iRMX™86 UNIVERSAL DEVELOPMENT INTERFACE REFERENCE MANUAL

CONTENTS

CONTENTS (continued)

TABLES

FIGURES

***

CHAPTER1
INTRODUCTION TO THE UNIVERSAL DEVELOPMENT INTERFACE

Intel's Universal Development Interface (known in this manual as the UDI) is a set of system calls that is compatible with each of Intel's operating systems. If an application system makes UDI system calls but no explicit calls to the resident Intel operating system, the application can be transported between operating systems. Figure 1-1 illustrates the relationship between application code, the processing hardware, and the layers of software that lie in between.

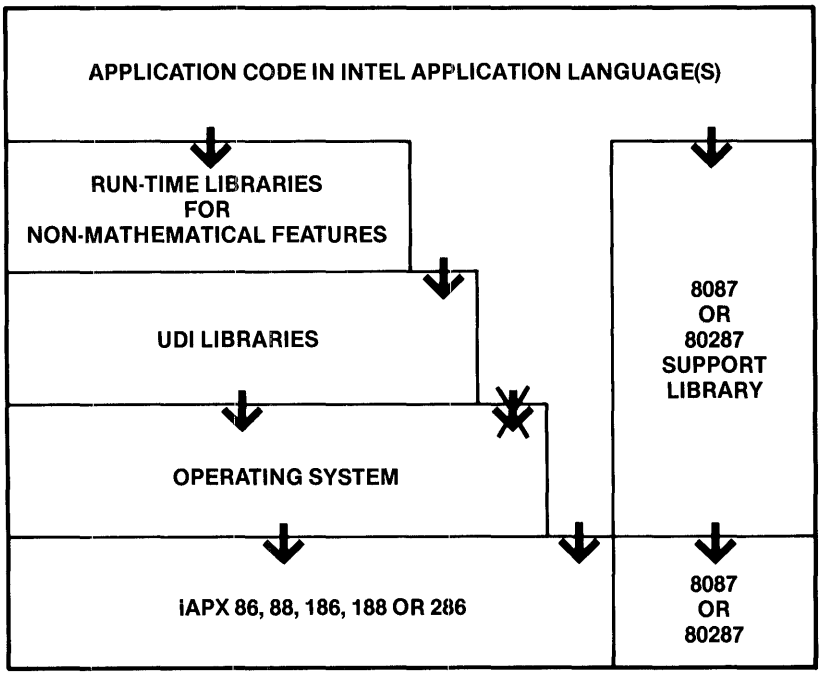

Figure 1-1. The Application-Software-Hardware Model

In Figure 1-1, the downward arrows represent command flow and data flow from the application code down to the hardware, where the commands are ultimately executed. (Not shown in the figure is another set of arrows showing the upward flow of data from the hardware to the application code.) Note that one of the downward arrows is crossed out, signifying that the application code does not make direct calls to the operating system. Rather, all interaction between the application code and the operating system is done through the UDI software.

By letting the UDI serve as the link between an application and the operating system, it is possible to switch operating systems simply by changing the interface between the UDI and the operating system. In other words, all that is necessary to make an application transportable between operating system environments is a UDI library for each operating system. This library always presents the same interface to the application, but its interface with the operating system is designed specifically and exclusively for that operating system. Intel provides UDI libraries for the iRMX 86, iRMX 88, Series III, and Series IV operating systems.

The UDI system calls, while presenting a standard interface to user programs, behave somewhat differently when used in different operating system environments. The reason for this is that the operating systems each have many unique characteristics, and some of them are reflected in the results of the UDI calls. For information about the UDI and the minor behavioral differences it exhibits between operating systems, refer to the RUN-TIME SUPPORT MANUAL FOR iAPX 86,88 APPLICATIONS.

The next chapter discusses the UDI in the context of the iRMX 86 Operating System.

***

CHAPTER 2
UDI SYSTEM CALLS IN THE iRMX™ 86 ENVIRONMENT

The purpose of this chapter is to describe the requirements and behavior of UDI system calls in the iRMX 86 environment.

SYSTEM CALL DICTIONARY

This section presents, in Table 2-1, a list of the UDI calls, arranged by functional category. Each entry in the list includes the name of the call, a concise description of its purpose, and its page number in this chapter.

Table 2-1. System Call Dictionary

SYSTEM CALL	FUNCTION PERFORMED	PAGE
	PROGRAM-CONTROL CALLS	
DQ$EXIT	Exits from the current application job.	2-20
DQ$OVERLAY	Causes the specified overlay to be loaded.	2-37
DQ$TRAP$CC	Assigns Control-C procedure.	2-51
	MEMORY-MANAGEMENT CALLS	
DQ$ALLOCATE	Requests a memory segment of a specified size.	2-8
DQ$FREE	Returns a memory segment to the system.	2-25
DQGETSIZE	Returns the size of a memory segment.	2-31
DQ$RESERVE$-IO$MEMORY	Requests memory to be set aside for overhead to be incurred by I/O operations.	2-42

Table 2-1. System Call Dictionary (continued)

SYSTEM CALL	FUNCTION PERFORMED	PAGE
	FILE-HANDLING CALLS	
DQ$ATTACH	Creates a connection to a file.	2-9
DQ$CHANGE$-ACCESS	Changes the access rights associated with a file or directory.	2-10
DQ$CHANGE$-EXTENSION	Changes the extension of a file name.	2-12
DQ$CLOSE	Closes a file connection.	2-13
DQ$CREATE	Creates a file.	2-14
DQ$DELETE	Deletes a file.	2-18
DQ$DETACH	Closes a file and deletes a connection to it.	2-19
DQ$FILE$INFO	Returns data about a file connection.	2-22
DQGETCON-NECTION$STATUS	Returns the status of a file.	2-28
DQ$OPEN	Opens a file connection.	2-34
DQ$READ	Reads the next sequence of bytes from a file.	2-39
DQ$RENAME	Renames a file.	2-41
DQ$SEEK	Moves the current position pointer of a file.	2-44
DQ$SPECIAL	Sets the line-edit mode for a terminal.	2-46
DQ$TRUNCATE	Truncates a file to a specified length.	2-53
DQ$WRITE	Writes a sequence of bytes to a file.	2-54

Table 2-1. System Call Dictionary (continued)

SYSTEM CALL	FUNCTION PERFORMED	PAGE
EXCEPTION-HANDLING CALLS		
DQ$DECODE$-EXCEPTION	Converts an numeric exception code into its equivalent mnemonic.	2-15
DQGETEXCEPT-ION$HANDLER	Returns a POINTER to the current exception handler.	2-30
DQ$TRAP$-EXCEPTION	Identifies a custom exception handler to replace the current handler.	2-52
UTILITY CALLS		
DQ$DECODE$TIME	Returns system time and date in both binary and ASCII-character format.	2-16
DQGETARGUMENT	Returns an argument from a command line.	2-26
DQGET-SYSTEM$ID	Returns the name of the underlying operating system supporting the UDI.	2-37
DQGETTIME	(Obsolete: included for compatability.)	2-33
DQ$SWITCH$BUFFER	Selects a new buffer to contain command lines.	2-49

OVERVIEW

This section discusses the functions of the many of the system calls, highlighting the interrelationships, if any, among the calls in the functional groups of Table 2-1.

MEMORY MANAGEMENT SYSTEM CALLS

When the iRMX 86 Operating System loads and runs a program, the program is allocated memory, in an amount that depends upon how the program was configured. The portion of memory not occupied by loaded code and data -- the free space pool -- is available to the program dynamically, that is, while the program runs. The Operating System manages memory as segments that programs can obtain, use, and return.

Programs can use the UDI system calls named DQ$ALLOCATE and DQ$FREE to get memory segments from the pool, and to return segments to the pool, respectively. They can also call DQGETSIZE to receive information about allocated memory segments.

FILE-HANDLING SYSTEM CALLS

About one-half of UDI system calls are used to manipulate files. Figure 2-1 shows the chronological relationships among the most frequently used file-handling system calls.

