TOTAL ENGINEERING SERVICES TEAM, INC.

RTU / SCADA SYSTEMS

DATABASE OPTION

Document No. 1180-01

P.O. Drawer 1760
671 Whitney Ave.
Gretna, La. 70054
(504) 368-6792

TOTAL ENGINEERING SERVICES TEAM, INC.

RTU / SCADA SYSTEMS

DATABASE OPTION

1.0 INTRODUCTION

The TEST INC. RTU/SCADA program supports the use of databases for storing numerical data over a period of time. These databases are created by the RTU program and can not be directly used by other software packages. Each database is basically just a table of rows and columns that is used to store data. The number of rows and columns that make up a database are determined at the time the database is created and can not be changed afterwards.

Each column in a database, referred to as a "field", represents a list of values for a single variable over time. When a database is created each column is assigned a default name. Those names can be changed at any time to reflect more meaningful representations of the data.

Each row in a database, referred to as a "record", represents the values for all fields at a particular time. Therefore, each record has a time and date associated with it and can be accessed by a logical index number or by a specific time and date.

NOTE: Systems which are used in different countries may need to include a line in the CONFIG.SYS file to control the date format. The following commands can be used in the CONFIG.SYS file to cause the dates to appear as follows:

```
COUNTRY = 001     <-- United States  mm/dd/yy  (Dos Default)
COUNTRY = 003     <-- Latin America  dd/mm/yy
```

Each task (task 0 or any task defined in the main configuration file during program startup) is able to open a database file, locate a record, retrieve data, update data, and close the database file completely independent of all other tasks. This means that different tasks can access different databases, or even the same database, at the same time. Each task is also capable of having more than one database file open at a time. However, only one database can be current for any task at any time.

Up to 5 database files can be open at the same time. These database files can all be opened by the same task, by 5 different tasks, or by any combination of tasks. Whenever a database file is opened, a flag is set for that database indicating the number of the task that opened the file. If another task tries to open the same file, the program will see that the file is already open and simply set another flag for that database indicating the number of that task too. By doing this, the program can keep track of all tasks that are currently using each database. When a task attempts to close a database file it first clears its flag for that database. This is all that is done if other tasks are still using that database. However, if no other tasks are using that database the database file will be closed. This method of keeping track of which tasks are using which databases will prevent any database file from physically being closed until all tasks are finished using the database.

Not only does each database keep track of which tasks are using it, but each task keeps track of which databases it is currently using. Each task also keeps track of which record it is currently positioned at within each database. Whenever a task opens a database for the first time (or simply selects it if it is already open), it will read in the first record and maintain a local copy of it. The response to any

request for data from that database by that task will come from the local copy of the current record. Also, any changes made to that database by that task will be performed on the same copy of that record. Only when that record is written back to the database file will the actual database be modified.

While each task can have more than one database open at a time and maintain a local copy of the current record from each database, the current positions and local copies are not affected when a task simply switches from one database to another. When a reference is made to a database field, it is always dealing with the local copy of the current record from the current database. Whenever a new record is selected for a database the program will automatically update the task's new position within the database and update the local copy of the current record. However, the program will not automatically write the local copy of the current record back to the database file before going to the new record. Therefore, it is up to the user to make sure that the local copy is always saved to the database file before locating a new record.

Most database functions are performed using the DB command. The available keywords that can be used with the DB command and a brief explanation of each are as follows:

DB CREATE - Create a new database file and set default values.
DB SELECT - Open a database file and make that database the current one.
DB LOCATE - Find a database record given a time & date or index number.
DB UPDATE - Save local copy of current record to current database file.
DB PURGE - Change database start date and erase all prior records.
DB CLOSE - Close the current database file for a task.
DB NEXT - Get the next logical record from the current database.

Other commands associated with databases include:

DUMP DB - Display information about open database files.
CONFIG DB - Edit current database.
CONFIG DB FIELDS - Edit current database field names.

A greater explanation of each of these commands is given in the sections that follow.

2.0 ENABLING DATABASES

The RTU program can operate in one of three modes (RTU, HOST, and DEMO mode). The mode is determined by a security lock (dongle) which is attached to the parallel port of the computer running the program. If no dongle is present when the program is started the system will operate in DEMO mode. Otherwise, the mode will be determined by the type of dongle being used (HOST or RTU). The **VER** command can be used at any time to determine the current mode of the program.

When operating in HOST mode all attempts to access a database are denied. In situations where a HOST dongle is being used and database access is necessary, the HOST dongle will have to be replaced with an RTU type dongle.

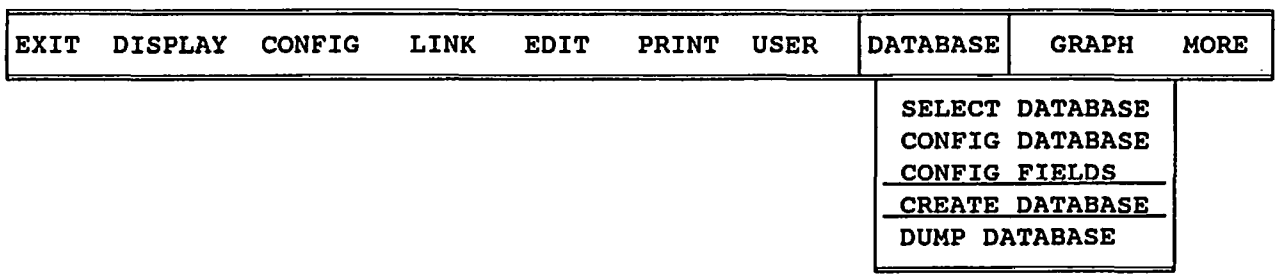
When operating in RTU mode or DEMO mode, databases can be used but access is limited by default to task 0 only. This default setup requires the least amount of memory for the program to run while still allowing databases to be used. To enable database processing for all tasks the command

```
LOAD DBASE
```

must get processed from within the main configuration (DAT) file during program startup. Unless databases are being used, this command should not be used in the main configuration file because it will force the program to allocate additional memory when it is started.

