zeros before storing the record.

With **the fixed suppression option**, ADABAS will store the value in the field without taking out trailing spaces or leading zeros. Use the fixed option for fields that will always be fully populated. Normally, we also make one and two byte fields fixed, except when they will most often be empty and can be grouped with other fields that will most often be empty. This allows us to take advantage of the null suppression option.

Null suppression causes ADABAS to drop trailing spaces and leading zeros, like normal suppression. Unlike normal suppression, null suppression allows ADABAS to string together null fields. Let's say we define a number of contiguous fields on the record with the same format and with null suppression. If those fields are all null, ADABAS will store two bytes for all those fields. One byte to indicate that the following fields are null and the other to indicate the number of null fields. This can bring greater space savings.

Null suppression affects the behavior of descriptors. For details, see the discussion on descriptors below.

Field Formats and Types

Other attributes of ADABAS fields, besides compression option, are name, format and length. ADABAS supports many standard field formats and each has a corresponding Natural field format. Most often, we will refer to the Natural field formats, even in Predict, Software AG's data dictionary. Natural field formats are much like those in other languages. Here are a few points consider:

- Alphanumeric — These are often called alpha fields, but they can hold numbers, too.
- Numeric — A.K.A. unpacked numeric. ADABAS stores them as packed fields to save space, but unpacks before returning the data.
- Logical [4] — ADABAS stores these as integers.
- Timestamp [5] — ADABAS stores these as P12.
- Date [6] — ADABAS stores these as P7.

---

[4] A.K.A. switches. These fields have two possible values, TRUE and FALSE, which ADABAS stores as 1 or 0.

[5] Contains the date and time to the tenth of a second.

[6] Stored in a proprietary format, which allows us to perform date arithmetic directly on the date field.

Of more interest to us in this discussion are the field types.

- Elementary Field — The mere mortal of ADABAS fields. Holds one value and has no super powers.

- Group Field — This is a field under which we group other related fields. For example, we could have a group field called ADDRESS and place STREET-ADDRESS, CITY and ZIP-CODE under it. The group field is only a logical construct and is not a real field on the database.

- Multiple Value Field (MU) — We can store more than one value in this field on a single record. Think of it as an array.

- Periodic Group (PE) — This is a repeating group of fields. Like the group field, it allows us to define a group of associated fields. Like the MU field, it allows us to store more than one value. Think of it as a group array. We might have a PE that consists of a status code and the date on which it became effective. In this way, we can store, on a single record, the history of the status and effective dates.

  You will sometimes see, in Predict, MC for MU and PC for PE. Construct, Software AG's code generator, uses this when generating code in some modules. The MC and PC cause Construct to create structures with smaller format buffers and thus reduce program overhead.

- Redefine (RE) — This is not really a field type, but I include it here because of its value in standardizing field sub-structures using Predict. ADABAS does not include redefined fields in physical file definitions. Thus they cost nothing in I/O operations. Construct makes good use of them in generating data definitions. If you use Construct, you should use redefinitions for elementary fields that have logical sub-components. For example, we might redefine a telephone number field into area code, prefix and suffix.

- C* variables — C* variables are cool. C* variables tell us the number of occurrences for repeating fields, PE's and MU's. The C* variable is not defined explicitly on the ADABAS record, but we can add them to a programmatic user view. See Appendix B for more on C* variables.

How the Program Sees the Data

For a Natural application program to receive data from ADABAS, a number of file definitions must be in place.

– The File Definition Table (FDT) describes the physical file structure to ADABAS.

```
DEFFDT   FILE=244
         FNDEF='01,AA,6,A,DE,UQ,NU'
         Field: EMPLOYEE-ID
         FNDEF='01,AB,40,A,NU'
         Field: LAST-NAME
         FNDEF='01,AC,30,A,NU'
         Field: FIRST-NAME
         FNDEF='01,AD,10,A,NU,MU'
         Field: SKILL-CODE
         FNDEF='01,AE,8,A,NU'
         Field: NOTES
         FNDEF='01,AF,100,A,NU'
         Field: DEPARTMENT
         SUBDE='AG=AA(3,4)'
         Field: HIRE-MONTH
         SUPDE='AH=AB(1,8),AC(1,6)'
         Field: NAME-SEARCH-KEY
         SUPDE='AI=AE(1,8),AB(1,40)'
         Field: DEPARTMENT-NAME-KEY
```

– The Data Definition Module (DDM) is a link between an ADABAS FDT and the Natural environment. We sometimes call this a view. It is what we see when we use the LIST FILE or LIST VIEW command in Natural.

```
DB 1    File 244  - EMPLOYEES-FILE

T L  DB  Name                            F Leng  S D
- -  --  --------------------------------- - ----  - -
  1  AE  DEPARTMENT                      A    8  N
  1  AI  DEPARTMENT-NAME-KEY             A   48  N S
        -------- SOURCE FIELD(S) -------
*       DEPARTMENT(1-8)
*       LAST-NAME(1-40)
  1  AA  EMPLOYEE-ID                     A    6  N D
  1  AC  FIRST-NAME                      A   30  N
  1  AG  HIRE-MONTH                      A    2  N S
*       -------- SOURCE FIELD(S) -------
*       EMPLOYEE-ID(3-4)
  1  AB  LAST-NAME                       A   40  N
  1  AH  NAME-SEARCH-KEY                 A   14  N S
*       -------- SOURCE FIELD(S) -------
*       LAST-NAME(1-8)
*       FIRST-NAME(1-6)
  1  AF  NOTES                           A  100  N
M 1  AD  SKILL-CODE                      A   10  N
```

– The programmatic user view is the link between the Natural program and the DDM. We also call this a view. Do not include unneeded or unused fields in views, because ADABAS must decompress all fields in the view. It also causes an increase in program overhead.

```
0010 DEFINE DATA LOCAL
0030 01 EMPLOYEES-FILE VIEW OF EMPLOYEES-FILE
0040   02 EMPLOYEE-ID
0050   02 REDEFINE EMPLOYEE-ID
0060     03 FILLER 2X
0070     03 EMP-HIRE-MONTH (A2)
0075     03 EMP-HIRE-DAY   (A2)
0080   02 LAST-NAME
0090   02 FIRST-NAME
0100   02 SKILL-CODE
0110   02 DEPARTMENT
0120 END-DEFINE
```

How ADABAS Finds the Data -- the Associator

We said at the beginning that ADABAS stores the access paths to the data instead of creating them when an application requests data. ADABAS stores these paths as descriptors in the Associator.

The most important component of the Associator, from the developers' viewpoint, is the **Inverted List**. For each descriptor field on a file, ADABAS creates and maintains an inverted list. This contains a unique list of the values contained in that descriptor field. With each value is a count of records that have that value and a list of ISNs for those records. The inverted list is the heart of selective data retrieval in ADABAS.

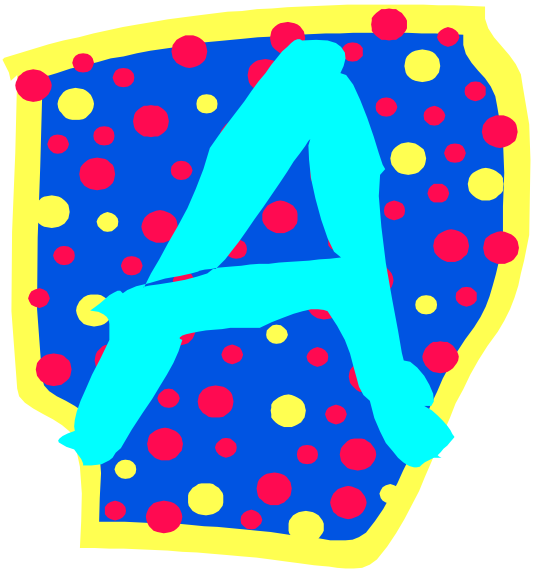
When ADABAS receives a request for data through a descriptor, it takes these steps:

1. Finds entry in inverted list whose value matches the requested value.

2. Processes ISN(s) from the entry through the Address converter to find and retrieve the data block containing the records with the ISN(s).

3. Reads record from the data block and decompresses fields defined in the user view

4. Returns record to the requesting program.



SSOCATIOR

Here's an example of an employee file, with employee ID and department as descriptors.

EMPLOYEES-FILE

Employee ID	Last Name	First Name	Skill Code	Dept.	Notes	ISN
750429	Rumplestiltskin	Rhonda	Typing, German	Payroll		1
811249	Branson	Billy	Gift of Gab	Sales		7
910462	Smithe	Wilma	Tech. Writing	Shipping		3
841107	Andersen	Andrew	Omelets	Shipping		6
840478	Kirkland	Patty	French, Russian	HR		2
951001	Abdul	Jerry	Gift of Gab	Sales		4
790152	Fine	Lawrence	Can hit a curve ball	HR		5

Employee ID Inverted List

Value	Count	ISNs
750429	1	1
790152	1	5
811249	1	7
840478	1	2
841107	1	6
910462	1	3
951001	1	4

Department Inverted List

Value	Count	ISNs
HR	2	2,5
Payroll	1	1
Sales	2	4,7
Shipping	2	3,6

If a program asks ADABAS for the record with employee ID 750429, ADABAS reads the inverted list for employee ID and finds that ISN 1 goes with the value 750429. Going through the Address Converter to find the physical location of record 1 of the employee file, ADABAS returns Rhonda Rumplestiltskin's record to the application program. That is the basic process or should I say the ADABASic process?

To make this work, ADABAS modifies the information in the inverted list each time we add, modify or delete record data that a descriptor uses. If we add a new record to the employee file, ADABAS will add entries to the employee ID and department inverted lists. If we deleted Lawrence Fine's record, ADABAS would delete the 790152 entry in the employee ID inverted list and change the HR entry in the department inverted list. This process keeps the inverted list in synchronization with the data and ensures that ADABAS will be able to quickly retrieve the records requested and allows great flexibility.

Descriptor Types

The descriptors and inverted list examples so far are for simple descriptors**. Simple descriptors** are elementary fields that are **inverted** or made into descriptors. ADABAS uses the

values recorded in that field to populate the inverted list. Let me tell you about other descriptor types.

Super descriptors wear red and blue leotards with a big yellow 'S' on the front. They also combine parts of or all of one or more fields to create an index. For example, we could create a super descriptor on the employee file that includes the first eight bytes of the last name with the first six bytes of the first name to create a name search key. Or we could create a super that combined department and last name. This department-last-name descriptor would allow us to read the employees file by name within department.