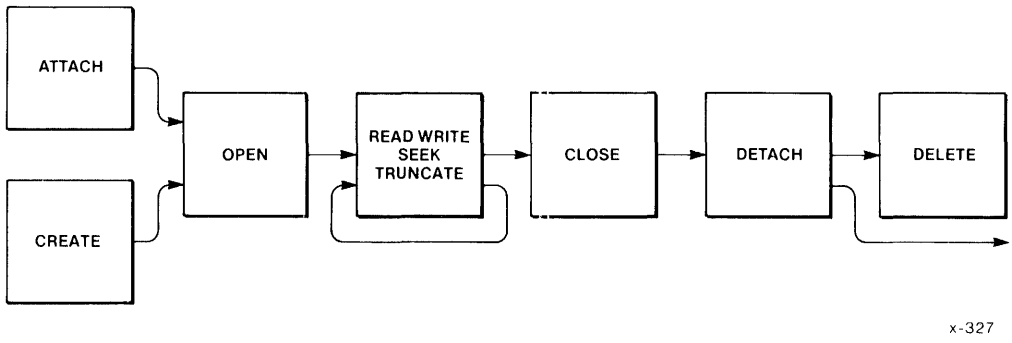

Figure 2-1. Chronology Of System Calls

The key to using iRMX 86 files is the connection. A program wanting to use a file first obtains (a token for) a connection to the file and then uses the connection to perform operations on the file. Other programs can simultaneously have their own connections to the same file. Each program having a connection to a file uses its connection as if it has exclusive access to the file.

A program obtains a connection by calling DQ$ATTACH (if the file already exists) or DQ$CREATE (to create a new file). When the program no longer needs the connection, it can call DQ$DETACH to delete the connection. To delete both the connection and the file, the program calls DQ$DELETE.

Once a program has a connection, it can call DQ$OPEN to prepare the connection for input/output operations. The program performs input or output operations by calling DQ$READ and DQ$WRITE. It can move the file pointer associated with the connection by calling DQ$SEEK. When the program has finished doing input and output to the file, it can close the connection by calling DQ$CLOSE. Note that the program opens and closes the connection, not the file. Unless the program deletes the connection, it can continue to open and close the connection as necessary.

If a program calls DQ$DELETE to delete a file, the file cannot be deleted while other connections to the file exist. In that case, the file is marked for deletion and is not actually deleted until the last of the connections is deleted. During the time that a file is marked for deletion, no new connections to it may be created.

CONDITION CODES AND EXCEPTION HANDLING CALLS

Every UDI call except DQ$EXIT returns a numeric condition code specifying the result of the call. Each condition code has a unique mnemonic name by which it is known. For example, the code 0, indicating that there were no errors or unusual conditions, has the name E$OK. Any other condition means there was a problem, so these conditions are called exceptions.

Exception conditions are classified as:

- Environmental Conditions. These are generally caused by conditions outside the control of a program; for example, device errors or insufficient memory.

- Programmer Errors. These are typically caused by mistakes in programming (for example, "bad parameter"), but "divide-by-zero", "overflow", "range check", and errors detected by the 8087 80287 Numeric Processor Extension (hereafter referred to generically as the NPX) are also classified as programmer errors.

The iRMX 86 NUCLEUS REFERENCE MANUAL contains a list of condition codes that the iRMX 86 Operating System can return, with the mnemonic and meaning of each code.

When an exception condition is detected, the normal (default) system action is to display an error message at the console and terminate the program. However, your program can establish its own exception handler by calling DQ$TRAP$EXCEPTION. The exception handler can interpret condition codes that are returned by calling DQ$DECODE$EXCEPTION. The rest of this section provides some facts that you need in order to write your own exception handler.

After an exception condition occurs and before your exception handler gets control, the iRMX 86 Operating System does the following:

1. Pushes the condition code onto the stack of the program that made the system call having the exception condition.

2. Pushes the number of the parameter that caused the exception onto the stack (1 for the first parameter, 2 for the second, etc.).

3. Pushes a word onto the stack (reserved for future use).

4. Pushes a word for the NPX onto the stack.

5. Initiates a long call to the exception handler.

If the condition was not caused by an erroneous parameter, the responsible parameter number is zero. If the exception code is E$NDP, the fourth item pushed onto the stack is the NPX status word, and the NPX exceptions have been cleared.

Programs compiled under the SMALL model of segmentation cannot have an alternate exception handler, but must use the default system exception handler. This is because alternate exception handlers must have a LONG POINTER, which is not available in the SMALL model.

MAKING UDI CALLS FROM PL/M-86 AND ASM86 PROGRAMS

This section describes how to make UDI calls from a program, using the DQ$ALLOCATE system call as an example. You can easily generalize from this example to see how to make the other UDI calls. There are two examples: one for a call from a PL/M-86 program and one for a call from an ASM86 program.

The way this chapter shows the DQ$ALLOCATE system call syntax is the following:

```
base$addr = DQ$ALLOCATE (size, except$ptr);
```

There are three parameters: size (which has the WORD data type), except$ptr (which has the POINTER data type), and base$addr (which has WORD data type or the SELECTOR data type, depending on the version of PL/M-86).

Each of the examples that follow request 128 bytes of memory and point to a WORD named "ERR" where the condition code is to be returned.

EXAMPLE PL/M-86 CALLING SEQUENCE

```
DECLARE   ARRAY_BASE    WORD, (or SELECTOR)
          ERR           WORD;
    .
    .
    .
    ARRAYBASE = DQ$ALLOCATE (128, @ERR);
```

EXAMPLE ASM86 CALLING SEQUENCE

```
MOV     AX,128
PUSH    AX          ; first parameter
LEA     AX,ERR
PUSH    DS          ; second parameter
PUSH    AX          ;
CALL    DQALLOCATE
MOV     ARRAYBASE,AX ; returned value
```

This example is applicable to programs assembled according to the COMPACT, MEDIUM, and LARGE models of segmentation. For the SMALL model, omit pushing the DS segment register.

DESCRIPTIONS OF SYSTEM CALLS

This section contains descriptions of the UDI system calls, which are arranged alphabetically. Every system call description contains the following information in this order:

- The name of the system call.
- A brief summary of the function of the call.
- The form of the call as it is invoked from a PL/M-86 program, with symbolic names for each parameter.
- Definition of input and output parameters.
- A complete explanation of the system call, including any information you will need to use the system call.

DQ$ALLOCATE

DQ$ALLOCATE requests a memory segment from the free memory pool.

```
base$addr = DQ$ALLOCATE (size, except$ptr);
```

INPUT PARAMETER

size	A WORD which, if not zero, contains the size, in bytes, of the requested segment. If the size parameter is not a multiple of 16, it will be rounded up to the nearest multiple of 16 before the allocation request is processed. if zero, indicates that the size of the request is 65536 (64K) bytes.

OUTPUT PARAMETERS

base$addr	A SELECTOR, into which the Operating System places the base address of the memory segment. If the request fails because the memory requested is not available, this value will be OFFFFH, and the system will return an E$MEM exception code.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$ALLOCATE system call is used to request additional memory from the free space pool of the program. Tasks may use the additional memory for any desired purpose.

DQ$ATTACH

The DQ$ATTACH system call creates a connection to an existing file.

```
connection = DQ$ATTACH (path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing the pathname of the file to be attached.

OUTPUT PARAMETERS

connection	A TOKEN for the connection to the file.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call allows a program to obtain a connection to any existing file. When the DQ$ATTACH call returns a connection, all existing connections to the file remain valid.

Your program can use the DQ$RESERVE$IO$MEMORY call to reserve memory that the UDI can use for its internal data structures when the program calls DQ$ATTACH and for buffers when the program calls DQ$OPEN. The advantage of reserving memory is that the memory is guaranteed to be available when needed. If memory is not reserved, a call to DQ$ATTACH might not be successful because of a memory shortage. See the description of DQ$RESERVE$IO$MEMORY later in this chapter for more information about reserving memory.

DQ$CHANGE$ACCESS

The DQ$CHANGE$ACCESS lets you change the access rights of the owner of a file (or directory), or the access rights of the WORLD user.