3.0 CREATING A DATABASE

A database can be created either by using an entry screen to enter all the necessary information or by using a single command which provides all the required information on one line. The entry screen can be accessed either by selecting the following items from the Main menu



or by using the DB CREATE command from the command prompt. Either way, the program will display the following entry screen with its default values:

```
                    CREATE DATABASE

Database File Name

Number of Fields        10

Start Date              (Current Date)

Updates Per Day         24

Number of Days          185


                <F2> Accept  •  <Esc> Cancel
```

From within the entry screen the **ESC** key can be pressed at any time to stop from creating a new database and return to either the Main menu or the command prompt. The **F2** key can be used to create a new database using the information provided. If no file type is specified in the database file name the extension **.SDB** will be used. If a file with the given database file name already exists at the time the **F2** key is pressed, a prompt will appear and the existing file will not be over-written unless the user acknowledges to do so.

Using several parameters the DB CREATE command can be used to create a database in a single command. The required parameters include the same parameters that are shown on the entry screen above. The format of the command is as follows:

```
DB CREATE FileName #ofFields StartDate UpdatesPerDay NumDays
```

If any of these parameters are missing or invalid the command will be ignored and no database will be created. Again, if no file type is specified in the database file name the extension .SDB will be used. **NOTE: If a file with the given database file name already exists at the time this command is processed, the user will not be prompted and the new database will be created by overwriting the existing file.**

The maximum **NUMBER OF FIELDS** allowed for any database is 512. When using the entry screen to create a database this field will not accept any number greater than 512. If a larger number is provided for this parameter in a command line used to create a database, the number of fields will simply be set to 512.

The **NUMBER OF DAYS** represents the time span for which data can be stored in a database. The default is 185 days which represents approximately 6 months. The **NUMBER OF DAYS** times the **UPDATES PER DAY** determines the number of records in the database. For example, if the default values are used to create a database (24 updates per day for 185 days) the database will contain 4440 records.

The first thing the program will do when creating a database is see if an existing database with the same name already exists and is currently being used by any task. If so, the existing database is immediately closed and future access is denied for any task that was using it.

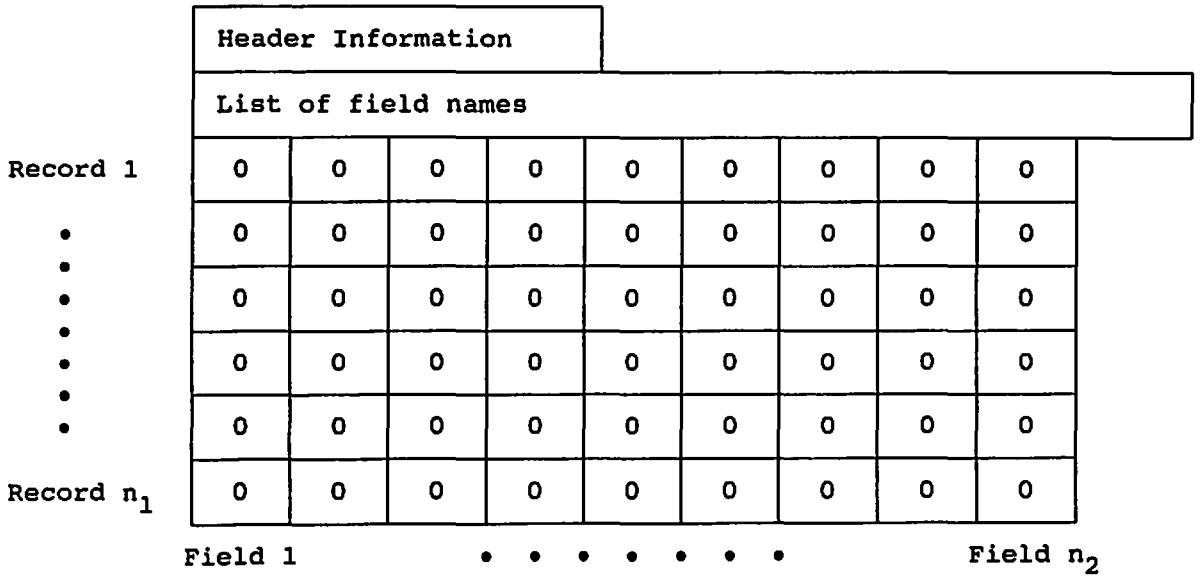
The program will then create the new database file by opening the file and writing a header at the beginning of the file. The header defines the properties of the database file. Whenever the database file is opened in the future the header is read to inform the program of the format and contents of the file. The information contained in the header includes:

Database version number
Number of records (# of days * updates per day)
Number of fields
First logical record
Date of first logical record

Date of last logical record
Number of days
Updates per day
Time interval between each record

Next, all fields of the database will be assigned default names of SDB_01, SDB_02, SDB_03, etc. and written out to the file. Finally, all database values will be initialized to 0 and written to the database file. Since creating a database involves physically writing out each record to disk it might take a few moments to create a database file that is extremely large.