Name Search Inverted List

Value	Count	ISNs
Abdul Jerry	1	4
AndersonAndrew	1	6
Branson Billy	1	7
Smithe Wilma	1	3
Fine Lawren	1	5
KirklandPatty	1	2
RumplestRhonda	1	1

Department Name Inverted List

Value	Count	ISNs
Hr Fine	1	5
Hr Kirkland	1	2
Payroll Rumplest	1	1
Sales Abdul	1	4
Sales Branson	1	7
Shippingandersen	1	6
Shippingsmithe	1	3

A **sub-descriptor** uses a portion of a field. Let's say that the third and fourth digits of the employee ID represent the month that the company hired an employee. We could create a sub-descriptor using these two bytes. We could use that sub-descriptor to retrieve all the employees hired in a given month.

Hire Month Inverted List

Value	Count	ISNs
01	1	5
04	3	1,2,3
10	1	4
11	1	6
12	1	7

Hyper descriptors exist mostly in the nightmares of DBAs and offer the greatest flexibility in populating an inverted list. They allow us to combine fields and derived or calculated values. DBAs dislike them, because someone has to write and maintain a program module that creates the desired value and can make database transactions less efficient. The database team must insert that program into the ADABAS nucleus through a user exit. We rarely use hyper descriptors, because we can accomplish the same thing programmatically with some additional database fields. So give your DBA a break and forget I mentioned hyper descriptors.

Phonetic descriptors are also rarely used. These descriptors allow us to match up to values in the inverted list that sound like the search value. If the last name were a phonetic descriptor, we could find Wilma Smithe by asking ADABAS for a match on 'SMITH'.

Software AG talks about another descriptor type, the **unique descriptor**. Actually, that uniqueness is only an attribute of a descriptor. We can make any descriptor a unique descriptor. When we have a unique descriptor, ADABAS ensures that only one record on the file has each value. If we try to add or update a record that would make a descriptor not unique, ADABAS issues an error message at execution time.

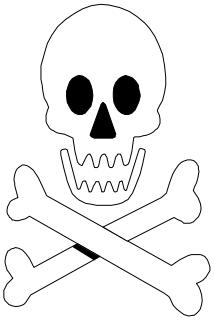
Should we use unique descriptors? Most often, we don't want an execution error when a user tries to add a duplicate value for a descriptor. That error would disrupt processing and present the user with an unfriendly message. So we add logic to programs to check the uniqueness of the descriptor value before trying to update or store a record. That means we have to duplicate the work that ADABAS is doing, but there is an advantage to using unique descriptors. Unique descriptors allow DBAs to enforce the uniqueness of a descriptor centrally, in the DBMS, instead of relying on application programs.

Null Suppressed Descriptor

ADABAS processes null suppressed descriptors differently when it maintains the inverted list for that descriptor. If the value for a null descriptor is null, ADABAS will not create an entry in the inverted list for that value. If any component value for a null super descriptor is empty, ADABAS will not create an inverted list entry.

This feature saves overhead in the associator and allows us to create more selective descriptors.

Redundant Descriptors



Avoid redundant descriptors; they waste resources. A redundant descriptor provides a redundant or unnecessary path to data on a file. If a super descriptor has for its first component a field that is a descriptor, the simple descriptor is redundant. We have a redundant descriptor on our employees file. DEPARTMENT-NAME-KEY has DEPARTMENT as the first component. DEPARTMENT is also a descriptor. DEPARTMENT is a redundant descriptor.

We can use DEPARTMENT-NAME-KEY to retrieve all records for a particular department. Let's say we wanted to retrieve all records for the human resources department. Instead of asking ADABAS to look for 'HR' in the department inverted list, we ask ADABAS for values in the DEPARTMENT-NAME-KEY inverted list starting from 'HR' through 'HR99999999999999999999999999999999999999999999'. Using this retrieval technique eliminates the need for DEPARTMENT as a descriptor and saves the associated overhead.

Getting Data FROM the Database

We retrieve data from ADABAS using Natural I/O statements. The best chance any Natural programmer has for wasting machine resources is using the wrong I/O statement or using one inappropriately. Mistakes with I/O statements are much more costly than mistakes using any other Natural statements. Here are the statements and how each works.

FIND

The FIND statement allows us to retrieve records by matching the value of a descriptor. This statement offers many options, but we will discuss the most basic ones. Here's an example of a FIND using our employees file:

```
  GET-EMPLOYEE-RECORD.
FIND EMPLOYEES-FILE WITH EMPLOYEE-ID = #EMPLOYEE-ID
  process employee
END-FIND /* GET-EMPLOYEE-RECORD.
```

When ADABAS receives the request represented by this FIND,

1) It searches the inverted list for the employee ID looking for an ID that is equal to the contents of #EMPLOYEE-ID

2) Builds a set or list of ISNs that match the value. This list is build in working storage

3) Processes the ISNs, one at a time, through the Address Converter to locate and retrieve the record.[7]

[7] To retrieve the record, ADABAS must first retrieve the data block in which the record resides.

Remember that ADABAS builds a list of ISNs before it returns any records. Once ADABAS creates the ISN list, changes to the database are not reflected in the list. In processing the GET-EMPLOYEE-RECORD FIND statement, ADABAS builds an ISN list of the records whose ID matches the value of #EMPLOYEE-ID. If someone adds another record with that value in the ID after the set is built, ADABAS will not return that new record to the calling program.

Keep in mind that ADABAS returns records in the order they appear in the ISN list. **So if the order, in which ADABAS returns the records, is important, don't use a FIND statement.**

In addition to the record, ADABAS returns this information in these system variables:

*NUMBER contains the number of ISNs in the set.

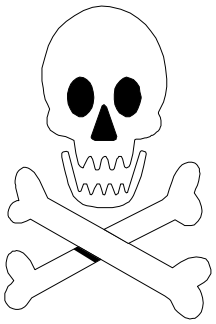
*COUNTER tells us how many times the program has gone through the FIND loop.

*ISN tells us the ISN of the current record.

There are several options for the FIND statement. We can limit the number of records by using a numeric literal or a variable like this:

```
  GET-EMPLOYEE-RECORD-WITH-LIMIT.
FIND #RECORD-LIMIT EMPLOYEES-FILE
  WITH EMPLOYEE-ID = #EMPLOYEE-ID
  process employee
END-FIND /* GET-EMPLOYEE-RECORD-WITH-LIMIT.
```

If we populate #RECORD-LIMIT with the value 1, Natural will terminate the FIND loop after the first record is processed. We can set loop limits like this on all I/O statements that initiate a processing loop.[8]



We can create a FIND that uses more than one descriptor. **Don't.** The compound or complex FIND can get programmers into more trouble than any other I/O statement. Let's look at an example:

```
  GET-APRIL-PAYROLL-EMPLOYEES.
FIND EMPLOYEES-FILE WITH HIRE-MONTH = '04'
  AND DEPARTMENT = 'PAYROLL'
  process employee
END-FIND /* GET-APRIL-PAYROLL-EMPLOYEES.
```

This FIND will give us all the employees who work in payroll and the company hired in April, but ADABAS will do too much work to deliver those records. First, ADABAS creates a set of employees hired in April. Second, it creates another set of employees who work in payroll. Third, it compares the two ISN lists to see which ISNs appear in both lists and put those ISNs

[8] READ, HISTOGRAM and FIND

into a third set. That is not a lot of records in our example database, but imagine that we have a large company that hired 3000 people in different April's and 100 people working in payroll. Then suppose that only three of the people in payroll were hired in April. ADABAS will have to sort through 3100 ISNs to deliver the three records we requested. It gets worse as we add more descriptors or use larger files.

For each descriptor we use in a FIND, ADABAS creates an ISN list and compares the lists to boil it down to one. The more descriptors, the more work ADABAS does**. Don't use multiple descriptors on FIND statements.** Use the one descriptor that you expect will return the fewest records and REJECT the ones you don't want. Here's our example re-done:

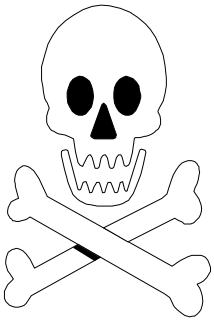
```
  GET-APRIL-PAYROLL-EMPLOYEES-BETTER.
FIND EMPLOYEES-FILE WITH DEPARTMENT = 'PAYROLL'
  REJECT IF EMPLOYEES-FILE.HIRE-MONTH NE '04'
  process employee
END-FIND /* GET-APRIL-PAYROLL-EMPLOYEES-BETTER.
```

Using this method, ADABAS will create only one ISN list that we filter programmatically. The program does a little more work, but ADABAS does less. The result is faster processing. This is why it is critical to ***know your data***.

We can also compare a descriptor to a range of values as in this example:

```
  GET-1STQTR-HIRE-EMPLOYEES.
FIND EMPLOYEES-FILE WITH HIRE-MONTH = '01' THRU '03'
  process employee
END-FIND /* GET-1STQTR-HIRE-EMPLOYEES.
```

Before you use the FIND with a range, be sure you ***know your data***. If you will be retrieving a large number of records, consider using a READ statement. If the order of the records is important, do not use the FIND.



There is something called **FIND SORTED BY. Forget you ever heard of it.** It can be a big resource hog. We don't need it. Don't use it.

We can use a WHERE clause with FIND statements. The WHERE clause works much like an ACCEPT statement. In some circumstances, the WHERE clause will do more than we expect. Use an ACCEPT or REJECT statement instead.

The IF NO RECORDS FOUND clause on a FIND will allow us to specify logic to be executed when ADABAS finds no records for the stated criteria. Like this:

```
  GET-1STQTR-HIRE-EMPS-NOREC.
FIND EMPLOYEES-FILE WITH HIRE-MONTH = '01' THRU '03'
  IF NO RECORDS FOUND
    WRITE 'No first qtr hires on file.'
    ESCAPE BOTTOM (GET-1STQTR-HIRE-EMPS-NOREC.)
  END-NOREC
  process employee
END-FIND /* GET-1STQTR-HIRE-EMPS-NOREC.
```

Notice the ESCAPE BOTTOM. Without some kind of branching instruction in the NO RECORDS FOUND clause, the program will execute the logic within the FIND loop.

Defining the User View for a FIND

When creating the programmatic user view for a FIND, we don't have to include the descriptor in the view if we only use it for search criteria. Include only the fields needed for other processing.

FIND NUMBER

What if we just want to know how many records match the value of a descriptor? We don't have read all the records; just use FIND NUMBER. Here's an example:

```
  COUNT-PAYROLL-EMPLOYEES.
FIND NUMBER EMPLOYEES-FILE
  WITH DEPARTMENT = 'PAYROLL'
DISPLAY *NUMBER (COUNT-PAYROLL-EMPLOYEES.)
```

Notice that there is no END-FIND here. FIND NUMBER, unlike FIND, does not initiate a processing loop. ADABAS returns *NUMBER representing the number of records that match the selection criteria. It returns no records.