```
CALL DQ$CHANGE$ACCESS (path$ptr, user, access, except$ptr);
```

INPUT PARAMETERS

path$ptr — A POINTER to a STRING containing a pathname of the file.

user — A BYTE specifying the user whose access is to be changed:

Value	User
0	Owner of the file
1	WORLD (all users on the system)

access — A BYTE specifying the type of access to be granted the user. This word is to be encoded as follows. (Bit 0 is the low-order bit.)

Bit	Meaning
0	User can delete the file or directory
1	Read (the file) or List (the directory)
2	Append (the file) or Add entry (to the directory)
3	Update (read and write to the file) or Change Access (to the directory)
4-7	Should be zero

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

In the general iRMX 86 environment, every program is associated with a user object, usually referred to as the default user for the program. The default user consists of one or more user IDs. Each file has an associated collection of user ID-access mask pairs, where each mask defines the access rights the corresponding user ID has to the file. When the program calls DQ$CREATE to create a file or DQ$ATTACH to get another connection to a file, the resulting connection receives all access rights corresponding to user IDs that are both associated with the file and in the default user. The purpose of the DQ$CHANGE$ACCESS system call is to change, for a particular file, the access rights associated with a particular user ID. This has the effect of changing the access granted when the program makes subsequent calls to DQ$ATTACH to get further connections to the file.

In the UDI subset of the iRMX 86 environment, a default user has two IDs. One of them, called the owner ID, is associated with the program. The other, called the WORLD, is associated universally with all programs. DQ$CHANGE$ACCESS can change, for the file, the access mask of either the owner ID or the WORLD.

Changing the access rights for a user ID have no effect on connections already obtained by the program. However, all subsequently-obtained connections reflect the changed access rights.

For more information about user IDs, default users, access masks, WORLD, access rights, owner IDs, and how connections are related to all of these entities, refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL.

NOTE

DQ$CHANGE$ACCESS affects only connections made after the call is issued. It does not affect existing connections to the file.

DQ$CHANGE$EXTENSION

DQ$CHANGE$EXTENSION changes or adds the extension at the end of a file name stored in memory (not the file name on the mass storage volume).

```
CALL DQ$CHANGE$EXTENSION (path$ptr, extension$ptr, except$ptr);
```

INPUT PARAMETERS

path$ptr	A POINTER to a STRING containing a pathname of the file to be renamed.
extension$ptr	A POINTER to a series of three bytes containing the characters to be added to the pathname. This is not a STRING. You must include three bytes, even if some are blank.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This is a facility for editing strings that represent file names in memory. If the existing file name has an extension, DQ$CHANGE$EXTENSION replaces that extension with the specified three characters. Otherwise, DQ$CHANGE$EXTENSION adds the three characters as an extension.

For example, a compiler can use DQ$CHANGE$EXTENSION to edit a string containing the name, such as :AFD1:FILE.SRC, of a source file to the name, such as :AFD1:FILE.OBJ, of an object file, and then create the object file.

Note that iRMX 86 file names may contain multiple periods, but if they do, the extension, if any, consists of the characters following the last period. Note also that an extension may contain more than three characters, but any extension created or changed by DQ$CHANGE$EXTENSION has at most three (non-blank) characters.

The three-character extension may not contain delimiters recognized by DQGETARGUMENT but may contain trailing blanks. If the first character pointed to by extension$ptr is a space, DQ$CHANGE$EXTENSION deletes the existing extension, if any, including the period preceding the extension.

DQ$CLOSE

DQ$CLOSE waits for completion of I/O operations (if any) taking place on the file, empties output buffers, and frees all buffers associated with the connection.

```
CALL DQ$CLOSE (connection, except$ptr);
```

INPUT PARAMETER

connection	A TOKEN for a file connection that is currently open.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$CLOSE system call closes a connection that has been opened by the DQ$OPEN system call. It performs the following actions, in order:

1. Waits until all currently-running I/O operations for the connection are completed.
2. Ensures that information, if any, in a partially-filled output buffer is written to the file.
3. Releases all buffers associated with the connection.
4. Closes the connection. The connection is still valid, and can be re-opened if necessary.

DQ$CREATE

DQ$CREATE creates a new file and establishes a connection to the file.

```
connection = DQ$CREATE (path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing a pathname of the file to be created.

OUTPUT PARAMETERS

connection	A TOKEN for the connection to the file.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This call creates a new file with the name you specify and returns a connection to it. If a file of the same name already exists, it is truncated to zero length and the data in it is destroyed.

To prevent accidentally destroying a file, call DQ$ATTACH before calling DQ$CREATE. If the file does not exist, DQ$ATTACH returns an E$FNEXIST exception code.

DQ$DECODE$EXCEPTION

DQ$DECODE$EXCEPTION translates an exception code into its mnemonic.

```
CALL DQ$DECODE$EXCEPTION (except$code, buff$ptr, except$ptr);
```

INPUT PARAMETER

except$code	A WORD containing the numeric exception code that is to be translated.

OUTPUT PARAMETERS

buff$ptr	A POINTER to a buffer (at least 81 bytes long) into which the system returns the mnemonic in a STRING.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Your program can call DQ$DECODE$EXCEPTION to exchange a numeric exception code for its hexadecimal equivalent followed by its mnemonic. For example, if you pass DQ$DECODE$EXCEPTION a value of 2 in the except$code parameter, the system returns the following string to the area pointed to by the buff$ptr parameter:

```
0002: E$MEM
```

The hexadecimal values and mnemonics for condition codes are listed in Appendix B.

DQ$DECODE$TIME

DQ$DECODE$TIME returns the current system time and date as a Double Word integer and as a series of ASCII character bytes.

```
CALL  DQ$DECODE$TIME (time$ptr, except$ptr);
```

OUTPUT PARAMETERS

time$ptr — A POINTER to a structure of the following form:

```
DECLARE DT STRUCTURE(
    SYSTEM$TIME         DWORD,
    DATE (8)            BYTE,
    TIME (8)            BYTE);
```

If the value in SYSTEM$TIME is 0 when DQ$DECODE$TIME is called, DQ$DECODE$TIME returns the current date and time in the DT structure, as follows. (See the following DESCRIPTION section for format information.):

SYSTEM$TIME receives the time as the number of seconds since midnight, January 1, 1978.

DATE receives the date portion of the time, in the form of ASCII characters.

TIME receives the time-of-day portion of the time, in the form of ASCII characters.

If the value in SYSTEM$TIME is not 0 when DQ$DECODE$TIME is called, DQ$DECODE$TIME accepts that value as the number of seconds since midnight, January 1, 1978, decodes the value, and returns it in the DATE and TIME fields.

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call returns the indicated date and time, each as a series of ASCII bytes. (Note that they are not STRINGs.)

DATE has the form MM/DD/YY for month, day, and year. The two slashes (/) are in the third and sixth bytes. For example, the date January 15th of 1982 would be returned as:

    01/15/82

TIME has the form HH:MM:SS for hours, minutes, and seconds, with separating colons (:). The value for hours ranges from 0 through 23. For example, the time 20 seconds past 3:12 PM would be returned as:

    15:12:20

If, when you call DQ$DECODE$TIME, the SYSTEM$TIME parameter is zero, the call first gets the system time (number of seconds since midnight, January 1, 1978) and then decodes it into the series of bytes as just described.

But if SYSTEM$TIME is not zero on input, DQ$DECODE$TIME uses it as the time to decode.

One thing your program can do with DQ$DECODE$TIME is first to call DQ$FILE$INFO to get two DWORD values associated with a file (the last time the file was updated and the time the file was created). Then the program can call DQ$DECODE$TIME to interpret the times.

DQ$DELETE

DQ$DELETE deletes an existing file.

```
CALL DQ$DELETE (path$ptr, except$ptr);
```

INPUT PARAMETER

path$ptr	A POINTER to a STRING containing a pathname of the file to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

A program can use this system call to delete a file. The immediate action this call takes is to mark the file for deletion. It does this rather than abruptly deleting the file, because it will not delete any file as long as there are existing connections to the file. DQ$DELETE will delete the file only when there are no longer any connections to the file, that is, when all existing connections have been detached. On the other hand, once the file is marked for deletion, no more connections may be obtained for the file by way of DQ$ATTACH.

DQ$DETACH

DQ$DETACH deletes a connection (but not the file) established by DQ$ATTACH or DQ$CREATE.

```
CALL DQ$DETACH (connection, except$ptr);
```

INPUT PARAMETER

connection	A TOKEN for the file connection to be deleted.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call deletes a file connection. If the connection is open, the DQ$DETACH system call automatically closes it first (see DQ$CLOSE). DQ$DETACH also deletes the file if the file has been marked for deletion and this is the last existing connection to the file.