Once a database file is created it will have the following format:



Although each record has a time and date associated with it, notice that there is no time and date stored in the database file for each record. The only dates that are stored in a database file are the **START DATE** that is provided when the database is created and the END DATE that is associated with the last record in the database. The END DATE is automatically calculated at the time the database is created and is stored with the START DATE in the file header.

The START DATE is the date associated with the first record in the database. The time associated with the first record is always midnight, which is displayed as 00:00:00 and indicates that no time has elapsed since the start of the day (00:00:00 am). The times and dates associated with all other records in a database are calculated as needed based on an offset from the first record.

The time and date associated with each consecutive record in a database is assumed to represent an incremental, fixed time interval from the previous record. The fixed time interval between each record is determined by dividing the number of minutes in a day (60 mins/hr * 24 hrs = 1440) by the number of times per day that the database is supposed to be updated (**UPDATES PER DAY** parameter). Once the time interval is known the time and date associated with any record can easily be determined. For example, suppose a database is created using the

following settings:

Number of Fields	6	
Start Date	01/01/90	<--- (U.S. format)
Updates Per Day	2	
Number of Days	30	

This database will contain 6 fields and 60 records (30 days * 2 updates per day) for a total of 360 values. The time interval between each record will be 12 hours which is calculated as follows:

$$\frac{1440 \text{ mins/day}}{2 \text{ updates/day}} = 720 \text{ mins/update}$$

$$\frac{720 \text{ mins/update}}{60 \text{ mins/hr}} = 12 \text{ hrs/update}$$

The format of the file and the associated time and date of each record would appear as follows:

		Header Information					
		List of field names					
01/01/90	00:00:00	0	0	0	0	0	0
01/01/90	12:00:00	0	0	0	0	0	0
01/02/90	00:00:00	0	0	0	0	0	0
01/02/90	12:00:00	0	0	0	0	0	0
01/03/90	00:00:00	0	0	0	0	0	0
	•	0	0	0	0	0	0
	•	0	0	0	0	0	0
	•	0	0	0	0	0	0
01/30/90	00:00:00	0	0	0	0	0	0
01/30/90	12:00:00	0	0	0	0	0	0

The amount of disk space that a database file requires can be determined by adding up the amount of space required by each of the components that make up the file. As explained earlier each database file consists of a header, a list of field names, and a table of values. The amount of disk space required by each of these components is shown below.

Header	32 bytes
List of Field Names	18 bytes per field
Actual Values	6 bytes per value

To calculate the amount of disk space required to create a database consider the following example. This example uses the program's default values which are:

Number of fields	10	
Updates per day	24	<--- once per hour
Number of days	185	<--- 6 months

The total amount of disk space required would be 216,612 bytes. This is calculated as follows:

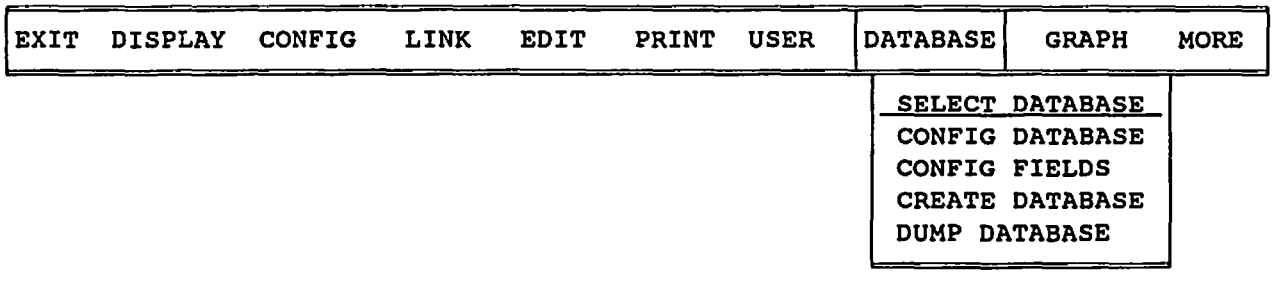
Header	32	
List of Field Names	180	<--- 10 fields x 18 bytes per field name
Actual Values	266,400	<--- (10 x 24 x 185) x 6 bytes per value
Total	266,612 bytes	

4.0 SELECTING A DATABASE

Each task (task 0 or any task defined in the main configuration (DAT) file during program startup) is able to open a database file, locate a record, retrieve data, update data, and close the database file completely independent of all other tasks. This means that different tasks can access different databases, or even the same database, at the same time. Each task is also capable of having more than one database file open at a time. However, only one database can be current for any task at any time.

Whenever a task performs a database operation it affects only the current database. To make a database current a task must select a database. When a database is selected the database file will automatically be opened if it is not already open. Otherwise, a flag is simply set adding the task number to the list of other tasks that are currently using that database file.

A database can be selected from the Main menu as shown below



or by using the DB SELECT command.

Choosing the SELECT DATABASE option from the Main menu will cause a

pick list to appear containing all the files with the extension .SDB. These are all the standard database files that have been created as described above. The ESC key can be used to exit the pick list and allow a file name to be entered that does not have the standard .SDB extension.

The format of the command line that can be used to select a database is

```
DB SELECT filename
```

where *filename* is the name of the database file. If no file type is specified the extension .SDB will be used. A drive and directory can also be specified in the file name.

When a database is selected the program will look to see if the specified database file is already open and being used, either by the task that processed the command or by any other task. There are 4 possible results of this function which are explained below.

1 - If the specified file does not exist the result is an error and the current database for that task remains unchanged.