FIND NUMBER is a handy and easy to use statement, particularly if we want to know if a particular unique value is on file. Be careful, though. Here are some uses of the FIND NUMBER that can get you in trouble.

FIND NUMBER where the expected number can exceed forty. ADABAS does much the same work for a FIND NUMBER that it does for a FIND. It builds the ISN list; it just doesn't return any records. When you expect the number to be less than forty, the overhead is less than the alternative method of the HISTOGRAM.[9] The less-than-forty rule is a rule of thumb given to me by Rick Gurney, ADABAS guru at Software, AG. In cases where the number might be higher, use HISTOGRAM to get the number of records.

FIND NUMBER with more than one descriptor. As with the FIND, ADABAS will have to create multiple ISN sets. If you need to count records based on the values in

[9] FIND NUMBER is more efficient than HISTOGRAM when the *NUMBER value is low, because FIND NUMBER does not issue a RC (ADABAS release command), while HISTOGRAM does. In later versions of ADABAS, FIND NUMBER was made more efficient, because it no longer writes the ISN list to working storage.

two fields, use a FIND on the descriptor expected to return the fewest records and use logic in the FIND loop to count the acceptable records.

FIND NUMBER with a WHERE clause. Don't be fooled by the WHERE clause. It may look like ADABAS is not reading any records in the COUNT-PAYROLL-EMPLOYEES-USING-WHERE statement below. It *will* read the records to check the value of HIRE-MONTH. Make the logic explicit by using a FIND and counting the records yourself.

```
  COUNT-PAYROLL-EMPLOYEES-USING-WHERE.
FIND NUMBER EMPLOYEES-FILE
  WITH EMPLOYEE-ID = #EMPLOYEE-ID
  WHERE HIRE-MONTH = '04'
```

FIND NUMBER to decide whether to FIND a record. **Don't do it.** Just FIND the record. If it is not there, ADABAS will do no extra work. Here's an example:

```
  IS-RECORD-ON-FILE.
FIND NUMBER EMPLOYEES-FILE
  WITH EMPLOYEE-ID = #EMPLOYEE-ID
IF *NUMBER (IS-RECORD-ON-FILE.) NE 0
    GET-RECORD-WHEN-EXISTS.
  FIND (1) EMPLOYEES-FILE
    WITH EMPLOYEE-ID = #EMPLOYEE-ID
    process employee
  END-FIND /* GET-RECORD-WHEN-EXISTS.
END-IF
```

In this example, ADABAS will

1. Create an ISN list where EMPLOYEE-ID matches #EMPLOYEE-ID for the FIND NUMBER statement.

2. Return the number of matching records in *NUMBER.

3. Throw away the ISN list.

4. Create the same ISN list where EMPLOYEE-ID matches #EMPLOYEE-ID, this time for the FIND statement.

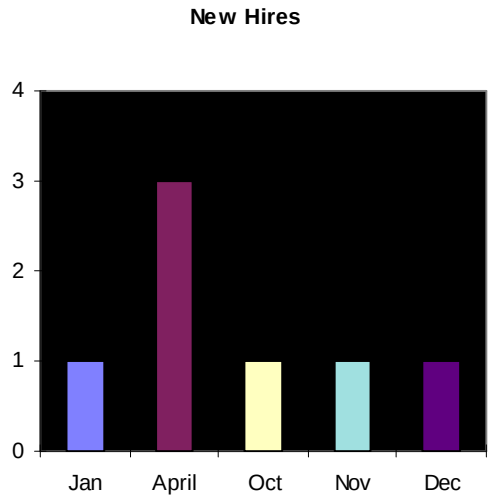
5. Return the record to the calling program.

Defining the User View for a FIND NUMBER

The programmatic user view for a FIND NUMBER does not need to have any fields, but it can have many fields. This is one of the reasons that the FIND NUMBER is so handy. If we already have a view of the file defined in the program for some other purpose, we can also use it for a FIND NUMBER.

HISTOGRAM

HISTOGRAM is one of the most efficient I/O statements in Natural. It also returns less information than others do.



To understand the information the HISTOGRAM statement returns, look at the New Hires histogram chart. Each column in the chart represents the number of new hires for a month. Each column also parallels an entry in the inverted list of HIRE-MONTH on page 11. The HISTOGRAM statement returns the value of a descriptor in the inverted list, the month in this case, and the number of records that have that value.

The Histogram is one of the most efficient statements, because it gets all its information from the inverted list. For a HISTOGRAM, ADABAS does not have to read records from data storage. Here's the HISTOGRAM statement that would return the information shown in the chart.

```
  COUNT-EMPLOYEES-BY-HIRE-MONTH.
HISTOGRAM EMPLOYEES-FILE HIRE-MONTH
  DISPLAY EMPLOYEES-FILE.HIRE-MONTH
    *NUMBER (COUNT-EMPLOYEES-BY-HIRE-MONTH.)
END-HISTOGRAM /* COUNT-EMPLOYEES-BY-HIRE-MONTH.
```

This is how ADABAS processes a HISTOGRAM:

1. Looks in the inverted list for the first value that is equal to or greater than the STARTING FROM value and less than or equal to the THRU value indicated on the HISTOGRAM statement. We did not use a STARTING FROM or THRU value in our example, because we wanted a report of all values in the inverted list. When we do not specify a starting value, ADABAS starts at the beginning of the inverted list.
2. Returns the following:
   a) Value of the descriptor
   b) *NUMBER, which contains the number of records that have this value
   c) *COUNTER, which is a count of the number of the times through the HISTOGRAM loop. This would also be the number of values in the inverted list so far.
   d) NOT *ISN. The manual indicates that the HISTOGRAM returns a value in *ISN. This is a trick. The value in *ISN is not a valid ISN.[10]

[10] *ISN will contain an occurrence value when the descriptor is in whole or part a repeating field.

3. Looks at the next entry in the inverted list. If the value is less than or equal to the THRU value[11], processes the entry as in step 2.
4. Continues processing inverted list entries until the value is greater than the THRU value or ADABAS reaches the end of the inverted list.

Here's an example of a HISTOGRAM with STARTING FROM and THRU values.

```
ASSIGN #HIRE-MONTH = '12'
  ANYONE-HIRED-IN-DECEMBER.
HISTOGRAM (1) EMPLOYEES-FILE HIRE-MONTH
  STARTING FROM #HIRE-MONTH THRU #HIRE-MONTH
  ASSIGN #DECEMBER-HIRE-ON-FILE = TRUE
END-HISTOGRAM /* ANYONE-HIRED-IN-DECEMBER.
```

We wanted to know if we had at least one employee who was hired in December. We set a starting and ending value of '12', so ADABAS would begin in the inverted list at the first value that was equal to and not greater than '12'. We set a loop limit of one, because we don't want ADABAS to go to the inverted list more than once. The following example is the wrong way to do this:

```
ASSIGN #HIRE-MONTH = '12'
  ANYONE-HIRED-IN-DECEMBER-WORSE.
HISTOGRAM EMPLOYEES-FILE HIRE-MONTH
  WHERE EMPLOYEES-FILE.HIRE-MONTH = #HIRE-MONTH
  ASSIGN #DECEMBER-HIRE-ON-FILE = TRUE
END-HISTOGRAM /* ANYONE-HIRED-IN-DECEMBER-WORSE.
```

Since we did not specify a STARTING FROM value, ADABAS will start at the beginning of the inverted list. It will go to the end of the inverted list, because we did not specify a THRU value. We will get the same result. ADABAS will eventually hit the '12' entry in the inverted list, but it goes through the entire inverted list, instead of going right to the '12' entry.

HISTOGRAM DESCENDING

After years in the wilderness thirsting for the ability to HISTOGRAM descending, Natural programmers have at last be delivered. Software AG came through with this in the latest versions of Natural.[12] Previously, all HISTOGRAMs went through the inverted list in ascending order only. By

[11] When a THRU or ENDING AT value is specified.

[12] The ability to HISTOGRAM and READ in descending order eliminates the need to use date nines compliment fields in order to get records in reverse date order. This technique has been used in many places and situations and required us to carry an extra field and descriptor. That additional overhead is no longer needed.

adding the key word DESCENDING, we can tell ADABAS to go through the inverted list in reverse order. Here's an example:

```
  HISTOGRAM-EMPLOYEES-IN-DESCENDING-ORDER.
HISTOGRAM EMPLOYEES IN DESCENDING SEQUENCE
  FOR EMPLOYEE-ID
    DISPLAY EMPLOYEES.EMPLOYEE-ID
END-HISTOGRAM /* HISTOGRAM-EMPLOYEES-IN-DESCENDING-ORDER.
```

This example will produce this report. Notice that the sequence of IDs is the reverse of the order in the inverted list:

```
EMPLOYEE-ID
------------
   951001
   910462
   841107
   840478
   811249
   790152
   750429
```

HISTOGRAM IN VARIABLE SEQUENCE

Oh, but, it gets even better. Not only can we HISTOGRAM descending, we can dynamically set the direction through the inverted list from descending to ascending and vice versa by using the key words IN VARIABLE SEQUENCE.

```
IF #SHOW-DATA-IN-DESCENDING-SEQUENCE
  ASSIGN #READ-DIRECTION = 'D'
ELSE
  ASSIGN #READ-DIRECTION = 'A'
END-IF
  HISTOGRAM-EMPLOYEES-IN-VARIABLE-ORDER.
HISTOGRAM EMPLOYEES IN VARIABLE #READ-DIRECTION SEQUENCE
  FOR EMPLOYEE-ID
  DISPLAY EMPLOYEES.EMPLOYEE-ID
END-HISTOGRAM /* HISTOGRAM-EMPLOYEES-IN-VARIABLE-ORDER.
```

Notice that we must define a variable (#READ-DIRECTION in this case) to hold the value A for ascending and D for descending. Any other value will result in an execution error.

Changing the value of #READ-DIRECTION from within the HISTOGRAM loop will not change the direction of the HISTOGRAM.