DQ$EXIT

DQ$EXIT transfers control from your program to the iRMX 86 Operating System. It does not return any value to the calling program, not even a condition code.

```
CALL DQ$EXIT (end$code);
```

INPUT PARAMETERS

end$code	A WORD containing the encoded reason for termination of the program. See the following description for information about this value.

DESCRIPTION

DQ$EXIT terminates a program. Before the actual termination, all of the program's connections are closed and detached, and all memory allocated to the program by DQ$ALLOCATE is returned to the memory pool.

DQ$EXIT does not return a condition code to the calling program.

If the calling program is running as an I/O job, the calling task, normally the command line interpreter (CLI), receives an iRMX 86 condition code based on the value your program supplied in the end$code field when it called DQ$EXIT. This assumes the following sequence of events:

1. The CLI calls RQ$CREATE$IO$JOB, specifying a response mailbox in the call.

2. Your program, running as a task in the created I/O job, performs its duties and then calls DQ$EXIT, specifying an end$code value.

3. DQ$EXIT converts the end$code value into an iRMX 86 condition code, as follows:

end$code Value	iRMX 86 Condition Code	Associated Mnemonic	Meaning
0	0C0H	E$UNKNOWN$EXIT	Termination was normal.
1	0C1H	E$WARNING$EXIT	Warning messages were issued.
2	0C2H	E$ERROR$EXIT	Errors were detected.
3	0C3H	E$FATAL$EXIT	Fatal errors were detected.
4	0C4H	E$ABORT$EXIT	The job was aborted.
5-65535	0C0H	E$UNKNOWN$EXIT	Cause of termination not known.

4. DQ$EXIT calls RQ$EXITIOJOB, specifying the iRMX 86 condition code in the user$fault$code field.

5. RQ$EXIT$IO$JOB places the condition code into the user$fault$code field of a message. Then RQ$EXIT$IO$JOB sends the message to the response mailbox set up by the earlier call to RQ$CREATE$IO$JOB.

6. The CLI, when it obtains the message from the response mailbox, can take appropriate actions. Note that it can call DQ$DECODE$EXCEPTION first, to convert the condition code into its associated mnemonic.

The CLI program supplied with the iRMX 86 Operating System ignores these UDI condition codes when they are returned in the user$fault$code field of the response message. Therefore, if you want the CLI to take actions based on that code, you must provide your own CLI.

For more information about RQ$CREATE$IO$JOB, RQ$EXITIOJOB, and the format of the response message, see the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

DQ$FILE$INFO

DQ$FILE$INFO returns information about a file.

```
CALL  DQ$FILE$INFO (connection, mode, file$info$ptr, except$ptr);
```

INPUT PARAMETERS

connection — A TOKEN containing a connection for the file.

mode — An encoded BYTE specifying whether DQ$FILE$INFO is to return the User ID of the owner of the file. Encode as follows:

Value	Meaning
0	Do not return owner's User ID.
1	Return the owner's User ID.

OUTPUT PARAMETERS

file$info$ptr — A POINTER to a structure into which the requested information is to be returned. The form of the structure is:

```
DECLARE FDATA STRUCTURE(
    OWNER(15)          STRING,
    LENGTH             DWORD,
    TYPE               BYTE,
    OWNER$ACCESS       BYTE,
    WORLD$ACCESS       BYTE,
    CREATE$TIME        DWORD,
    LAST$MOD$TIME      DWORD,
    RESERVED(20)       BYTE);
```

where:

OWNER — A STRING containing (if requested) the User ID of the file owner.

TYPE — A value indicating the type of file, as follows:

Value	File Type
0	Data file
1	Directory file

OWNER$ACCESS An encoded BYTE whose bits specify the type of access granted to the owner, as follows. When a bit is set, it means the type of access is granted; otherwise the type of access is denied. (Bit 0 is the low-order bit.)

Bit	Associated Type of Access
0	Delete
1	Read (the data file) or Display (the directory)
2	Append (to the data file) or Add Entry (to the directory)
3	Update (read and write to the file) or Change Access (to the directory)

WORLD$ACCESS An encoded BYTE whose bits specify the type of access granted to the WORLD (all users on the system). When a bit is set, it means the type of access is granted; otherwise the type of access is denied. (Bit 0 is the low-order bit.)

Bit	Associated Type of Access
0	Delete
1	Read (the data file) or Display (the directory)
2	Write (to the data file) or Add Entry (to the directory)
3	Update (read and write to the file) or Change Access (to the directory)

CREATE$TIME The date and time that the file was created, expressed as the number of seconds since midnight, January 1, 1978. (You can convert this date/time to ASCII characters by calling DQ$DECODE$TIME.)

LASTMODTIME	The date and time that the file or directory was last modified. For data files, modified means written or truncated; for directories, modified means an entry was changed or an entry was added. (You can convert this date/time to ASCII characters by calling DQ$DECODE$TIME.)
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$FILE$INFO returns information about a data file or a directory file.

DQ$FREE

DQ$FREE returns to the system a segment of memory obtained earlier by DQ$ALLOCATE.

```
CALL DQ$FREE (base$addr, except$ptr);
```

INPUT PARAMETER

base$addr	A TOKEN containing the base address of the segment to be deleted. This value is the token returned by DQ$ALLOCATE when the segment was obtained.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The DQ$FREE system call returns the specified segment to the memory pool from which it was allocated.

DQGETARGUMENT

The DQGETARGUMENT system call returns arguments, one at a time, from a command line entered at the system console. This command line is either that which invoked the program containing the DQGETARGUMENT call or a command line entered while the program was running.

```
delimit$char =  DQ$GET$ARGUMENT (argument$ptr, except$ptr);
```

INPUT PARAMETER

argument$ptr	A POINTER to a buffer which will receive the argument in the form of a STRING. The buffer must be at least 81 bytes long.

OUTPUT PARAMETERS

delimit$char	A BYTE which receives the delimiter character.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Your program can call GET$ARGUMENT to get arguments from a command line. Each call returns an argument and the delimiter character following the argument.

Your program can use this command in two ways. One way is to get arguments from the command line used to invoke the program at the console. In this case, you can assume that the command line is already in a buffer that has automatically been provided for this purpose.

The other way to use this command is to get arguments from command lines that are entered in response to requests from your program. In this case, your program must supply a buffer when calling DQ$READ, so this is the buffer you want to be used when your program calls DQGETARGUMENT. To set this up, your program must call DQ$SWITCH$BUFFER before the call to DQGETARGUMENT.

A delimiter is returned only if the exception code is zero. The following delimiters are recognized by the iRMX 86 Operating System:

,) (= # ! $ % \ + - > < ~

as well as a space ( ) and all characters with ASCII values in the range 0 through 20H.

Before returning arguments in response to DQGETARGUMENT, the system does the following editing on the contents of the command buffer:

- It strips out ampersands (&) and semicolons (;).
- Where multiple blanks are adjacent to each other between arguments, it replaces them with a single blank. (Tabs are treated as blanks.)
- It converts lowercase characters to uppercase unless they are part of a quoted string.

When returning arguments in response to DQGETARGUMENT, the system considers strings enclosed between matching pairs of single or double quotes to be literals. The enclosing quotes are not returned as part of the argument.

EXAMPLE

The following example illustrates the arguments and delimiters returned by successive calls to DQGETARGUMENT. The example assumes that the contents of the buffer are

```
PLM86 LINKER.PLM PRINT(:LP:) NOLIST
```

The following shows what is returned in this case if DQGETARGUMENT is called five times.

CALL NUMBER	ARGUMENT RETURNED	DELIMITER RETURNED
1	(05H)PLM86	space
2	(0AH)LINKER.PLM	space
3	(05H)PRINT	(
4	(04H):LP:	)
5	(06H)NOLIST	cr

Note that the argument returned has the form of an iRMX 86 string, with the first byte devoted to specifying the length of the string. In the second call, there are ten characters in the argument, so the first byte contains 0AH.

Note that the last delimiter for each example is a carriage return (cr). This is how your program can determine that there are no more arguments in the command line.

DQGETCONNECTION$STATUS

The DQGETCONNECTION$STATUS system call returns information about a file connection.

```
CALL DQ$GET$CONNECTION$STATUS (connection, info$ptr, except$ptr);
```

INPUT PARAMETER

connection — A WORD containing a token for the connection whose status is desired.