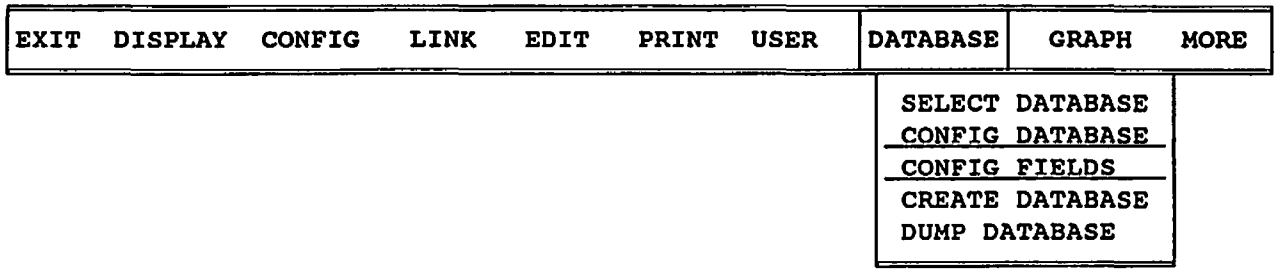
2 - If the specified file exists but is not yet being used by any task, the program will open the file, set a flag for that database indicating which task is using it, make that database the current one for that task, make the first record the current one, and read in a local copy of that record.

3 - If the specified file exists and is already being used by another task (but not the current task), the program will set a flag for that database indicating that the current task is now using it too, make that database the current one for that task, make the first record the current one, and read in a local copy of that record.

4 - If the specified file exists and is already being used by the current task, the program will simply make that database the current one for that task. The current record for that task within that database will not be changed.

5.0 EDITING DATABASE FIELD NAMES

At the time a database is created all fields are assigned the default names of SDB_01, SDB_02, SDB_03, etc. These field names can be changed as often as desired using an entry screen. This entry screen can be reached from the Main menu as shown below



or by using the CONFIG DB FIELDS command from the command prompt.

In order for the entry screen to be displayed there must be a current database for task 0. If no database is current at the time the menu selection is made a pick list will be displayed and a database will have to be selected. If no database is current at the time the CONFIG DB FIELDS command is processed the command will not be successful. The DB SELECT command can be used before the CONFIG DB FIELDS command to select the proper database before trying to edit the field names.

As an example, the entry screen used to edit the field names for a newly created database called XXX.SDB which contains 10 fields would appear as follows:

XXX.SDB FIELD NAMES

```
Field 1     SDB_01
Field 2     SDB_02
Field 3     SDB_03
Field 4     SDB_04
Field 5     SDB_05
Field 6     SDB_06
Field 7     SDB_07
Field 8     SDB_08
Field 9     SDB_09
Field 10    SDB_10
```

<F2> Accept • <Esc> Cancel

The default values SDB_01, SDB_01, and so on, would be changed to more meaningful names such as WELL1, WELL2, etc. When editing is complete the database will remain current for task 0 and the database file will remain open until a DB CLOSE command is processed.

Each database field name can contain up to 17 characters. If all the information in a database pertains to a single RTU the field names can be simple tag names. However, RTU names should be included in the field names if the information in a database includes data from several RTUs. When specifying an RTU name in a database field name the standard dot notation should be used where the RTU name is followed by a period and then the tag name (RTU.TAG). This will eliminate any confusion as to which fields in the database belong to which RTU.

For example, if a database is used to track the gas flow and oil production for a single RTU the database could be setup to include two fields with the following names:

```
GAS_FLOW     OIL_PROD
```

However, if a single database is used to track the gas flow and oil production for two RTUs, the field names should probably include the RTU names. Assuming that the two RTUs are called RTU1 and RTU2 the database could be setup to include four fields with the following names:

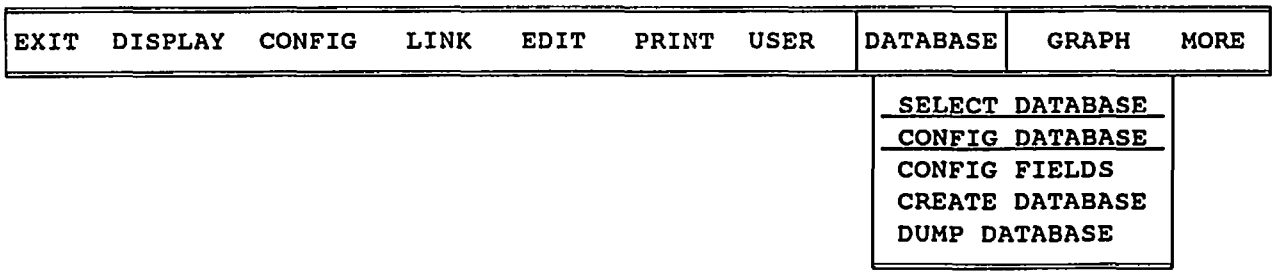
```
RTU1.GAS_FLOW     RTU1.OIL_PROD     RTU2.GAS_FLOW     RTU2.OIL_PROD
```

Although it is good practice, it is not absolutely necessary to specify an RTU name in the database field names when a database contains data for more than one RTU. For example, the database field names listed above could be setup as simple tag names which uniquely identify each field as shown below:

```
GAS_FLOW1     OIL_PROD1     GAS_FLOW2     OIL_PROD2
```

6.0 EDITING A DATABASE

The values of a database can be edited using an entry screen which can be accessed from the Main menu as shown below



or by using the CONFIG DB command from the command prompt.