Defining the User View for a HISTOGRAM

Unlike views for other I/O statements, views for HISTOGRAMs can contain only the descriptor used in the HISTOGRAM. Natural will not allow any other fields in the view. We can redefine the descriptor in the view. This can be very useful when using a super descriptor, but the component fields for the redefinition cannot be the same as other fields in the DDM. Here's an example of a view used for a HISTOGRAM on the DEPARTMENT-NAME-KEY:

```
DEFINE DATA LOCAL
01 EMPLOYEES-FILE VIEW OF EMPLOYEES-FILE
  02 DEPARTMENT-NAME-KEY
  02 REDEFINE DEPARTMENT-NAME-KEY
    03 #DEPARTMENT (A8)  /* cannot call this DEPARTMENT
    03 #LAST-NAME  (A40) /* cannot call this LAST-NAME
END-DEFINE
```

READ LOGICAL (READ BY)

READ logical allows us to read records from a file 'sorted' by a descriptor. I put sorted in quotes, because we are not sorting the records when we read them. ADABAS has already sorted them using the inverted list. As ADABAS added each record, it put the value for the descriptor in the inverted list in sorted order. Here's an example:

```
  HR-EMPLOYEES.
READ EMPLOYEES-FILE BY DEPARTMENT-NAME-KEY
  STARTING FROM 'HR'
    IF EMPLOYEES-FILE.DEPARTMENT NE 'HR'
      ESCAPE BOTTOM (HR-EMPLOYEES.)
    END-IF
    DISPLAY EMPLOYEES-FILE.LAST-NAME
      EMPLOYEES-FILE.FIRST-NAME
END-READ /* HR-EMPLOYEES.
```

This example will produce this report:

```
     LAST-NAME            FIRST-NAME
     -------------------- ----------------
     Fine                 Lawrence
     Kirkland             Sally
```

The steps ADABAS takes to process a logical READ are like a HISTOGRAM, except ADABAS returns the requested record, too.

1. ADABAS looks in the inverted list for the first value that is equal to or greater than the STARTING FROM value and less than or equal to the THRU value indicated on the READ statement. When we do not specify a starting value, ADABAS starts at the beginning of the inverted list.
2. Converts the ISN on the inverted list entry through the address converter and retrieves the record from data storage.[13]
3. Returns the record and the following additional information:
   a) *COUNTER, which is a count of the number of the times through the READ loop.
   b) *ISN, which contains the ISN of the record returned
4. Repeats steps 2 and 3 until all ISNs on that inverted list entry are processed.
5. Looks at the next entry in the inverted list. If the value is less than or equal to the THRU value[14], processes the entry as in step 2.
6. Continues processing inverted list entries until the value is greater than the THRU value or ADABAS reaches the end of the inverted list.

[13] To retrieve the record, ADABAS must first retrieve the data block in which the record resides.

[14] When a THRU or ENDING AT value is specified.

We could have used a FIND statement to produce the HR employee list, but the sequence would be different. The report would look like the report below, because the FIND returns the record in the order the ISNs sit in the inverted list entry.

```
    LAST-NAME           FIRST-NAME
    ------------------- ----------------
    Kirkland            Sally
    Fine                Lawrence
```

The FIND would also cause ADABAS to do more work, because it must first build an ISN list in working storage, before returning any records to the program.

The WHERE clause is available for use with READ statements. It works much like an ACCEPT or REJECT statement. If you have an ESCAPE BOTTOM condition in a READ loop, like in our example above, it would be better to use an ACCEPT or REJECT statement instead.

Natural allows us to code READ statements using WITH instead of STARTING FROM, like this:

```
  HR-EMPLOYEES-USING-WITH.
READ EMPLOYEES-FILE WITH DEPARTMENT = 'HR'
  DISPLAY EMPLOYEES-FILE.LAST-NAME
    EMPLOYEES-FILE.FIRST-NAME
END-READ /* HR-EMPLOYEES-USING-WITH.
```

At first glance, this code looks like it will read only records where DEPARTMENT equals 'HR'. It's a trick. Using WITH on a READ statement is exactly like using STARTING FROM. ADABAS will process the inverted list entries *starting from 'HR'*, but will continue through the inverted list until the end of the list. It is a good habit to use STARTING FROM, instead of WITH, on a READ to avoid confusion.

One way to set the ending point of a READ is with logic like this:

```
  HR-EMPLOYEES-USING-ESCAPE.
READ EMPLOYEES-FILE BY DEPARTMENT-NAME-KEY
  STARTING FROM 'HR'
  IF EMPLOYEES-FILE.DEPARTMENT NE 'HR'
    ESCAPE BOTTOM (HR-EMPLOYEES-USING-ESCAPE.)
  END-IF
  DISPLAY EMPLOYEES-FILE.LAST-NAME
      EMPLOYEES-FILE.FIRST-NAME
END-READ /* HR-EMPLOYEES-USING-ESCAPE.
```

Or we can use the THRU or ENDING AT like this:

```
  HR-EMPLOYEES-USING-THRU.
READ EMPLOYEES-FILE BY DEPARTMENT-NAME-KEY
  STARTING FROM 'HR' THRU 'HR999999'
  DISPLAY EMPLOYEES-FILE.LAST-NAME
      EMPLOYEES-FILE.FIRST-NAME
END-READ /* HR-EMPLOYEES-USING-THRU.
```

The HR-EMPLOYEES-USING-ESCAPE example will work exactly like the HR-EMPLOYEES-USING-THRU example. ADABAS will read the same records and the program will produce the same report. There are two differences, when using the THRU clause. 1) We must define the descriptor in the view being read. 2) Natural will not let us UPDATE the view being read.

READ DESCENDING

As with the HISTOGRAM, programmers have for years wanted to be able to READ LOGICAL in descending order.[15] At last, we can. By adding the key word DESCENDING to READ LOGICAL, we can tell ADABAS to go through the inverted list in reverse order. Here's an example:

```
  READ-EMPLOYEES-IN-DESCENDING-ORDER.
READ EMPLOYEES IN DESCENDING SEQUENCE BY EMPLOYEE-ID
    DISPLAY EMPLOYEES.EMPLOYEE-ID
      EMPLOYEES.LAST-NAME (AL=30)
END-READ /* READ-EMPLOYEES-IN-DESCENDING-ORDER.
```

This example will produce this report. Notice that the sequence of IDs is the reverse of the order in the inverted list:

```
EMPLOYEE-ID            LAST-NAME
------------ ------------------------------
   951001    Abdul
   910462    Smithe
   841107    Andersen
   840478    Kirkland
   811249    Branson
   790152    Fine
   750429    Rumplestiltskin
```

[15] The ability to HISTOGRAM and READ in descending order eliminates the need to use date nines compliment fields in order to get records in reverse date order. This technique has been used in many places and situations and required us to carry an extra field and descriptor. That additional overhead is no longer needed.

Also with a READ LOGICAL, we can dynamically set the direction through the inverted list from descending to ascending and vice versa by using the key words IN VARIABLE SEQUENCE.

```
IF #SHOW-DATA-IN-DESCENDING-SEQUENCE
  ASSIGN #READ-DIRECTION = 'D'
ELSE
  ASSIGN #READ-DIRECTION = 'A'
END-IF
  READ-EMPLOYEES-IN-VARIABLE-ORDER.
READ EMPLOYEES IN VARIABLE #READ-DIRECTION SEQUENCE
  BY EMPLOYEE-ID
  DISPLAY EMPLOYEES.EMPLOYEE-ID
    EMPLOYEES.LAST-NAME (AL=30)
END-READ /* READ-EMPLOYEES-IN-VARIABLE-ORDER.
```

Notice that we must define a variable (#READ-DIRECTION in this case) to hold the value A for ascending and D for descending. Any other value will result in an execution error.

Changing the value of #READ-DIRECTION from within the READ loop will not change the direction of the READ.

Defining the User View for a READ Logical

Define only the fields needed for processing. We don't need to include descriptors that are used only in the BY clause of the READ, of a descriptor used with a THRU.

READ PHYSICAL (NO DESCRIPTOR)

READ physical is the least expensive way to read many records. It is much like a sequential file read. In processing a physical read, ADABAS goes directly to data storage and reads the records as they sit in storage. It does not go through an inverted list. It does not go through the address converter.[16] Minimal overhead, maximum efficiency.

ADABAS returns the records in no discernible pattern, not even ISN order. If the order that ADABAS returns the records is important, do not use READ physical.

If you expect to read at least 30**%** of a file, use READ physical and REJECT the records you don't want. It will be faster than a FIND or a logical READ, because of the saved overhead. The **30% rule is a rule of thumb**. The actual percentage depends on things like block size in data storage, record lengths and other things. 30% is a practical guide.

Here's an example of a physical READ:

```
  ALL-EMPLOYEES.
READ EMPLOYEES-FILE
  DISPLAY EMPLOYEES-FILE.LAST-NAME
    EMPLOYEES-FILE.FIRST-NAME
END-READ /* ALL-EMPLOYEES.
```

Defining the User View for a READ Physical

Define only the fields needed for processing.

[16] There is an additional savings for the READ physical, because ADABAS will retrieve each data storage block once. In processing a FIND, READ logical or READ BY ISN, ADABAS might re-read some blocks. To gain further savings with a physical READ, use MULTI-FETCH. This will cause ADABAS to read in more than one data storage blocks at a time and do less internal I/O.

GET

The GET statement allows us to tell ADABAS to retrieve a record using the ISN. This is the most efficient way to retrieve a single, specified record. The trick is that we must know what the ISN is. Most often, we use GET to re-GET a record that we had retrieved in a previous I/O statement. For example, we retrieve an employee record in the beginning of our program like this:

```
  GET-RECORD-WE-MIGHT-NEED-AGAIN.
FIND (1) EMPLOYEES-FILE WITH EMPLOYEE-ID = #EMPLOYEE-ID
  ASSIGN #HOLD-ISN
    = *ISN (GET-RECORD-WE-MIGHT-NEED-AGAIN.)
  process employee
END-FIND /* GET-RECORD-WE-MIGHT-NEED-AGAIN.
```

Later in the processing, the record is no longer in the buffer or view and we need it again. So we do this:

```
  GET-RECORD-AGAIN.
GET EMPLOYEES-FILE #HOLD-ISN
```

We can specify the ISN in a variable or as a numeric literal. If we ask ADABAS to retrieve an ISN that is not in the Address Converter, ADABAS issues an execution error message[17] and Natural stops program execution.

ADABAS takes these steps to process a GET:

1. Uses the Address Converter to find the appropriate data storage block.
2. Retrieves the record from data storage.[18]
3. Returns the record.

Defining the User View for a GET

Define only the fields needed for processing.

[17] Natural error number 3113 or ADABAS response code 113.

[18] To retrieve the record, ADABAS must first retrieve the data block in which the record resides.

READ BY ISN

READ BY ISN is the almost as inexpensive as the READ physical, but has limited uses. Usually, we won't need records in ISN order. Use READ BY ISN when you specifically want the records in ISN order or you want a particular ISN. In most other cases, use READ physical, because it is cheaper.