OUTPUT PARAMETERS

info$ptr — A POINTER to a structure into which the Operating System is to place the status information. The structure has the following format:

```
DECLARE INFO STRUCTURE(
     OPEN           BYTE,
     ACCESS         BYTE,
     SEEK           BYTE,
     FILE$PTR$      DWORD);
```

Where:

OPEN — 1 if the connection is open; 2 otherwise.

ACCESS — Access privileges of the connection. The right is granted if the corresponding bit is set to 1. (Bit 0 is the low-order bit.)

Bit	Access
0	Delete
1	Read
2	Write
3	Update (read and write)

SEEK — Types of seek supported.

Value	Meaning
0	No seek allowed
3	Seek forward and backward

Other values are not meaningful.

FILE$PTR — This DWORD integer marks the current position in the file. The position is expressed as the number of bytes from the beginning of the file, the first byte being byte 0. This field is undefined if the file is not open or if seek is not supported by the device. (For example, seek operations are not valid for a line printer.)

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

DQGETCONNECTION$STATUS returns information about a file CONNECTION. You might use this system call, for example, if your program has performed several read or write operations and it is necessary to determine where the file pointer is now located.

DQGETEXCEPTION$HANDLER

DQGETEXCEPTION$HANDLER returns the address of the current exception handler.

```
CALL DQ$GET$EXCEPTION (address$ptr, except$ptr);
```

OUTPUT PARAMETERS

address$ptr	A POINTER to a POINTER into which this system call returns the entry point of the current exception handler.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

DQGETEXCEPTION$HANDLER is an system call that returns to your program the address of the current exception handler. This is the address specified in the most recent call, if any, to DQ$TRAP$EXCEPTION. Otherwise the value returned is the address of the system default exception handler.

This routine always returns a two-word pointer, even if called from a program compiled under the SMALL model of segmentation.

DQGETEXCEPTION$HANDLER is used in conjunction with DQ$TRAP$EXCEPTION and DQ$DECODE$EXCEPTION. See the descriptions of these calls for more information.

DQGETSIZE

DQGETSIZE returns the size of a previously-allocated memory segment.

```
size = DQ$GET$SIZE (base$addr, except$ptr);
```

INPUT PARAMETER

base$addr	A TOKEN for a segment of memory that has been allocated by the DQ$ALLOCATE call. This is the same address returned by DQ$ALLOCATE when the segment was allocated.

OUTPUT PARAMETERS

size	A WORD which, if not zero, contains the size, in bytes, of the segment identified by the base$addr parameter. if zero, indicates that the size of the segment is 65536 (64K) bytes.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

The GET$SIZE system call returns the size, in bytes, of a segment. The size of the segment might not be exactly what was originally requested for the segment, because DQ$ALLOCATE allocates memory in 16-byte paragraphs. If a request is for a size that is not a multiple of 16, DQ$ALLOCATE increases the size of the request to the next higher multiple of 16 before acting upon the request.

DQGETSYSTEM$ID

DQGETSYSTEM$ID returns the identity of the operating system providing the environment for the UDI.

```
CALL DQ$GET$SYSTEM$ID (id$ptr, except$ptr);
```

OUTPUT PARAMETERS

id$ptr	A POINTER to a 21-byte buffer into which DQGETSYSTEM$ID places a STRING identifying the operating system.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call returns the string:

iRMX 86

followed by 13 blanks.

DQGETTIME

DQGETTIME returns the current date and time in character format.

```
CALL DQ$GET$TIME (buff$ptr, except$ptr);
```

This system call performs no action except that it returns. It is included only for compatibility with previous versions of the UDI. You should use the DQ$DECODE$TIME system call for this function.

DQ$OPEN

The DQ$OPEN system call opens a file for I/O operations, specifies how the file will be accessed, and specifies the number of buffers needed to support the I/O operations.

```
CALL DQ$OPEN (connection, access, num$buf, except$ptr);
```

INPUT PARAMETERS

connection — A TOKEN for the file connection to be opened.

access — A BYTE specifying how the connection will be used to access the file. This value is encoded as follows:

Value	Meaning
1	Read only
2	Write only
3	Update (both reading and writing)

num$buf — A BYTE containing the number of buffers needed for this connection. Specifying a value larger than 0 implicitly requests that "double buffering" (that is, read-ahead and/or write-behind) is to be performed automatically.

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call prepares a connection for use with DQ$READ, DQ$WRITE, DQ$SEEK, and DQ$TRUNCATE commands. Any number of connections to the same file may be open simultaneously.

The DQ$OPEN system call does the following:

- Creates the requested buffers.
- Sets the connection's file pointer to zero. This a place marker that tells where in the file the next I/O operation is to begin.
- Starts reading ahead if num$buf is greater than zero and the access parameter is "Read only" or "Update."

Selecting Access Rights

The system does not allow reading using a connection open for writing only nor writing using a connection open for reading only. If you are not certain how the connection will be used, specify updating. However, if the specified connection does not support the specified type of access, an exception code is returned.

Selecting the Number of Buffers

The process of deciding how many buffers to request is based on three considerations -- compatibility, memory, and performance.

COMPATIBILITY. If you expect to run your UDI program on other systems, you should request no more than two buffers.

MEMORY. The amount of memory used for buffers is directly proportional to the number of buffers. So you can save memory by using fewer buffers.

PERFORMANCE. The performance consideration is more complex. Up to a certain point, the more buffers you allocate, the faster your program can run. The actual break-even point, where more buffers don't improve performance, depends on many variables. Often, the only way to determine the break-even point is to experiment. However, the following statements are true of every system:

- To overlap I/O with computation, you must request at least two buffers.
- If performance is not at all important but memory is, request no buffers.

Requesting zero buffers means that no buffering is to occur. That is, each DQ$READ or DQ$WRITE is followed immediately by the physical I/O operation necessary to perform the requested reading or writing. Interactive programs should open :CI: and :CO: with a request for no buffers.

If your program normally calls DQ$SEEK before calling DQ$READ or DQ$WRITE, it should request one buffer.

Your program can use the DQ$RESERVE$IO$MEMORY call to reserve memory that the UDI can use for its internal data structures when the program calls DQ$ATTACH and for buffers when the program calls DQ$OPEN. The advantage of reserving memory is that the memory is guaranteed to be available when needed. If memory is not reserved, a call to DQ$OPEN might not be successful because of a memory shortage. See the description of DQ$RESERVE$IO$MEMORY later in this chapter for more information about reserving memory.

DQ$OVERLAY

In systems using overlays, the root module calls DQ$OVERLAY to load an overlay module.

```
CALL DQ$OVERLAY (name$ptr, except$ptr);
```

INPUT PARAMETER

name$ptr	A POINTER to a STRING containing the name of an overlay module. The name must be in uppercase.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

A root module in an overlay system calls DQ$OVERLAY each time it wants to load an overlay module.

If your assembly language or PL/M-86 program uses the DQ$OVERLAY procedure, you should take care to ensure that you link the UDI library to your program correctly. The iAPX 86, 88 FAMILY UTILITIES USER'S GUIDE contains an example of linking an overlay program. This example lists a two-step link process, as follows:

1. Link the root and each of the overlays separately, specifying the OVERLAY control, but not the BIND control, in each LINK86 command.

2. Link all the output modules together in one module, specifying the BIND control, but not the OVERLAY control.

This is the same process you should use when linking your iRMX 86 overlay programs.

In addition, you must link the entire UDI library to the root portion of the program and not to any of the overlays. To do this, use the INCLUDE control to include the UDI externals file when assembling or compiling the root portion of the program. By including this file with the root module, you ensure that the root module makes external references to all UDI routines. This prevents unsatisfied external references when the root is linked to the overlays.

DQ$READ

The DQ$READ system call copies bytes from a file into a buffer.

```
bytes$read = DQ$READ (connection, buff$ptr, bytes$max,
                      except$ptr);
```

INPUT PARAMETERS

connection	A TOKEN for the connection to the file. This connection must be open for reading or for both reading and writing, and the file pointer of the connection must point to the first byte to be read.
buff$ptr	A POINTER to the buffer that is to receive the data from the file.
bytes$max	A WORD containing the maximum number of bytes to be read from the file.

OUTPUT PARAMETERS

bytes$read	A WORD containing the number of bytes actually read. This number is always equal to or less than the bytes$max.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call reads a collection of contiguous bytes from the file associated with the connection. The bytes are placed into the buffer specified in the call.