The process of editing a database is exactly the same as the process used to edit the database field names as explained above. That is, in order for the entry screen to be displayed there must be a current database for task 0. Therefore, always be sure to select a database, either with the DB SELECT command or from the Main menu, before using the CONFIG DB command to edit a database.

The entry screen used to edit a database contains a title, a list of all database field names, and the values for the current record. The title contains the database file name, the time and date associated with the current record, the logical record number, and the total number of records in the database. The **PgUp** and **PgDn** keys can be used to move among the different records in the database. As the current record changes, the time, date, and current record number in the title are automatically updated. Any changes made to a record are immediately saved to the database file when the **PgUp** or **PgDn** key is used to go to another record. The **ESC** key can be used to exit the entry screen and cancel any changes. However, only changes to the current record will be cancelled. Changes to any other records would have already been saved. The **F2** key can be used to exit the entry screen and save any changes to the current record as well. When editing is complete the database will remain current for task 0 and the database file will remain open until a DB CLOSE command is processed.

To illustrate the database entry screens the following example is given:

```
XXX.SDB                    01/01/90  00:00:00                    1/4440

WELL1          0
WELL2          0
WELL3          0
SDB_04         0
SDB_05         0
SDB_06         0
SDB_07         0
SDB_08         0
SDB_09         0
SDB_10         0

<F2> Save • <PgUp> Previous Record • <PgDn> Next Record • <Esc> Cancel
```

This display shows the first record of database XXX.SDB which has 4440 records total, whose start date is 01/01/90, and whose first 3 field names have been changed to WELL1, WELL2, and WELL3. The remaining fields are apparantly not being used to store data at this time.

When editing a database the entry screen will always display the record that is current at the time the edit begins. The first record of a database is always the current record when the database is first opened (or first selected by a task). Therefore, if editing began immediately after opening a database the entry screen would display the contents of the first record in the database. To get to a record that is further down in the database, say for example record 200, the PgDn key would have to be pressed 200 times. Obviously, this method is very inconvenient.

As an alternative to paging down to the desired record, the DB LOCATE command could first be used to find the desired record and make it current. The database entry screen could then be called up and the editing would begin with the desired record. For example, suppose we wish to edit the 200th record in a database called XXX.SDB. The following commands could be entered from the command prompt to open the database and start editing at that record:

```
DB SELECT X
DB LOCATE 200
CONFIG DB
DB CLOSE
```

Notice that the DB CLOSE command is added to the end of these commands. This command is not necessary but just included as a reminder that no database is automatically closed after editing.

Specific records in a database can be located either by index number or the unique time and date associated with each record. Refer to the DB LOCATE command for more information on locating specific records.

7.0 LOCATING A DATABASE RECORD

There are 2 commands that can be used to locate a particular database record. These two commands are

```
DB NEXT
```
and
```
DB LOCATE
```

The DB NEXT command will cause a task's current record to be incremented by 1. When used in a loop in a command file this command provides an easy way to assign data to, retreive data from, or simply examine each record in a database. For example, suppose you wanted to scan a database for a value greater than 5000 in a particular field. If the name of the database is MAIN_DB and the name of the field is TOTAL the following command file could be used.

```
DB SELECT MAIN_DB       ; open database, start at first record
:LOOP
  IF TOTAL > 5000       ; found the record
    RETURN              ; return with current record
  ELSE                  ; not found yet
    DB NEXT             ; go to next record
  ENDIF
  IF @DBERR(0) > 0      ; reached end of database
    DB CLOSE            ; close database
    RETURN              ; return with no current record or database
  ENDIF
GOTO LOOP               ; continue looking
```

File processing will terminate when the first record is found that contains a value greater than 5000 in the TOTAL field. If such a record exists the database will remain open and that record will become the current record. However, if such a record does not exist the scan will continue until the end of the file is reached. At that time the @DBERR function will detect the error and the current database will be closed. (See the section called DATABASE ERRORS later in this manual for details about the @DBERR() function).

When the current record is the last record in the database and the DB NEXT command is processed an error will result. The current database will remain open but the current record will become invalid (set to -1). The only way to obtain a valid current record again is to either close the database and re-open it or use the DB LOCATE command.

The DB LOCATE command can be used to locate a particular record either by specifying a record number or a time and date. The format of the command is

```
DB LOCATE index_no    or     DB LOCATE time date
```

Examples:

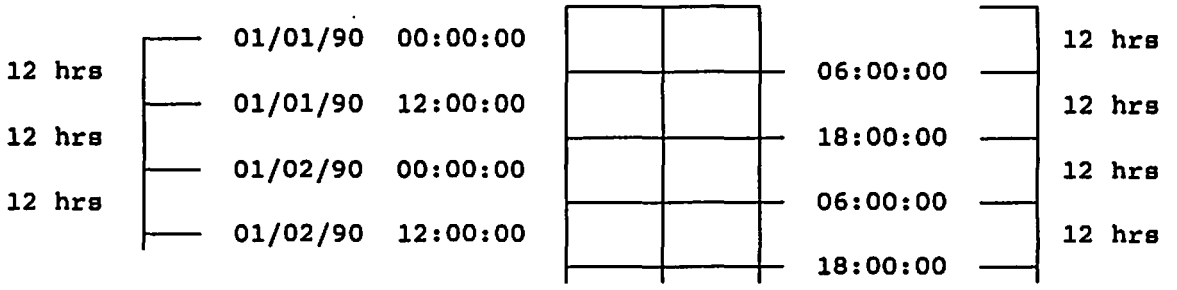
```
DB LOCATE 1                     ; 1st record in database
DB LOCATE 52                    ; 52nd record in database
DB LOCATE 04:00:00 05/06/90     ; specific time and date
DB LOCATE $T $D                 ; current time and date
```

An error will result if the record index number, time, or date specified is invalid. An error will also result if an attempt is made to locate a record which preceeds the beginning of the database or a date is specified which exceeds the current date. When an error occurs the current record is set to -1. Refer to the @DBERR() function for a list of error codes.