These are the steps ADABAS takes to process a READ BY ISN:

1. Reads the ISN from the Address Converter.
2. Retrieves the record from data storage.[19]
3. Returns the record and the following additional information:
   a) *COUNTER, which is a count of the number of the times through the READ BY ISN loop.
   b) *ISN, which contains the ISN of the record returned
4. Goes to the next ISN in the Address Converter and repeats steps 2 and 3 until all ISNs for the file have been processed.

Notice that ADABAS bypasses the inverted list. There is no descriptor; so there is no need for the inverted list. We save the cost of that overhead.

In the following example, we are retrieving a single record using the value in #EMPLOYEE-ISN. We would have gotten that value in a previous I/O statement.

```
  READ-ONE-BY-ISN.
READ (1) EMPLOYEES-FILE BY ISN
  STARTING FROM #EMPLOYEE-ISN THRU #EMPLOYEE-ISN
  DISPLAY EMPLOYEES-FILE.LAST-NAME
    EMPLOYEES-FILE.FIRST-NAME
END-READ /* READ-ONE-BY-ISN.
```

Many people suggest the above method to retrieve a single record using the ISN, instead of a GET statement. Unlike the GET, the above logic will not cause an execution error if the ISN is not on file.

Defining the User View for a READ BY ISN

Define only the fields needed for processing.

[19] To retrieve the record, ADABAS must first retrieve the data block in which the record resides.

Getting Data INTO the Database

So far we've talked about retrieving data from the database. How the heck did it get in there in the first place? Here are the statements that we use in Natural to maintain the database.

STORE

To add a record to the database, use a STORE statement. It looks like this:

```
   ADD-EMPLOYEE.
STORE EMPLOYEE-FILE
```

ADABAS will store a record on the employee file with whatever data is in the EMPLOYEE-FILE view. There's not much to it. Move the data to the fields in the programmatic user view and execute the STORE statement. When it processes the STORE, ADABAS adds the record to data storage, updates the Address Converter and updates inverted lists for the employee file as needed.

If we attempt to store a non-unique value in a unique descriptor, ADABAS will issue an execution error and stop program execution.[20]

Defining the User View for a STORE

To save overhead, define only the fields that will contain data. There is no need to include other fields in the view; ADABAS will assume excluded fields are null.

[20] Natural error number 3098 or ADABAS response code 98.

UPDATE

The UPDATE statement is also a simple statement, but much more is involved than with a STORE.

```
UPDATE (GET-RECORD-FOR-UPDATE.)
```

Notice the UPDATE has a label that refers to a previous I/O statement, not a view name like the STORE. To update an ADABAS record, we must first retrieve it from the database. ADABAS will hold the record, when we get it for UPDATE. This stops anyone else from updating or deleting the record while our program has it. It also means that the record is unavailable to others until our program or ADABAS releases the hold on the record. Sometimes, other programs will not even have read-only access.

The logic to READ, GET or FIND a record for update looks just like a read-only READ, GET or FIND. If we refer an UPDATE statement back to an I/O statement that retrieved the record, that I/O statement becomes a READ, GET or FIND with hold. ADABAS will hold every record it reads, even when the UPDATE is within a condition and is not executed for all records.[21]

The GET-RECORD-JUST-TO-LOOK example below is just a FIND. The GET-RECORD-FOR-UPDATE example is a FIND with hold.

```
  GET-RECORD-JUST-TO-LOOK.
FIND (1) EMPLOYEES-FILE WITH EMPLOYEE-ID = #EMPLOYEE-ID
END-FIND /* GET-RECORD-WE-MIGHT-NEED-AGAIN.

  GET-RECORD-FOR-UPDATE.
FIND (1) EMPLOYEES-FILE WITH EMPLOYEE-ID = #EMPLOYEE-ID
  UPDATE (GET-RECORD-FOR-UPDATE.)
END-FIND /* GET-RECORD-FOR-UPDATE.
```

Here we read the entire employees file and update only a few records. The UPDATE is conditional, but ADABAS will hold all the records we read. This creates wasteful overhead in ADABAS. If we have to read many records and only a few to update, it is cheaper to retrieve the record a second time. Like this:

```
  GET-CHEAPER-UPDATE.
READ EMPLOYEES-FILE
  IF EMPLOYEES-FILE.HIRE-DAY = '01' OR = '15'
      GET-REC-AGAIN.
    GET EMPLOYEES-FILE *ISN (GET-CHEAPER-UPDATE.)
    UPDATE (GET-REC-AGAIN.)
  END-IF
END-READ /* GET-CHEAPER-UPDATE.
```

Now the UPDATE refers back to the GET statement. Only the records we GET are held instead of all the records we READ. **If we must read many records and expect to update less than half, we should consider re-GETting the record for update.** The same holds for DELETEs.

If we attempt to update a record with a non-unique value in a unique descriptor, ADABAS will issue an execution error and stop program execution.[22]

Defining the User View for a UPDATE

Define only the fields that will contain data. Natural will not let you update a programmatic user view that contains a super or sub-descriptor.

[22] Natural error number 3098 or ADABAS response code 98.

DELETE

To completely remove a record from a file, use the DELETE statement. Several things about the DELETE statement are exactly like the UPDATE:

1) We must first retrieve the record from the file.

2) ADABAS will hold the record. No one else can retrieve the record until our program or ADABAS releases the record.

3) If we point a DELETE statement back to an I/O statement that retrieved a record, that I/O statement becomes a READ, GET or FIND with hold.

4) If we use a FIND or READ to get a record for deletion, the DELETE statement must be within the FIND or READ loop. If we use a GET to retrieve a record, the GET statement must appear in the program before the DELETE statement.

Here's an example of DELETE in action:

```
  GET-RECORD-FOR-DELETION.
FIND (1) EMPLOYEES-FILE WITH EMPLOYEE-ID = #EMPLOYEE-ID
  DELETE (GET-RECORD-FOR-DELETION.)
END-FIND /* GET-RECORD-FOR-DELETION.
```

Sometimes, the DELETE statement is not best way to delete a large number of records[23] from a file. If the file has many descriptors, the overhead of updating the inverted lists can be large. In some cases, it is more efficient to unload the file and reload the file leaving out the unwanted records. A number of database factors guide a decision like this. If you expect to delete tens of thousands of records from a file, talk to the DBAs about the relative costs of different methods.

DON'T HOLD RECORDS YOU DON'T DELETE

As with UPDATEs, be careful of conditional DELETEs. **If you have to read many records and will only DELETE a few, consider GETting the record a second time.** See DON'T HOLD RECORDS YOU DON'T UPDATE on page 30 for more details.

Defining the User View for a DELETE

Since a DELETE removes a record from the database, it doesn't matter what fields are in the programmatic user view or what data they contain. In fact, we don't need any fields in the DELETE view.

[23] By large number of records, I mean tens of thousands or more in a single operation or job. The amount of overhead depends on the number and complexity of descriptors and other factors. For a file without many or complex supers a million records might not be considered a large number.

END TRANSACTION

END TRANSACTION (A.K.A. ET) is the most important I/O statement in maintaining an ADABAS database. The END TRANSACTION finalizes updates to the database. When a program tells ADABAS to STORE, UPDATE or DELETE a record, ADABAS holds those transactions in a buffer. ADABAS commits those changes to the database or makes them permanent, when it receives an ET command.

The END TRANSACTION allows us to define a logical transaction. A logical transaction consists of one or more STOREs, UPDATEs and DELETEs in combination. The transactions need not go against the same file. Let's look at an example using address file.[24]

The address file contains current and former addresses for our employees. When an employee gets a new address, we add a new record and change the current indicator to no for the old address. These two updates to the database comprise a logical transaction, because they must both happen or neither should happen. Otherwise, the current indicator on the addresses would be out of synch and we lose data integrity. We use ET logic to ensure that integrity.

In our example, Billy Branson moves to a new address in Westview. Here's the code to make that happen:

```
0090 ASSIGN ADDRESS-FILE.EMPLOYEE-ID = '811249'
0100 ASSIGN ADDRESS-FILE.STREET = '825 W. SIXTH ST.'
0110 ASSIGN ADDRESS-FILE.CITY = 'WESTVIEW'
0120 ASSIGN ADDRESS-FILE.CURRENT-IND = 'YES'
0130 ASSIGN ADDRESS-FILE.ADDRESS-COUNT = 2
0140 STORE ADDRESS-FILE
          ...
          ...
          ...
0280   GET-BRANSON-EMP-REC.
0290 FIND (1) ADDRESS-FILE
0300   WITH EMPLOYEE-ADDRESS-KEY = '811249'
0310   ASSIGN ADDRESS-FILE.CURRENT-IND = 'NO'
0320   UPDATE (GET-BRANSON-EMP-REC.)
0330 END-FIND /* GET-BRANSON-EMP-REC.
0340 END TRANSACTION
```

If the system crashed between line 140 and line 320, ADABAS would back out the STORE on line 140 and the data would still be in synch. If all updates to the database were immediate and permanent, we would not be able to ensure the data integrity as easily.

---

[24] Defined in Appendix D.

END TRANSACTIONs in Batch

In the olden days,[25] we had to take great care in placing ET's. The END TRANSACTION finalized all pending database updates by flushing the buffer that held those transactions. An ET was the most expensive ADABAS internal I/O operation. Great debates raged whether to issue ET's after every 50 updates or every 100 updates. ADABAS no longer flushes the buffer for every ET. Now it sets pointers and flushes the buffers at the most efficient time.

This performance improvement allows us to issue ET's when they best suit the logic of our programs. We can afford to issue an END TRANSACTION after every on-line logical transaction. We may not want to do that in batch processing with mass updates. Even cheap ET's add up when updating millions of records. We also have to consider restart logic. END TRANSACTION helps here, too.

ET Data and Restarting Batch Processes

Each time we issue an ET, we can ask ADABAS to retain some restart data for us. This data could be a record key and control information that we use to restart a sequential process. The key to this ET data[26] is the user ID or a program ID. We can retrieve this transaction data at the beginning of the batch program, examine it and determine where or whether to restart the batch. The statement to retrieve the transaction data is **GET TRANSACTION DATA.** Here's an example of transaction data used to restart a program that archives old addresses.

```
0010 DEFINE DATA LOCAL
0020 01 ADDRESS-FILE VIEW OF ADDRESS-FILE
0030   02 EMPLOYEE-ID
0040   02 CURRENT-IND
0050 01 #ET-DATA        (A9)
0060 01 REDEFINE #ET-DATA
0070   02 EMPLOYEE-ID   (A6)
0080   02 ADDRESS-COUNT (N2)
0090   02 SUCCESS-IND   (A1)
0100 01 #START-ID (A6)
0110 END-DEFINE
0120   GET-RESTART-DATA.
0130 GET TRANSACTION DATA #ET-DATA
0140 IF #ET-DATA.SUCCESS-IND = 'Y'
0150   WRITE 'LAST RUN SUCCESSFUL. NO RESTART'
0160 ELSE
0170   ASSIGN #START-ID = #ET-DATA.EMPLOYEE-ID
0180 END-IF
0190   REMOVE-OLD-ADDRESS.
0200 READ ADDRESS-FILE BY EMPLOYEE-ID
0210     STARTING FROM #START-ID
```

[25] When Benjamin Franklin was the DBA for the Continental Congress.