The Buffer

The buff$ptr parameter tells the Operating System where to place the bytes when they are read. Your program must provide this buffer. DQ$READ copies as many bytes as it is instructed to copy (unless it encounters the end of the file), so if the buffer is not long enough, copying continues beyond the end of the buffer.

Number of Bytes Read

The number of bytes that your program requests is the maximum number of bytes that DQ$READ copies into the buffer. However, there are two circumstances under which the system reads fewer bytes.

- If the DQ$READ detects an end of file before reading the number of bytes requested, it returns only the bytes preceding the end of file. In this case, the bytes$read parameter is less than the bytes$desired parameter, yet no exceptional condition is indicated.

- If an exceptional condition occurs during the reading operation, information in the buffer and the value of the bytes$read parameter are meaningless and should be ignored.

Connection Requirements

The connection must be open for reading or updating. If it is not, DQ$READ returns an exceptional condition.

DQ$RENAME

The DQ$RENAME system call changes the pathname of a file.

```
CALL DQ$RENAME (path$ptr, new$path$ptr, except$ptr);
```

INPUT PARAMETERS

path$ptr	A POINTER to a STRING that specifies the pathname for the file to be renamed.
new$path$ptr	A POINTER to a STRING that specifies the new pathname for the file. This path must not refer to an existing file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call allows your programs to change the pathname of a data file or a directory. Be aware that when you rename a directory, you are changing the pathnames of all files contained in the directory. When you rename a file to which a connection exists -- this is permitted -- the connection to the renamed file remains established.

A file's pathname may be changed in any way, provided that the file or directory remains on the same volume.

DQ$RESERVE$IO$MEMORY

The DQ$RESERVE$IO$MEMORY lets your program reserve enough memory to ensure that it can open and attach the files it will be using.

```
CALL DQ$RESERVE$IO$MEMORY (number$files, number$buffers, except$ptr);
```

INPUT PARAMETERS

number$files	The maximum number of files the program will have attached simultaneously. This value must not be greater than 12. Moreover, no more than 6 of these files may be open simultaneously.
number$buffers	The total number of buffers (up to a maximum of 12) that will be needed at one time. For example, if your program will have two files open at the same time, and each of them has two buffers (specified when they are opened), number$files should be two and number$buffers four.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

DQ$RESERVE$IO$MEMORY sets aside memory on behalf of the calling program, guaranteeing that it will be available when needed later for attaching and opening files. This memory is used for internal UDI data structures when the program requests file connections via DQ$ATTACH and for buffers when the program opens file connections via DQ$OPEN. Memory reserved in this way is not eligible to be allocated by DQ$ALLOCATE. Your program should call DQ$RESERVE$IO$MEMORY before making any calls to DQ$ALLOCATE.

In the call to DQ$RESERVE$IO$MEMORY, you may specify as many as 12 files (that can be attached using the reserved memory) and as many as 12 buffers (that can be requested when opening files).

NOTE

If a program calls DQ$RESERVE$IO$MEMORY after making one or more calls to DQ$ATTACH or DQ$OPEN, the memory used by those calls are immediately applied against the file and buffer counts specified in the DQ$RESERVE$IO$MEMORY call, possibly exhausting the memory supply being requested.

If your program calls DQ$RESERVE$IO$MEMORY more than once in a program, it simply changes the amount of memory reserved.

RESTRICTION

This system call is effective only if your program uses exclusively UDI system calls to communicate with the iRMX 86 Operating System.

DQ$SEEK

DQ$SEEK moves the file pointer associated with the specified connection.

```
CALL DQ$SEEK (connection, mode, move$count, except$ptr)
```

INPUT PARAMETERS

connection — A TOKEN for the open connection whose file pointer is to be moved.

mode — A BYTE indicating the type of file pointer movement being requested, as follows:

Mode	Meaning
1	Move the pointer backward by the specified move count. If the move count is large enough to position the pointer past the beginning of the file, set the pointer to the first byte (position zero).
2	Set the pointer to the position specified by the move count. Position zero is the first position in the file. Moving the pointer beyond the end of the file is permitted.
3	Move the file pointer forward by the specified move count. Moving the pointer beyond the end of the file is permitted.
4	First move the pointer to the end of the file and then move it backward by the specified move count. If the specified move count would position the pointer beyond the front of the file, set the pointer to the first byte in the file (position zero).

move$count — A DWORD specifying how far, in bytes, the file pointer is to be moved.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

When performing non-sequential I/O, your programs can use this system call to position the file pointer before using the DQ$READ, DQ$TRUNCATE, or DQ$WRITE system calls. The location of the file pointer specifies where in the file a DQ$READ, DQ$WRITE, or DQ$TRUNCATE operation is to begin. If your program is performing sequential I/O on a file, it need not use this system call.

It is legitimate to position the file pointer beyond the end of a file. If your program does this and then invokes the DQ$READ system call, DQ$READ behaves as though the read operation began at the end of file. If your program calls DQ$WRITE when the file pointer is beyond the end of the file, the data is written as requested. Be aware that if you expand your file in this manner, the expanded portion of the file can contain undefined information.

DQ$SPECIAL

DQ$SPECIAL specifies whether line editing features are to be available to operators entering information at the console.

```
CALL DQ$SPECIAL (mode, conn$ptr, except$ptr);
```

INPUT PARAMETERS

mode — A BYTE used to specify the mode of terminal input. The values and their meanings are:

Value	Meaning
1	Transparent
2	Line editing
3	Immediate transparent

Each of these types is explained in the DESCRIPTION section.

conn$ptr — A POINTER to a TOKEN for a connection to the :CI: file. The connection must have been established by DQ$ATTACH.

OUTPUT PARAMETER

except$ptr — A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call changes the mode in which your program receives input from a console input device. When your system starts to run, the mode is line editing (mode 2). But by using DQ$SPECIAL you can change from line editing to one of the transparent modes, or back to line editing.

The Line Editing Modes

The meanings of the mode parameter are as follows:

Value	Meaning
1	Transparent. Interactive programs often need to obtain characters from the console exactly as they are typed. This is made possible by transparent mode. In transparent mode, all characters are placed in the buffer specified by the call to DQ$READ. (The only exceptions are CTRL/C, which terminates the program, and CTRL/D, which is discarded.) DQ$READ returns control to the calling program when the number of characters entered equals the number of characters specified in the read request.
2	Line Editing. This option means that the console operator has the opportunity to correct typing errors with special keys before the application program receives the characters typed. Line editing characters and their effects are described following the descriptions of these line editing modes.
3	Immediate Transparent. This option is nearly the same as Transparent 1 mode, except that in Transparent 3 mode DQ$READ returns control to your program immediately after it is called, regardless of whether any characters have been typed since the last call to DQ$READ. If no characters have been typed, this is indicated by the bytes$read parameter of the DQ$READ call. Characters that are typed between successive calls to read the terminal are held in the "type-ahead" buffer.

The Line Editing Characters

The following characters and control characters have the following special editing capabilities on console input when line editing mode (mode 2) is in effect:

CARRIAGE RETURN or LINE FEED	Terminates the current line and positions the cursor at the beginning of the next line. Entering either of these characters adds a carriage return/line feed pair to the input line.
RUBOUT	Deletes (rubs out) the previous character in the input line. Each RUBOUT removes a character from both the screen and the type-ahead buffer, and moves the cursor back to that character position.

CTRL/R	If the current input line is not empty, this character reprints the line with editing already performed. This enables the operator to see the effects of the editing performed since the most recent line terminator was entered. If the current line is empty, CTRL/R reprints the previous line. Additional CTRL/Rs display previous lines until all saved lines have been displayed. After that, each additional CTRL/R displays the last line again.
CTRL/U	Discards the current line and the entire contents of the type-ahead buffer.
CTRL/X	Discards the current input line. It also displays the "#" character at the terminal, followed by a carriage return/line feed.

DQ$SWITCH$BUFFER

DQ$SWITCH$BUFFER substitutes a new command line for the existing one.

```
char$offset = DQ$SWITCH$BUFFER (buff$ptr, except$ptr);
```

INPUT PARAMETER

buff$ptr	A POINTER to a STRING containing the "new" command line, that is, the one whose arguments are to be returned by subsequent calls to DQGETARGUMENT.