A special situation arises when an attempt is made to locate a record beyond the scope of the database. When this happens an error does not occur but some of the information in the database is deleted to make room for the new record. Therefore, always use special care with the DB LOCATE command to make sure that important information is not accidentally deleted. Refer to the following section for more information about this subject.

Another point worth mentioning about the DB LOCATE command is how the program locates a record using a time and date that does not exactly match the time and date associated with a record. What the program does is round the

specified time up or down to locate the record with the time and date that is closest to the specified time and date. For example, consider the portion of the database shown below with the associated time and date of each record shown on the left:



Notice that there is a 12 hour time interval between each record which is shown on the far left. However, each record does not contain data for the 12 hours represented by the interval between each record. For example, record 1 does not represent data from 00:00:00 unitl 12:00:00 on 01/01/90 and record 2 does not represent data from 12:00:00 of 01/01/90 until 00:00:00 of 01/02/90.

Although each record does represent data for a 12 hour time interval, the time interval is centered around the time associated with each record. The actual time interval associated with each record is shown on the right. For example, record 2 represents data from 06:00:00 until 18:00:00 on 01/01/90 and record 3 represents data from 18:00:00 on 01/01/90 until 06:00:00 on 01/02/90.

The reason for this shift in time intervals is due to the rounding that takes place. For example, using the portion of the database shown above suppose the command DB LOCATE 20:00:00 01/01/90 is processed. The third record will become the current record instead of the second record. At first this might seem awkward because an attempt to locate a record using a date of 01/01/90 actually returns a record with a date of 01/02/90. However, the rounding allows the DB LOCATE command to find the record with the time and date that most closely matches to the specified time and date.

8.0 AUTOMATIC SCROLL FEATURE

When the DB LOCATE command is used to locate a record beyond the scope of a database the oldest records in the database are automatically deleted to make room for the new records. The minimum number of records that can be deleted at one time is equivalent to the number of updates per day the database was created to handle. The reason for this is that the first record in the database must always start at midnight. Therefore, we must always delete at least one day's worth of data rather than just one record's worth of data. If the number of updates per day is greater than 1 then the number of records used to store a day's worth of data will be greater than 1.

This automatic scroll feature allows systems to be set up to record data indefinitely without increasing the size of the database or running out of room. For example, if a database is set up to store 6 months worth of data it can be updated

regularly and always maintain data for the most recent 6 months. Consider the following file used to update a database on a regular basis:

```
DB SELECT xxx          ; open database file
DB LOCATE $T $D        ; get record for current time and date
CALC field = xxx       ; update local copy of record
DB UPDATE              ; save local copy of record to file
DB CLOSE               ; close database file
```

If this file is processed regularly the database will eventually become completely filled with data. When the day comes to locate a record that exceeds the limits of the database by one day, the program will delete the records with the oldest date and make room for the new day's worth of data. On the next day and everyday following that the same thing will happen again. Therefore, the database can always maintain data for the most recent 6 months (or whatever the database is setup to handle). For example, consider the following database which was created to handle 2 updates per day for 30 days. The database is shown with x's to indicate that it is filled with data.

		Header Information					
		List of field names					
01/01/90	00:00:00	x	x	x	x	x	x
01/01/90	12:00:00	x	x	x	x	x	x
01/02/90	00:00:00	x	x	x	x	x	x
01/02/90	12:00:00	x	x	x	x	x	x
01/03/90	00:00:00	x	x	x	x	x	x
	•	x	x	x	x	x	x
	•	x	x	x	x	x	x
	•	x	x	x	x	x	x
01/30/90	00:00:00	x	x	x	x	x	x
01/30/90	12:00:00	x	x	x	x	x	x

If a DB LOCATE 04:00:00 01/31/90 command is processed the first day's worth of data will be deleted (2 records), the database will contain room for the new day's worth of data, and the database will appear as follows:

		Header Information					
		List of field names					
01/02/90	00:00:00	x	x	x	x	x	x
01/02/90	12:00:00	x	x	x	x	x	x
01/03/90	00:00:00	x	x	x	x	x	x
01/03/90	12:00:00	x	x	x	x	x	x
01/04/90	00:00:00	x	x	x	x	x	x
	•	x	x	x	x	x	x
	•	x	x	x	x	x	x
	•	x	x	x	x	x	x
01/31/90	00:00:00	0	0	0	0	0	0
01/31/90	12:00:00	0	0	0	0	0	0

To the user it will appear as if all records have physically moved up in the database creating room for more data at the end of the database. However, to keep things efficient the program does not physically move records around. The way the automatic scroll works is identical to the purge feature built into the program. In fact, an automatic scroll is actually performed by using the date provided in the DB LOCATE command to process a DB PURGE command. Refer to the section called PURGING A DATABASE for details about the internal processes involved with scrolling and purging.

9.0 REFERENCING DATABASE FIELDS

When a command line is processed the program will attempt to identify and evaluate any tag names on the line by first looking for a match with a channel name, then a local or public variable, and finally a database field name. Since it is possible (and often desirable) for a database field to have the same name as a channel or variable, there must be a way to allow the expression evaluator to differentiate between the two. This distinction can be made by adding the # character to the end of the field name when referencing a database field (Tag#).