[26] ADABAS will accept up to 2000 bytes.

```
0210   IF ADDRESS-FILE.CURRENT-IND = 'NO'
0220     WRITE WORK 1 ADDRESS-FILE
0230     ASSIGN #ET-DATA.EMPLOYEE-ID
0240       = ADDRESS-FILE.EMPLOYEE-ID
0240     ASSIGN #ET-DATA.ADDRESS-COUNT
0250       = ADDRESS-FILE.ADDRESS-COUNT
0260     DELETE (REMOVE-OLD-ADDRESS.)
0270     END TRANSACTION #ET-DATA
0280   END-IF
0290 END-READ /* REMOVE-OLD-ADDRESS.
0300 ASSIGN #ET-DATA.SUCCESS-IND = 'Y'
0310 RESET #ET-DATA.EMPLOYEE-ID
0320 END TRANSACTION #ET-DATA
0330 END
```

At the beginning of the program (Line 0120), we look for transaction data left by the last run of this program. If there is no transaction data, the fields in #ET-DATA will be spaces. After each DELETE (Line 0270), we record the employee ID and address count in ET data. If the process abends, #ET-DATA will contain the key of the last employee address successfully processed. When we successfully end of the process, we issue another ET. The last END TRANSACTION is only to populate ET data with the success switch set to yes.

Notice in our example we are doing an ET after every DELETE. Even though ET's are cheaper than they used to be, an END TRANSACTION after every database update is not a good idea for a batch process. **The rule of thumb is an ET after every 10 to 20 updates**. The actual number also depends on the size of the **hold queue**. The hold queue is a buffer, where ADABAS holds records and pending transactions. That means ET's must be frequent enough that we don't fill up the hold queue, but not so frequent that we waste resources.

Defining the User View for a END TRANSACTION

It's a trick question. END TRANSACTION finalizes all pending transactions for a job or user in the active database. ET is not file specific.

BACKOUT TRANSACTION

BACKOUT TRANSACTION (BT) allows us to tell ADABAS that we changed our minds. When ADABAS receives a BT, it backs out any pending UPDATEs, DELETEs and STOREs, and releases all held records. Perhaps we issued updates to one file and then detected an edit error. We can back out the update. Here's how it works with our address file transaction.

```
0090 ASSIGN ADDRESS-FILE.EMPLOYEE-ID = '811249'
0100 ASSIGN ADDRESS-FILE.STREET = '825 W. SIXTH ST.'
0110 ASSIGN ADDRESS-FILE.ADDRESS-COUNT = 2
0120 ASSIGN ADDRESS-FILE.CITY = 'WESTVIEW'
0125 ASSIGN ADDRESS-FILE.CURRENT-IND = 'YES'
0130 STORE ADDRESS-FILE
          ...
          ...
          ...
0280   GET-BRANSON-EMP-REC.
0290 FIND (1) ADDRESS-FILE
0295   WITH EMPLOYEE-ADDRESS-KEY = '811249'
0300   IF NO RECORDS FOUND
0310     WRITE 'expected record not found.'
0320     BACKOUT TRANSACTION
0330     STOP
0340   END-NOREC
0300   ASSIGN ADDRESS-FILE.CURRENT-IND = 'NO'
0310   UPDATE (GET-BRANSON-EMP-REC.)
0320 END-FIND /* GET-BRANSON-EMP-REC.
0330 END TRANSACTION
```

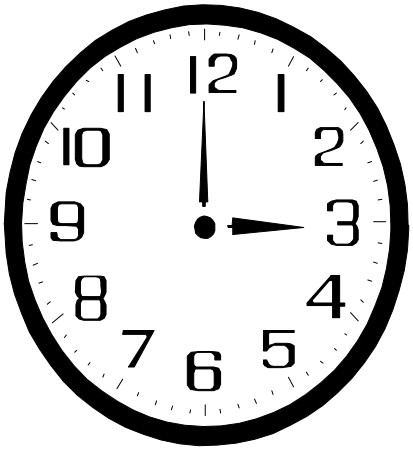
We added the new address, but found an edit error. We can't let the new address record stay on the database, so we issue a BT.

Defining the User View for a BACKOUT TRANSACTION

BACKOUT TRANSACTIONs are not file specific. We don't need a view.

IN LIMBO -- BETWEEN ETs

Between the time we STORE, UPDATE or DELETE a record and the time we issue an END TRANSACTION or BACKOUT TRANSACTION, ADABAS keeps the database updates in the hold queue. ADABAS waits for us to make up our minds with an ET or BT. If the system crashes ***or*** if ADABAS gets tired of waiting on us, ADABAS backs out all the pending transactions.



There is a time limit on how long ADABAS will keep transactions in the hold queue. Once the limit is reached, ADABAS acts as if we issued a BT. DBAs set the time limit.

So while ADABAS is waiting, what is the status of the data we update? While database update transactions are in the hold queue, ADABAS makes it look as if the transactions have been permanently applied to the database. If our program or other programs access the data we have updated, they will see the new data. Let's look at an example. We store a new record on the employees file and read it immediately afterwards.

```
0330 ASSIGN EMPLOYEES-FILE.EMPLOYEE-ID = '124673'
0340 ASSIGN EMPLOYEES-FILE.LAST-NAME = 'COHEN'
0350 ASSIGN EMPLOYEES-FILE.FIRST-NAME = 'IMELDA'
0360 STORE EMPLOYEES-FILE
0370   GET-REC-JUST-STORED.
0380 FIND (1) EMPLOYEES-FILE WITH EMPLOYEE-ID = '124673'
0390   WRITE 'FOUND EMP' EMPLOYEES-FILE.EMPLOYEE-ID
0400 END-FIND /* GET-REC-JUST-STORED.
0410 BACKOUT TRANSACTION
0420   GET-REC-JUST-STORED-AGAIN.
0430 FIND (1) EMPLOYEES-FILE WITH EMPLOYEE-ID = '124673'
0440   IF NO RECORDS FOUND
0450     WRITE 'REC NOT FOUND'
0460   END-NOREC
0470 END-FIND /* GET-REC-JUST-STORED-AGAIN.
```

After line 360, the new record is on the database for all intents and purposes. After line 410, it is as if we never stored it. Here's the output for the above logic:

```
MORE
Page      1                                12/10/01  13:37:46

FOUND EMP 124673
REC NOT FOUND
```

ADABAS C for Me

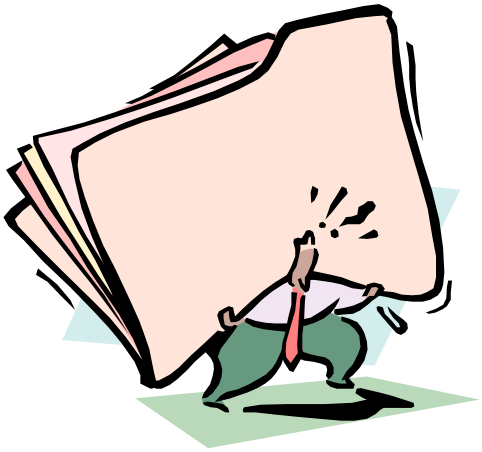
ADABAS is a world class database management system. It has been in use continuously for over thirty years and handles some of the largest databases in the world. World class Natural programmers use ADABAS well, because they know how ADABAS works. They use that knowledge to write efficient I/O logic, but they also know how to avoid using it poorly. Selecting the appropriate Natural I/O statement is the key.

Selecting the appropriate Natural I/O statement and using it properly is the key to success with ADABAS C.

Don't be like this guy. KNOW WHAT ADABAS IS DOING WHEN YOU ASK FOR DATA.

It is just as important for good Natural programmers to know their data. Different file characteristics make the FIND NUMBER more efficient than the HISTOGRAM, sometimes. Sometimes, it is smarter to use READ LOGICAL instead of READ PHYSICAL. Other times READ PHYSICAL is better, and even other times READ BY ISN is better. It depends on the data.

Inefficient I/O can make your ADABAS database a burden on your organization.



Study the design and content of your files so you can use the best I/O statement for the job.

KNOW YOUR DATA and fly through your database.

Debug Utility

I've included this discussion of the Natural TEST utility as a bonus. Even when we are sure we know what our program is doing and we are sure we know what ADABAS is doing with out our I/O requests, we need a way to validate that. TEST gives us a number of tools to help.

Why use TEST?

With the Natural TEST utility, we can test and debug our Natural programs as they execute. We can interrupt the execution of our Natural module at a specified line or lines or upon the change in the value of a specified variable or variables. Once TEST interrupts the program execution, we can

1) examine the source code

2) execute the module line by line

3) examine and change the contents of variables, before continuing execution.

We can do all this without changing the source code of our program and without re-stowing our program.[27] Also by turning on TEST, we can trap Natural errors

. DBLOG is a subset of the TEST utility, which allows us to review ADABAS calls that our programs performed. DBLOG replaces ADALOG, a previous utility in Natural.

Keep in mind that when TEST interrupts the execution of our program; we are actually executing the program. We are not simulating, testing or pretending. If our program updates files, the files will actually be updated.

```
09:26:55                *** Natural TEST UTILITIES ***               02/19/01
 Test Mode OFF                 -  Debug Main Menu  -             Object

                      Code  Function
                      ----  ----------------------------
                        T   Set Test Mode ON
                        E   Debug Environment Maintenance
                        S   Spy Maintenance
                        B   Breakpoint Maintenance
                        W   Watchpoint Maintenance
                        C   Call Statistic Maintenance
                        X   Statement execution statistics maintenance
                        L   List Object Source
                        V   Variable Facility
                        ?   Help
                        .   Exit
                      ----  ----------------------------
                Code .. _   Object Name .. ________


 Command ===>
 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12---
       Help     Exit Last     Flip              Canc
```

[27] To facilitate this, we must create the NATURAL source and object code at the same time. The STOW command will ensure this.