OUTPUT PARAMETERS

char$offset	A WORD into which the UDI places a number. This number represents the number of bytes from the beginning of the "old" command line to the last character of the last argument so far processed by DQGETARGUMENT. In other words, the value in char$offset tells how many characters in the old command line have been processed by the time of this call.
except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

When your program is invoked from the console, the Operating System places the invocation command into a buffer. Typically, your program will use DQGETARGUMENT to obtain the arguments in that command. If your program subsequently calls DQ$READ to obtain an additional command line from the console, it can call DQ$SWITCH$BUFFER to designate the buffer with the new command line as that from which arguments are to be obtained when DQGETARGUMENT is called.

You can use DQ$SWITCH$BUFFER any number of times to point to different strings in your program. However, you cannot use DQ$SWITCH$BUFFER to return to the command line that invoked the program, because only the Operating System knows the location of that buffer. Therefore, you should use DQGETARGUMENT to obtain all arguments of the invocation command line before issuing the first call to DQ$SWITCH$BUFFER.

A second service of DQ$SWITCH$BUFFER is that it returns the location of the last byte of the last argument so far obtained from the old buffer by calls to DQGETARGUMENT. Therefore, in addition to using DQ$SWITCH$BUFFER to switch buffers, you can use it after one or more DQGETARGUMENT calls to determine where in the buffer the next argument starts. However, doing this "resets" the buffer, in the sense that the next call to DQGETARGUMENT would return the first argument in the buffer. To return to the desired point in the buffer, where you can continue to extract arguments, call DQ$SWITCH$BUFFER again, but when doing so, use the sum of the starting address of the buffer and the value returned by the previous call to DQ$SWITCH$BUFFER. The following is an example showing how to use the second service of DQ$SWITCH$BUFFER:

```
DECLARE
    mybuffer$ptr        POINTER,
    buff$ptr            POINTER,
    arg$ptr             POINTER,
    buff                STRUCTURE(
                            offset      WORD,
                            segment     WORD) AT (@buff$ptr),
    next$char           WORD,
    char$offset         WORD,
    condition$code      WORD,
    delimit$char        BYTE;
          .
          .
          .

/* initialize buff$ptr and next$char */
    buff$ptr = mybuff$ptr;
    next$char = 0;
          .
          .
          .

/* determine where in the buffer the next argument starts */
    char$offset = DQ$SWITCH$BUFFER( buff$ptr, @condition$code );
         if condition$code <> E$OK then /* do error processing */
    next$char = char$offset + next$char;

/* return to desired point in buffer */
    buff.offset = buff.offset + char$offset;
    char$offset = DQ$SWITCH$BUFFER( buff$ptr, @condition$code );
         if condition$code <> E$OK then /* do error processing */

/* get next argument */
    delimit$char = DQ$GET$ARGUMENT( arg$ptr, @condition$ptr );
         if condition$code <> E$OK then /* do error processing */
          .
          .
          .
```

DQ$TRAP$CC

The DQ$TRAP$CC lets you specify a procedure that is to get control if an operator enters CTRL/C at the console.

```
CALL DQ$TRAP$CC (entry$pnt, except$ptr);
```

INPUT PARAMETER

entry$pnt	A POINTER to the entry point of your CTRL/C procedure.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Normally, when an operator enters CTRL/C at the console, the system empties the type-ahead buffer and aborts the currently-executing program. By calling DQ$TRAP$CC, your program can designate any other procedure, so that it will automatically get control instead whenever CTRL/C is entered at the console.

DQ$TRAP$EXCEPTION

DQ$TRAP$EXCEPTION substitutes an alternate exception handler for the default exception handler provided by the operating system.

```
CALL DQ$TRAP$EXCEPTION (address$ptr, except$ptr);
```

INPUT PARAMETER

address$ptr	A POINTER to a POINTER containing the entry point of the alternate exception handler.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

Normally, the exception handler terminates the program that made the call producing the exception condition and displays a message to that effect on the console screen. DQ$TRAP$EXCEPTION designates an alternative exception handler as the one to which control should pass when an exceptional condition occurs.

See the section EXCEPTION-HANDLING SYSTEM CALLS at the beginning of this chapter for an explanation of the conditions of the stack when your exception handler receives control.

DQ$TRUNCATE

DQ$TRUNCATE moves the end-of-file to the current position of a named file connection's file pointer, thereby freeing the portion of the file lying beyond the file pointer.

```
CALL DQ$TRUNCATE (connection, except$ptr);
```

INPUT PARAMETER

connection	A TOKEN for a connection to the named data file that is to be truncated. The file pointer of this connection marks the place where truncation is to occur. The byte indicated by the pointer is the first byte to be dropped from the file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call truncates a file at the current setting of the file pointer and releases all file space beyond the pointer for reallocation to other files. If the pointer is at or beyond the end of file, no truncation is performed. Unless the file pointer is already at the proper location, your program should use the DQ$SEEK system call to position the pointer before calling DQ$TRUNCATE.

The connection should have write, or read and write access rights, established when the connection was opened.

DQ$WRITE

The DQ$WRITE system call copies a collection of bytes from a buffer into a file.

```
CALL DQ$WRITE (connection, buff$ptr, count, except$ptr;
```

INPUT PARAMETERS

connection	A WORD containing a token for the connection to the file into which the information is to be written.
buff$ptr	A POINTER to a buffer containing the data to be written to the specified file.
count	A WORD containing the number of bytes to be written from the buffer to the file.

OUTPUT PARAMETER

except$ptr	A POINTER to a WORD where the system places the condition code. Condition codes are described in Appendix B.

DESCRIPTION

This system call causes the Operating System to write the specified number of bytes from the buffer to the file.

Connection Requirements

If the connection is not open for writing or updating, DQ$WRITE returns an exception code.

Number of Bytes Written

Occasionally, DQ$WRITE writes fewer bytes than requested by the calling program. This happens under the following two circumstances:

- When DQ$WRITE encounters an I/O error.
- When the volume to which your program is writing becomes full.

Where the Bytes Are Written

DQ$WRITE starts writing at the location specified by the connection's file pointer. After the writing operation is completed, the file pointer points to the byte immediately following the last byte written.

If your program must reposition the file pointer before writing, it can do so by using the DQ$SEEK system call.

***

CHAPTER 3
UDI EXAMPLE

This chapter presents an example of UDI system calls. After the program listing are the compiler and linker commands used to build the program, and a listing of the link map.