For example, suppose a channel used to keep track of total oil production

has the tag name TOT_PROD. Also suppose that a database exists which contains a field called TOT_PROD that is used to track total oil production over a period of time. Finally, assume that the total oil production is determined from the sum of two channels, WELL1 and WELL2. To update the value of the TOT_PROD channel the following command could be used:

```
CALC TOT_PROD = WELL1 + WELL2
```

To update the value of the TOT_PROD database field for the current record the following command could be used:

```
CALC TOT_PROD# = WELL1 + WELL2
```

When the expression evaluator finds the # character in a tag name it automatically strips off the # character and everything following it. The remaining tag is then assumed to represent a database field name. The expression evaluator will skip the search for a channel or variable name and go directly to the current database to search for a field name that matches the tag. If no match is found the evaluation will fail and no attempt will be made to search the fields of any other open databases.

Using the # character not only distinguishes between a channel or variable name and a database field name, but it also increases command processing time. If a database contains a field with a name that is different from any channel or variable name the # character does not need to be used when referencing the database field. After failing to find a matching channel and variable name, the evaluator will search the current database and find the matching field name. However, if the # character is used when referencing the database field the evaluator will skip the search for a matching channel or variable name and go directly to the database. This will speed up the time it takes to process the command because the evaulator will not spend time looking for a matching tag in places it will not be found.

In some situations it may be desirable to have access to the information in a database other than the current database. This is possible for any task as long as the database is a least open for use by the task. For example, consider the situation where the information in one database must be transferred to another database. To begin, a task would open the two databases. Although the two databases can be open at the same time they can not both be current at the same time. Therefore, the task would have to get the information from the current database and transfer it to the other database (or vice versa).

To access a field in a database that is open for a task but not current the # character would have to be added to the end of the field name followed by the name of the database (Tag#Database). As mentioned earlier, if the expression evaluator finds the # character in a tag name when processing a command line it will automatically strip off the # character and everything following it. However, anything found after the # character is assumed to be a database file name. That name tells the evaluator which database file to search for the matching field name. To be successful the database must be open but does not have to be current.

For example, consider the situation in which the information in one database must be transferred to another database. Assume that the two database are called

DB1.SDB and DB2.SDB. Also assume that the data is to be transferred from a field called TOTAL in DB1 to a field called TOTAL in DB2. The following file could be used to transfer contents of the entire field:

```
DB SEL DB1                    ; open first database
DB SEL DB2                    ; this database is now current
:LOOP
  CALC TOTAL = TOTAL#DB1      ; transfer from DB1 to DB2
  DB UPDATE                   ; save record to DB2

  DB SEL DB1                  ; make DB1 current
  DB NEXT                     ; get next record from DB1
  IF @DBERR(0) > 0            ; if went beyond end of database
    DB CLOSE                  ; close DB1
    DB SEL DB2                ; make DB2 current
    DB CLOSE                  ; close DB2
    RETURN                    ; stop processing this file
  ENDIF

  DB SEL DB2                  ; make DB2 current again
  DB NEXT                     ; get next record from DB2
  IF @DBERR(0) > 0            ; if went beyond end of database
    DB CLOSE                  ; close DB2
    DB SEL DB1                ; make DB1 current
    DB CLOSE                  ; close Db1
    RETURN                    ; stop processing this file
  ENDIF

GOTO LOOP                     ; continue looping
```

One last note regarding the search for a database field name should be made. When the evaluator is searching for a database field name the tag being searched for must exactly match the database field name. This means that if an RTU is specified in the database field name using the dot notation described above, then the same RTU name must be specified anytime the database field is referenced. The expression evaluator does not assume the current RTU name preceeds a tag name if one is not specified.

10.0 SAVING A DATABASE RECORD

When editing a database the current record is automatically saved whenever the **PgUp**, **PgDn**, or **F2** key is pressed. The **PgUp** and **PgDn** keys are used to flip from one record to another. The **F2** key is used to save the current record and exit the entry screen. These are the only times that a database record is automatically saved. Any changes made to a database record without using the database entry screen can only be saved using the DB UPDATE command.

Whenever a task opens a database for the first time (or simply selects it if it is already open by another task) it reads in the first record and maintains a local copy of it. Any changes to that record will be made to the local copy. The actual database will not be changed until the local copy is written back to the database

file. If another record becomes current (DB NEXT or DB LOCATE commands) or the database file is closed (DB CLOSE command) before the local copy is saved, the changes to that record will be lost.

The DB UPDATE command is used to save the local copy of the current record to the database file. The normal sequence of commands used to update a database is as follows:

```
DB SELECT xxx           ; open database file
DB LOCATE xxx           ; get local copy of desired record
CALC field = xxxxx      ; update local copy
DB UPDATE               ; save local copy of record to database
DB CLOSE                ; close database file
```

If the DB UPDATE command is not processed the changes will not be saved.

11.0 CLOSING A DATABASE FILE

The DB CLOSE command is used to close then current database for a task. When a task processes this commnad it will no longer have a current database, even if other databases for that task are still currently open. The program does not automatically make another open database current when one is closed. The only way to make another database current is to select it using the DB SELECT command. To close all databases used by a task each database would have to be individually selected and closed.

When an attempt is made to close a database file a check is done to see what tasks are currently using the database. If the only task using the database is the one processing the DB CLOSE command the database file will physically be closed. Otherwise, the file will remain open and only a flag will be cleared indicating that the task processing the DB CLOSE command is no longer using the database.

Before using the DB CLOSE command the user should be sure that all changes to the current database have been saved. This command does not automatically save any information before closing a file.