Setting up

Breakpoints

To specify lines, at which we want to interrupt execution, we define breakpoints. TEST will interrupt execution of the program, before Natural executes the line with a breakpoint set. The following steps explain defining breakpoints:

1. Access the TEST main menu by entering TEST at the command line or NEXT prompt or by enter the terminal command, %<TEST.[28]
2. At the main menu, select option T to turn on the TEST utility. Notice that option T then becomes 'turn off TEST'.
3. At the main menu, select L and enter the name of our Natural program.
4. When TEST presents the source code, find the line at which we want to interrupt execution.[29]
5. Enter SE for 'set breakpoint' next to that line of code and press enter.[30]
6. Exit TEST. PF3 will get us out of most places in TEST.
7. Execute the program. When the program gets to that line, Natural will interrupt execution and present the TEST window. We'll discuss the TEST window later.

Watchpoints

To specify the variables, the change of which will interrupt execution, we define watchpoints. TEST will interrupt execution of the program when the variable with the watchpoint changes in value. If you have a field, which changes and you're not sure where the change occurs, setting a watchpoint is a good way to find out. The following steps explain defining watchpoints:

1. Access the TEST main menu by entering TEST at the command line or NEXT prompt or by enter the terminal command, %<TEST.
2. At the main menu, select option T to turn on the TEST utility, if it is not on already.
3. At the main menu, select option W.
4. At the watchpoint maintenance menu, enter S and the program name.
5. Once TEST presents the list of variables, find the variable, for which we want the watchpoint.[31] Notice that all the variables, including those defined in external objects, are listed.
6. Enter SE for 'set watchpoint' next to that variable and press enter.
7. Exit TEST.
8. Execute the program. When the contents of that variable change, Natural will interrupt execution and present the TEST window.

---

[28] This last one is particularly handy. It allows you to access the debugger even though you may have already started the program you want to debug.

[29] See the cheat sheet for navigation commands when viewing source code.

[30] We can set breakpoints at executable lines of code only. For example, we cannot set a breakpoint on a comment line. Note that where loops or I/O statements have labels, the line with the label is the executable line.

[31] See the cheat sheet for navigation commands when viewing the variable list.

Spies

Breakpoints and watchpoints are collectively called spies. Spy maintenance, option S from the main menu, allows us to display a list all of breakpoints and watchpoints. From that list, we can activate (AC), deactivate (DA), delete (DE) and modify (MO) them by entering the appropriate action code.

The spies that we set go away when we log off, but we can save these spies by going to the TEST main menu and taking option E for Debug Environment Maintenance. This function allows us to save the debug environment, the spies, and reload them at a later time.

Natural Errors

When TEST is turned on and a Natural execution error occurs, the TEST window will pop up. From there we can view the source code at the line that *really* caused the error[32] and view the contents of the program's variables. TEST will do this even when have not set or activated any spies.

Call Statistics

Another useful part of TEST is the Call Statistics. With this tool you can determine exactly which modules have been FETCHed or CALLNATed. This is particularly useful with complicated processes. Here's how to turn call statistics on.

1) Go to the TEST main menu and select option C. This takes you to the Call Statistics Maintenance menu.

2) Turn TEST with option T.

3) Turn on call statistics with option C.

4) Execute your complicated process.

5) Return to the TEST main menu (The terminal command, %<TEST, is an easy way to do this that does not require you to leave the process you are testing.) and select option C.

6) From the Call Statistics Maintenance menu, select option 2. This will display all the modules that were invoked since you turned on call statistics. In this way you can discover which modules are invoked without having to set many breakpoints in the various modules.

---

[32] NATURAL error messages don't always point to the program and line that actually caused the error. TEST usually does.

TEST Window

We've set spies and turned TEST on. We've executed our program and hit a spy or a Natural execution error has occurred. TEST interrupts the execution of our program and the TEST window pops up. From there we have several options:

- G lets us just continue executing our program.
- L lets us view the source code starting at the line that was last executed.
- M lets us go to the TEST main menu.
- R lets us turn off TEST and continue executing our program.
- V lets us view the list of variables and their contents.

L is for List

Upon selecting L, TEST will put us into the program listing showing the line, which caused the break, in the center of the screen. You'll see 'Lastline' in the message column to the right of the last line executed. Within the L function, we can execute the program one line at a time by pressing PF2, the step key. This allows us to see which lines are executed and in what order they are executed. In other words, we can follow the logic of our program as it executes.

We can set additional spies, before continuing execution of the program.

V is for Variables

Within the V function, we can display and modify the contents of the variables, internally and externally defined, in the executing program. We can also view system variables like *ISN, *COUNTER and *NUMBER. Since a program can have many instances of these system variables, we need a way to tell which is which. In later versions of Natural, these system variables appear in the variable list with line numbers. In older versions, these system variables have no references. We can't tell one *COUNTER from another.

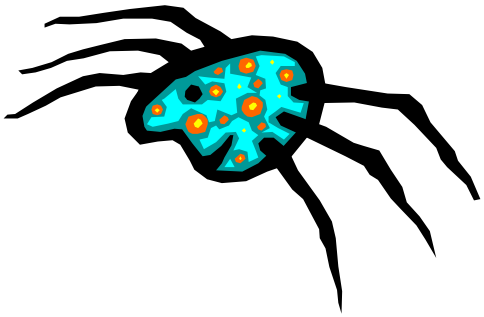
The variable list shows the variable name, format and length and the contents. We see the contents as alphanumeric by default, but we can change the display to hexadecimal by pressing the Hex key, PF11. PF10 will take us back to alphanumeric display.

If the contents of the field won't fit in the Contents column or the field is an array, enter DI to the left of the field and press enter. The resulting screen will show the whole field. For arrays, PF7 and PF8 will scroll through the contents of the different occurrences.

To modify the contents of the field, enter MO to the left of the field and press enter. The resulting screen looks like the display screen, but we can enter new data. By pressing PF11, we can enter data in hexadecimal format.[33]

If you're trying to track down a field that caused a data exception, look for 'Invalid Format' in the Content column.

Track 'em down with DEBUG



[33] We cannot change or display the contents of a control variable.

DBLOG (formerly known as ADALOG)

DBLOG will report the ADABAS commands that we've executed. Notice I said ADABAS commands not Natural I/O statements. A single Natural statement can trigger several ADABAS commands to the database. Knowing what ADABAS commands our programs execute helps us write more efficient I/O statements. To use DBLOG:

1. Enter TEST DBLOG ON at the command line or NEXT prompt.

2. Execute the program or programs for which we want to record ADABAS data.

3. Enter TEST DBLOG at the command line or NEXT prompt. DBLOG shows a list of the ADABAS commands executed since we turned on DBLOG. Here's an example:

```
15:40:21                ***** Natural Test Utilities *****            12/15/01
 User Z8342                        - DBLOG Trace -                Library X74
 M No   Cmd DB FNR Rsp     ISN         ISQ     CID  CID(Hex) OP   Pgm    Line
 _  233 RC 151 252                              ?Ø?? 26800701 SI CNUEXIST 2680
 _  234 L3 151 252            65622             ???? 27200701  HV CNUEXIST 2720
 _  235 RC 151 252                              ???? 27200701 SI CNUEXIST 2720
 _  236 L9 151 245                            1 ?Ø?? 26800701 HV CNUEXIST 2680
 _  237 RC 151 245                            1 ?Ø?? 26800701 SI CNUEXIST 2680
 _  238 L9 151 245                            1 ?Ø?? 26800701 HV CNUEXIST 2680
 _  239 RC 151 245                            1 ?Ø?? 26800701 SI CNUEXIST 2680
 _  240 L9 151 245                            1 ?Ø?? 26800701 HV CNUEXIST 2680
 _  241 RC 151 245                             1 ?Ø?? 26800701 SI CNUEXIST 2680
 _  242 L9 151 252                            1 ?Ø?? 26800701 HV CNUEXIST 2680
 _  243 RC 151 252                              ?Ø?? 26800701 SI CNUEXIST 2680
 _  244 L3 151 252            65622             ???? 272007 01 HV CNUEXIST 2720
 _  245 RC 151 252                              ???? 27200701 SI CNUEXIST 2720
 _  246 L3 151 252   21       21289             ???? 12300401 HV CNUSRCRD 1230
 _  247 RC 151 252                              ???? 12300401    CNUSRCRD 1230
 _  248 RC 151 252                              ???? 12300401    CNUSRCRD 1230
 _  249 RC 151 247                              ???? 31200201    CSGENPGM 1230

 Command ===>
 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12
       Help  Print Exit  Top   Posi  Bot    -     +                     Canc
```

This report includes the ADABAS command code, the database and file numbers, the ISN, program name and line number among other things. See Appendix C for a list of ADABAS commands.

DBLOG Snapshot

The DBLOG Snapshot facility is good for two things:

1) Discovering exactly when an ADABAS command is executed.

2) Viewing the ADABAS buffers when an ADABAS command is executed or an error occurs. This is mostly a DBA thing. Let them enjoy it. Looking at buffer contents keeps them happy, busy and off the streets.

To turn on the snapshot facility, enter TEST DBLOG MENU from the command line or NEXT prompt. On the DBLOG menu, enter 'S' for snapshot and the command you want

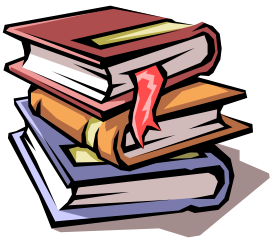
ADABAS to snapshot. Then execute your program(s). The first time ADABAS issues that command, DBLOG stops the currently executing program[34] and shows a screen like this:

```
16:45:13              ***** Natural DBA Utility *****              2001-04-14
                             - Snapshot Report -

 Command Code : ET           Command ID   :    ? 00000004 File Number  : 9700
 Response Code:      0       ISN          :             0
 ISN Low Limit: 00000000     ISN Quantity :             0
 FB Length    : 0000         RB Length    : 0000          SB Length    : 0000
 VB Length    : 0004         IB Length    : 0000          Com. Option 1: H
 Com. Option 2:              Additions 1  :               Additions 2  :    ?
 Additions 3  :              Additions 4  :
 Global FID   : 00000000000000000 Command Time : 0000003 Pgm: X7A4000 Lin: 4620
 Control Block
 0000 * 00D5C5E3 00000004 97000000 00000000 *  NET   ?p         * 0000
 0010 * 00000000 00000000 00000000 00000004 *                 ? * 0010
 0020 * 0000C800 40404040 40404040 00000014 *   H              ? * 0020
 0030 * 00000000 00000000 00000000 00000000 *                   * 0030
 0040 * 00000000 00000000 00000003 00000000 *             ?     * 0040
 0050 * 00000000 00000000 00000000 00000000 *                   * 0050
 0060 * 00000000 00000000 00000000 00000000 *                   * 0060
 Command ===> CB_____________________________________________________________

 Enter-PF1---PF2---PF3---PF4---PF5---PF6---PF7---PF8---PF9---PF10--PF11--PF12-
       Help        Exit  CB    FB    RB     -     +    SB    VB    IB    Canc
```

In this example, I wanted to find out where an ET was issued in a process that used several programs. In the report, I see that the ADABAS issued the ET on line 4620 of X7A4000.