THE EXAMPLE LISTING

```
$compact
$optimize(3)
/*.....................................................................
 *
 *  Program UPPER
 *
 *       This program demonstrates the use of UDI file-handling and
 *  command-line-parsing system calls.  The program reads an input
 *  file of characters and converts all lowercase alphabetic characters
 *  to uppercase.  The converted data are written to a second file.
 *
 *  UPPER expects the command line that invokes it to be of the form:
 *
 *       UPPER infile [TO outfile]
 *
 *  (If "TO outfile" is not specified, :CO: is assumed.)
 *.....................................................................
 */

upper: DO;

/* Literal declaration of TOKEN as SELECTOR */

$include(:include:ltksel.lit)

/* External declaration files for UDI system calls */

$include(:include:uexit.ext)
$include(:include:uclose.ext)
$include(:include:uwrite.ext)
$include(:include:uread.ext)
$include(:include:uopen.ext)
$include(:include:ucreat.ext)
$include(:include:ugtarg.ext)
$include(:include:uatach.ext)
$include(:include:udcex.ext)
```

```
DECLARE
    CR      LITERALLY '0DH',
    LF      LITERALLY '0AH',
    E$OK    LITERALLY '0'
    TOKEN   LITERALLY 'SELECTOR';

DECLARE
    co$conn     TOKEN;

$subtitle('check$exception')

/*.....................................................................
 *  Procedure to check an exception code.  If the exception code is
 *  not E$OK, print a message and exit.
 *.....................................................................
 */

check$exception: PROCEDURE(exception, info$p) REENTRANT;
    DECLARE
        exception   WORD,
        info$p      POINTER,
        info        BASED info$p STRUCTURE(
            count       BYTE,
            char(1)     BYTE),
        exc$buf     STRUCTURE(
            count       BYTE,
            char(80)        BYTE),
        dummy       WORD;

    IF exception <> E$OK THEN
        DO;
            CALL dq$decode$exception(exception, @exc$buf, @dummy);

            CALL dq$write(co$conn, @exc$buf.char, exc$buf.count, @dummy);

            CALL dq$write(co$conn, @(': '), 2, @dummy);

            CALL dq$write(co$conn, @info.char, info.count, @dummy);

            CALL dq$write(co$conn, @(CR, LF), 2, @dummy);

            CALL dq$exit(3);
        END;

    END check$exception;
```

```
$subtitle('Main')
/*.....................................................................
 *
 *          --- MAIN PROGRAM ---
 *
 *.....................................................................
 */


    DECLARE st WORD;

    DECLARE
        in$name(50)     BYTE,
        out$name(50)    BYTE,
        in$conn         TOKEN,
        out$conn        TOKEN,
        delim           BYTE;

    DECLARE
        buffer(1024)    BYTE,
        in$bp           POINTER,
        in$char         BASED in$bp BYTE,
        nextchar        BASED in$bp (2) BYTE,
        in$count        WORD,
        i               WORD;



    /*.................................................................
     *  Create a connection to :CO: (console output).
     *.................................................................
     */

    co$conn = dq$create(@(4, ':CO:'), @st);

    CALL dq$open(co$conn, 2, 0, @st);



    /*.................................................................
     *  Ignore the name of the program (the first argument).
     *.................................................................
     */

    delim = dq$get$argument(@buffer, @st);
    CALL check$exception(st, 0);
    IF delim = CR THEN
        CALL dq$exit(0);
```

```
/*.....................................................................
 *  Attach the input file, and open it.
 *.....................................................................
 */

delim = dq$get$argument(@in$name, @st);
CALL check$exception(st, 0);

in$conn = dq$attach(@in$name, @st);
CALL check$exception(st, @in$name);

CALL dq$open(in$conn, 1, 2, @st);
CALL check$exception(st, @in$name);



/*.....................................................................
 *  Find out if there is an output file specified.  If so, attach
 *  and open it.  If not, use :CO: for output.
 *.....................................................................
 */

IF delim <> CR THEN
    DO;
        delim = dq$get$argument(@buffer, @st);
        CALL check$exception(st, 0);
        IF (delim = CR) OR
           (buffer(0) <> 2) OR
           (buffer(1) <> 'T') OR
           (buffer(2) <> 'O') THEN
            DO;
                CALL dq$write(co$conn, @('Invalid output file', CR,
                    LF), 21, @st);
                CALL dq$exit(3);
            END;

        delim = dq$get$argument(@out$name, @st);
        CALL check$exception(st, 0);

        out$conn = dq$create(@out$name, @st);
        CALL check$exception(st, @out$name);

        CALL dq$open(out$conn, 2, 2, @st);
        CALL check$exception(st, @out$name);
    END;
ELSE
    out$conn = co$conn;
```

```
    /*.....................................................................
     *  Read from input, convert, and write to output
     *.....................................................................
     */

    DO WHILE 1;
        in$count = dq$read(in$conn, @buffer, size(buffer), @st);
        CALL check$exception(st, @in$name);
        IF in$count = 0 THEN
            GOTO end$of$file;

        DO i=0 TO in$count-1;
            IF (buffer(i) >= 'a') AND (buffer(i) <= 'z') THEN
                buffer(i) = buffer(i) + 'A'-'a';
        END;

        CALL dq$write(out$conn, @buffer, in$count, @st);
        CALL check$exception(st, @out$name);
    END;
end$of$file:

    /*.....................................................................
     *  Close input and output files, and exit
     *.....................................................................
     */

    CALL dq$close(in$conn, @st);
    CALL check$exception(st, @in$name);

    CALL dq$close(out$conn, @st);
    CALL check$exception(st, @out$name);

    CALL dq$exit(0);

END upper;
```

COMPILING AND LINKING

The program UPPER was compiled and linked on an iRMX 86-based system with the following commands:

```
attachfile :sd:lib/rmx86 as :lib:
plm86 upper.p86
link86 upper.obj, :lib:compac.lib  to upper  bind mempool(5000H)
```

The link map is on the next page.

```
iRMX 86 8086 LINKER, V2.0

INPUT FILES: UPPER.OBJ, :LIB:COMPAC.LIB
OUTPUT FILE: UPPER
CONTROLS SPECIFIED IN INVOCATION COMMAND:
  BIND MEMPOOL(5000H)
DATE:  14/02/83  TIME:  12:05:37

LINK MAP OF MODULE UPPER

LOGICAL SEGMENTS INCLUDED:
LENGTH ADDRESS ALIGN  SEGMENT          CLASS           OVERLAY
 02F6H ------    W    CODE             CODE
 001EH ------    W    CONST            CONST
 0475H ------    W    DATA             DATA
 0454H ------    W    STACK            STACK
 0000H ------    W    MEMORY           MEMORY
 0000H ------    G    ??SEG

INPUT MODULES INCLUDED:
UPPER.OBJ(UPPER)
:LIB:COMPAC.LIB(DQATTACH)
:LIB:COMPAC.LIB(DQCLOSE)
:LIB:COMPAC.LIB(DQCREATE)
:LIB:COMPAC.LIB(DQDECODEEXCEPTION)
:LIB:COMPAC.LIB(DQEXIT)
:LIB:COMPAC.LIB(DQGETARGUMENT)
:LIB:COMPAC.LIB(DQOPEN)
:LIB:COMPAC.LIB(DQREAD)
:LIB:COMPAC.LIB(DQWRITE)
:LIB:COMPAC.LIB(SYSTEMSTACK)



GROUP MAP

GROUP NAME:  CGROUP
   OFFSET  SEGMENT NAME
   0000H   CODE

GROUP NAME:  DGROUP
   OFFSET  SEGMENT NAME
   0000H   CONST
   001EH   DATA

SYMBOL TABLE OF MODULE UPPER

BASE    OFFSET TYPE SYMBOL              BASE    OFFSET TYPE SYMBOL

G(1)    0293H  PUB  DQATTACH            G(1)    029EH  PUB  DQCLOSE
G(1)    02A9H  PUB  DQCREATE            G(1)    02B4H  PUB  DQDECODEEXCEPTION
G(1)    02BFH  PUB  DQEXIT              G(1)    02CAH  PUB  DQGETARGUMENT
G(1)    02D5H  PUB  DQOPEN              G(1)    02E0H  PUB  DQREAD
G(1)    02EBH  PUB  DQWRITE             S(4)    006CH  PUB  SYSTEMSTACK
```

***

APPENDIX A
DATA TYPES

The following data types are recognized by the iRMX 86 Operating System.

BYTE	An unsigned, eight-bit binary number.
WORD	An unsigned, two-byte, binary number.
INTEGER	A signed, two-byte, binary number. Negative numbers are stored in two's-complement form.
POINTER	Two consecutive words containing the base address of a (64K-byte processor) segment and an offset in the segment. The offset is in the word having the lower address.
OFFSET	A word whose value represents the distance from the base address of a segment.
SELECTOR	The base address of a segment.
TOKEN	A word or selector whose value identifies an object. A token can be declared literally a WORD or a SELECTOR depending on your needs.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes that follow it in the string.
DWORD	A 4-byte unsigned binary number.

***

APPENDIX B
iRMX™ 86 CONDITION CODES

This appendix contains the exception codes that are generated by the iRMX 86 Operating System. Exception codes are any condition codes other than E$OK, the normal code. Exception codes are classed as either "Environmental Conditions" or "Programmer Errors", although the latter includes certain hardware errors as well as errors that result from programming.

The values of these exception codes fall into ranges based on the iRMX 86 layer which first detects the condition. Table B-1 lists the layers and their respective ranges, with numeric values expressed in hexadecimal notation.

Table B-1. Exception Code Ranges

Layer	Environmental	Programming
Nucleus	1H to 1FH	8000H to 801FH
I/O Systems	20H to 5FH	8020H to 805FH
Application Loader	60H to 7FH	8060H to 807FH
Human Interface	80H to AFH	8080H to 80AFH
Universal Development Interface	C0H to DFH	80C0H to 80DFH
Reserved for Intel	E0H to 3FFFH	80E0H to BFFFH
Reserved for users	4000H to 7FFFH	C000H to FFFFH

The iRMX 86 NUCLEUS REFERENCE MANUAL gives the value of each code and its associated mnemonic, as well as a short description of its significance. In addition, the table shows the layer(s) of the system that could generate the code, in case you wish to refer the the appropriate manual.

***

INDEX

Primary references are underscored.

***