12.0 PURGING A DATABASE

The DB PURGE command can be used to clear out old data from a database. The format of the command is

```
DB PURGE date
```

where `date` is replaced by a valid date in the form of mm/dd/yy (or possibly dd/mm/yy). The command will not work if the specified date preceeds the database start date or exceeds the current date. When the command is processed the program will locate the record associated with midnight of the date specified. That record will become the first and current record and all prior records in the

database will be deleted.

To the user it will appear as if all remaining records have been physically shifted up to the beginning of the database. However, to keep things simple and efficient the program does not physically move the data for any records. Instead, the program simply clears out the old records and keeps track of the logical position of the new first record in the database. All other records can then be located by adding the logical record number to the physical offset of the first record in the database. The deleted records will logically become the last records in the database, although they may physically be the first records. The program is smart enough to know that when the physical end of the database is reached there may still be more records available that can be accessed by looping around to the beginning of the database.

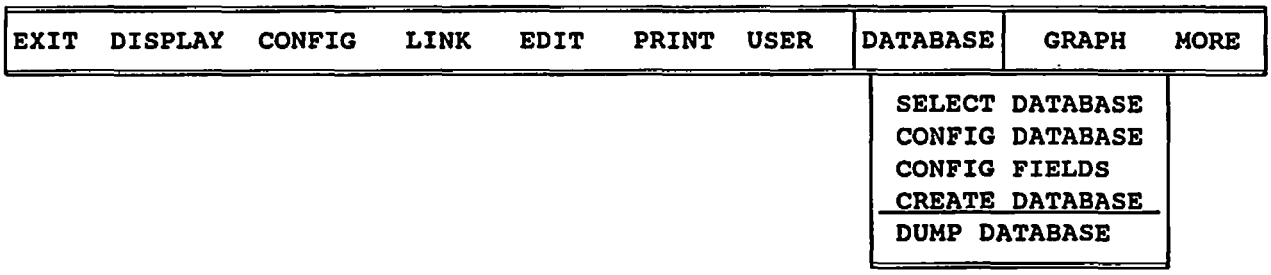
For example, if a database is created that contains 12 records, the records will physically and logically be located at positions 1-12 in the database. If a purge is done and the first 4 records are deleted the layout would look like this:

	Physical Records	Logical Records	
data is deleted --->	1	9	
data is deleted --->	2	10	
data is deleted --->	3	11	
data is deleted --->	4	12	<--- new last record
data unchanged --->	5	1	<--- new first record
data unchanged --->	6	2	
data unchanged --->	7	3	
data unchanged --->	8	4	
data unchanged --->	9	5	
data unchanged --->	10	6	
data unchanged --->	11	7	
data unchanged --->	12	8	

This explanation of what actually happens during a purge is provided only as a reference to how the database system really works. From the user's point of view it will appear as though all remaining records are physically shifted to the beginning of the database during a purge. No extra work is ever required by the user to keep track of the logical or physical location of the records.

13.0 DISPLAYING DATABASE STATUS

The status of all open databases can be displayed from the Main menu as shown below



or by using the DUMP DB or DUMP DBASE command from the command prompt. First, all currently open databases are listed along with the number of each task using the databases. Second, each task that has a current database is listed with additional information about the status of the database. Such information includes:

```
Task #
Current Database Name
Current Record Number
Error Code
Start Date
Number of Fields
Updates per Day
Number of Days
Number of Records
```

14.0 DATABASE ERRORS

Each task keeps track of the current status of each database it is using. A list of database errors that can occur and their related codes are as follows:

1 - Tried to process a DB NEXT command to go beyond last record
2 - Invalid time or date specified in DB LOCATE command
3 - Date specified in DB LOCATE command preceeds database start date
4 - Date specified in DB LOCATE command exceeds current date
5 - Index specified in DB LOCATE command is less than 1
6 - Index specified in DB LOCATE command is too big, forces search to go beyond current date

When a task encounters a database related error, the error code will get set and the current record for that database will no longer be valid (set to -1). The current status of a database can be checked from within a command file by using the @DBERR() function. If no error exists the error code will be 0, otherwise it will be one of the codes listed above.

For example, suppose a database called OIL_PROD.SDB contains two fields, called WELL1 and WELL2, which have been tracking the production of two oil platforms for the last 6 months. If for some reason the user would like to store the total of these two fields in the database, a command file could be created using the @DBERR() function to help sum up the existing values. First of all, the database field names could be edited and a field not currently being used could be given the name TOTAL. The following command file could then be processed to sum up

fields WELL1 and WELL2 and store the result in TOTAL.

```
DB SELECT OIL_PROD                    ; open OIL_PROD database
:LOOP
  CALC TOTAL = WELL1 + WELL2          ; sum two fields into TOTAL
  DB UPDATE                           ; save the modified record
  DB NEXT                             ; get next record
  IF @DBERR(0) > 0                    ; check for database error
    DB CLOSE                          ; close OIL_PROD database
    RETURN                            ; stop processing this file
  ENDIF
GOTO LOOP                             ; continue looping
```

Although this example may not be practical, it illustrates how the @DBERR(0) function can be used to continuously do something with a database as long as no error exists. In this example, the DB NEXT command will eventually try to access a record beyond the last record in the database. When this occurs the @DBERR() function will return a value of 1. This in turn will cause the DB CLOSE command to be executed from the command file followed by the RETURN command.

NOTE: The parameter used by the @DBERR() function is insignificant. That is, the parameter can be any numerical value. This example uses a value of 0 but the function will work just the same using any value at all.