[34] Unlike DEBUG, which only interrupts the execution of the program.

Appendixes



Appendix A – Labels and Qualifiers

I ***strongly*** recommend the use of labels and qualifiers.

Labels

You may have noticed that the examples here use labels on I/O statements rather than line number references. In the old days, we only had line numbers to refer back to a previous statement, like an I/O statement. Many experienced programmers like line number references and are reluctant to use labels.[35]

Use labels.

1) Labels, by definition, are more descriptive than line numbers.

2) Labels can make a program self-documenting, especially if they are not cryptic, like R1.

3) **Labels are more reliable as references.** Sometimes Natural loses track of line numbers in the compiled object, particularly if the program has copy code. Natural does not lose track of labels.

Qualifiers

By qualifiers, I mean prefixes on field names like EMPLOYEES-FILE in this example:

```
DISPLAY EMPLOYEES-FILE.LAST-NAME
```

Qualifiers are view names or other group field names in data definitions.

Use qualifiers.

1) Qualifiers document the origin of the field.

2) Qualifiers can make a program self-documenting.

3) Qualifiers facilitate the use of MOVE BY NAME statements, which are more efficient than individual MOVE statements.

4) Natural requires qualifiers when a field name is not unique within a module.

5) **Qualifiers on database fields can prevent some program bugs,** as in this example:

[35] My habit is to use labels on all I/O statements and loop processes.

```
0010 DEFINE DATA LOCAL
0020 01 EMPLOYEES-FILE VIEW OF EMPLOYEES-FILE
0030   02 EMPLOYEE-ID
0040 01 #OTHER-DATA-STRUCTURE
0050   02 EMPLOYEE-ID (A6)
0060 END-DEFINE
0070   GET-RECORD.
0080 READ EMPLOYEES-FILE BY DEPARTMENT
0090   WRITE EMPLOYEE-ID
0100 END-READ /* GET-RECORD.
0110 END
```

In some cases and using some releases of Natural, line 90 in the compiled object will refer to #OTHER-DATA-STRUCTURE.EMPLOYEE-ID. In other cases, it will refer to EMPLOYEES-FILE.EMPLOYEE-ID. It is not reliable without the qualifier.

In previous versions of Natural, we could expect that references to database fields within an I/O loop would always refer to the file definition or view associated with the inner most active loop. This is no longer true. **Use qualifiers.**

```
0010 DEFINE DATA LOCAL
0020 01 EMPLOYEES-FILE VIEW OF EMPLOYEES-FILE
0030   02 EMPLOYEE-ID
0040 01 #OTHER-DATA-STRUCTURE
0050   02 EMPLOYEE-ID (A6)
0060 END-DEFINE
0070   GET-RECORD.
0080 READ EMPLOYEES-FILE BY DEPARTMENT
0090   IF EMPLOYEES-FILE.EMPLOYEE-ID NE ' '
0100     WRITE EMPLOYEES-FILE.EMPLOYEE-ID
0110   END-IF
0120 END-READ /* GET-RECORD.
0130 END
```

Line 100 in this example will always refer to the EMPLOYEE-ID in the view EMPLOYEES-FILE.

Appendix B – Miscellaneous Notes

C* Variables

We can add C* variables to user views that contain repeating fields like MU's and PE's. Name the C* variable the same as the repeating fields, but with 'C*' at the front. Here's an example using the skill code on our employee file:

```
01 EMPLOYEES-FILE VIEW OF EMPLOYEES-FILE
  02 SKILL-CODE
  02 C*SKILL-CODE
  02 EMPLOYEE-ID
```

C*SKILL-CODE will tell how many occurrences of SKILL-CODE[36] have data on each record we read. For PE's, C* variables tell the last occurrence that has data. In the following example, C*SKILL-CODE saves us from processing null occurrences of the skill code.

```
  EMPLOYEE-SKILL-RPT.
READ EMPLOYEES-FILE BY EMPLOYEE-ID
    ASSIGN #IX = 1
    WRITE-SKILL-CODES.
  REPEAT UNTIL #IX GT EMPLOYEES-FILE.C*SKILL-CODE
     WRITE EMPLOYEES-FILE.EMPLOYEE-ID (IS=ON)
       EMPLOYEES-FILE.SKILL-CODE (#IX)
    ADD 1 TO #IX
  END-REPEAT /* WRITE-SKILL-CODES.
END-READ /* GET-RECORD-FOR-UPDATE.
```

My Update Went Away

Let's say we wrote this program to update a record, but the update seems to go away after a few minutes. Here's the program:

```
  UPDATE-THAT-GOES-AWAY.
FIND (1) ADDRESS-FILE WITH EMPLOYEE-ID = #EMPLOYEE-ID
  ASSIGN ADDRESS-FILE.CURRENT-IND = 'YES'
  UPDATE (UPDATE-THAT-GOES-AWAY.)
END-FIND /* UPDATE-THAT-GOES-AWAY.
END
```

Have you found the bug? There is no ET. ADABAS applies my update to the database, but backs it out when it reaches the hold time limit. Not an unusual mistake.

Other times, the ET is conditional logic that is not being executed. If we have database updates that go away after a short time, the most common reason is a missing or unexecuted END TRANSACTION.

[36] SKILL-CODE is an MU.

My Program Updates Every Record But the Last

We have a program that updates a series of records. It works fine, but it doesn't update the last record on the file. Here's the program:

```
  UPDATE-ADDRESSES.
READ ADDRESS-FILE
  ASSIGN ADDRESS-FILE.CURRENT-IND = 'YES'
  UPDATE (UPDATE-ADDRESSES.)
  ADD 1 TO #ET-COUNTER
  IF #ET-COUNTER GT 15
    END TRANSACTION
    RESET #ET-COUNTER
  END-IF
END-READ /* UPDATE-ADDRESSES.
END
```

We have logic to issue an ET after every fifteenth UPDATE. Why is the last record not updated? We're missing some logic:

```
  UPDATE-ADDRESSES.
READ ADDRESS-FILE
  ASSIGN ADDRESS-FILE.CURRENT-IND = 'YES'
  UPDATE (UPDATE-ADDRESSES.)
  ADD 1 TO #ET-COUNTER
  IF #ET-COUNTER GT 15
    END TRANSACTION
    RESET #ET-COUNTER
  END-IF
END-READ /* UPDATE-ADDRESSES.
IF #ET-COUNTER NE 0
  END TRANSACTION
END-IF
END
```

COMMAND	Natural I/O statement	Action
A1	UPDATE	Updates a record with data in the buffer (view)
BT	BACKOUT TRANSACTION	Backs out a logical transaction or reverses all updates to the database since the last ET
ET	END TRANSACTION	Ends a logical transaction or finalizes all updates to the database since the last ET
E1	DELETE	Removes a record
L1	FIND	Returns a record that met the selection criteria
	GET	Reads the record that has the indicated ISN
	READ BY ISN	Reads records in order by ISN
L2	READ Physical (no descriptor)	Reads records from data storage in the order they are stored physically. In other words, in no discernible order
L3	READ Logical (with descriptor)	Reads records in the order they appear in the inverted list for the descriptor
L4	FIND	same as L1, but with hold
	GET	same as L1, but with hold
	READ BY ISN	same as L1, but with hold
L5	READ Physical (no descriptor)	same as L2, but with hold
L6	READ Logical (with descriptor)	same as L3, but with hold
L9	HISTOGRAM	Searches the inverted list for a particular value
N1	STORE	Adds a record with data in the buffer (view)
RC	None	Closes out a sequential process
S1	FIND	Searches inverted list for records that match the stated criteria
S4	FIND	same as S1, but with hold

EMPLOYEES-ADDRESS-FILE

Employee	Address #	Street	City	Current	ISN
750429	1	125 Morton Drive	Stockbridge	Yes	4
790152	1	731 Hunt Valley Road	Backlash	Yes	5
811249	1	1526 Shady Lane Avenue	Brookfield	Yes	1
840478	1	103 Morning Glory Circle	Centerville	No	8
840478	2	122 Wistful Vista	Centerville	Yes	7
841107	1	1456 Wingate Road	Madison	No	10
841107	2	204 Transylvania Drive	Bancroft	Yes	2
910462	1	1313 Mockingbird Lane	Rochester	Yes	6
951001	2	Hollywood and Vine	Springfield	No	9
951001	1	14321 N. Northwest Street	Sue Falls	Yes	3

Employee Address Key (Employee + Address #)

Value	Count	ISN
7504291	1	4
7901521	1	5
8112491	1	1
8404781	1	8
8404782	1	7
8411071	1	10
8411072	1	2
9104621	1	6
9510011	1	3
9510012	1	9

Navigation

WITHIN THE SOURCE LISTING

- SCAN for a value and use PF5 to repeat scan
- TOP to go to the top of the listing
- BOT to go to the bottom of the listing
- PF9 to go to the line that caused the break
- PF21 to go to the last line executed

WITHIN THE VARIABLE LIST

- to SCAN wait for the next release
- TOP to go to the top of the listing
- BOT to go to the bottom of the listing

TO GO FROM THE VARIABLE LIST TO THE SOURCE LISTING

- PF9 from the variable list will take us to the source listing and the line that caused the break
- PF21 from the variable list will take us to the source listing and the last line executed

TO GO FROM THE SOURCE LISTING TO THE VARIABLE LIST

- DI VA
- DI VA and the variable name will take us to the display screen for that variable
- DI VA and the start of a variable name with * will give us a list a variables

OTHER STUFF

PF4 puts the last command entered on the command line

DBLOG

- TEST DBLOG to turn on DBLOG
- TEST DBLOG ON to reset after DBLOG has been turned on
- TEST DBLOG to view ADABAS report after DBLOG has been turned on
- TEST DBLOG OFF to turn off DBLOG
- TEST DBLOG MENU to go to the DBLOG menu

Appendix F – Final Exam

Here's a test of what we've learned in ADABAS C for Natural Developers. We have a payroll file of all our employees. The only descriptor on the file is SALARY. We need a program to give everyone a 10% raise. What I/O statement do you use to retrieve the records for this program and why?

To obtain an explanation of why each I/O statement is appropriate or inappropriate for this task, send an e-mail to me at jgruber@demainsoft.com.