PC

PLUS Manual

Networks
Backup & Restore
API

REFLECTION®
SERIES
SOFTWARE

Walker Richer & Quinn, Inc.

PLUS Manual

Networks
Backup & Restore
API
Version 4.0

REFLECTION®
SERIES
SOFTWARE

Walker Richer & Quinn, Inc.

Novell, Inc.
Netware for VMS, NACS, and Nasi are trademarks.

Quarterdeck Office Systems
DESQview is a trademark.

Ungermann-Bass, Inc.
NIU is a registered trademark.

Xerox Corporation
Ethernet is a registered trademark.

Walker Richer & Quinn, Inc.
P.O. Box 31876
Seattle, WA 98103-1876
USA

Telephone: (206) 324-0407
TELEX: 311 743 WRQ UR
FAX: (206) 329-7565
Technical Support: (206) 325-4357
Bulletin Board: (206) 322-8047

In Europe
Telephone: +31-70-356-09-63
FAX: +31-70-356-12-44

Reflection *PLUS Manual*
Version 4.0
August 1990

This manual is printed on recycled paper

Contents

Backup and Restore 41

Section 1

Networks

Non-PLUS versions of Reflection include some network support:

Reflection 2 and 4
Include multi-session CTERM and LAT command interfaces

Reflection 1 and 7
Include support for AdvanceNet and Office Extend

PLUS versions include these options and many others.

The configuration fields that affect network connections are as follows:

Reflection 2 and 4
Datacomm port
Data bits/parity
Check parity
Receive pacing
Transmit pacing
Session# (LAN)

Reflection 1 and 7
Datacomm port
Parity
Check parity
Receive pacing
Transmit pacing
Enq/Ack pacing
Session# (LAN)

If you are communicating with a host that uses Enq/Ack pacing, keep **Enq/Ack pacing** on the Datacomm Configuration menu at *YES* (the default). Either the server or the PC must have Enq/Ack pacing for communication with an HP 3000—setting both will not cause a problem. *Termtype=10* is assumed. See *Configuring Network Servers* below for more information.

Use the information below to configure your network asynchronous communications server. Then determine what network you are using and read the related section. Data communication between a PC and a host over a network requires a hardware interface between the PC and the network, and a hardware interface between the host and the network. In addition, there must be software available for Reflection to communicate with the PC hardware interface.

1.1 Configuring Network Servers

It is generally best to configure an asynchronous communications server to do flow control with the host. Reflection will do flow control with the network (and host), but best results are obtained by having the server do flow control as well.

Some servers allow transmit pacing and receive pacing to be configured independently:

- Flow control from an asynchronous server to the host is *transmit pacing*—the host sends an XOFF to the server when the host is not ready to receive data.
- Flow control from the host to an asynchronous server is *receive pacing*—the server sends an XOFF to the host when the server is not ready to receive data.

"Classic" HP 3000

If the host is a "classic" HP 3000, the asynchronous server should be configured to do ENQ/ACK flow control if the server supports ENQ/ACK (most do not). The server can be configured to do XON/XOFF receive pacing (the server sends an XOFF to the HP when its buffer is full), though this is usually not necessary: the HP will only send 80 characters at a time, then send an ENQ, and wait for an ACK.

If the asynchronous server does not do ENQ/ACK, then make these changes:

- Configure Reflection for ENQ/ACK pacing (the default). At the Datacomm Configuration screen, set **Enq/Ack pacing** to *YES*. The throughput will be less, because each ENQ from the host will have to be sent across the network to Reflection.
- Transmit pacing on the server should be none. It should not do XON/XOFF transmit pacing (the host sends an XOFF to the server when the host buffer is full), because the XON character is a DC1, which is the HP 3000 read trigger (host prompt). If the asynchronous server interprets the DC1 as an XON, it considers the character a flow control character, and does not pass it on to Reflection.
- If the server does not allow independent configuration of transmit and receive pacing, its pacing should be set to none.

MPE/XL HP 3000

MPE/XL hosts do not do ENQ/ACK pacing. The server must therefore be configured to do XON/XOFF receive pacing. Transmit pacing should be set to none.

If the server does not allow independent configuration of transmit and receive pacing, then the read trigger (host prompt) must be re-configured in both Reflection and on the host. The read trigger is a DC1 (^{D}1) by default. Changing it to DC4 (^{D}4) is recommended.

- To re-configure the host prompt in Reflection, type the following on the command line:

  ```
  SET HOST-PROMPT "^T"
  ```

- To re-configure on the host, run the TRIGGER program on the HP 3000. This program is available via the WRQ bulletin board (see page iii for the number).

VAX and UNIX

For a VAX or UNIX host, the server should be configured to do XON/XOFF in both directions (receive and transmit). Depending on the network configuration, it may be possible to configure the server to do no pacing and 8-bit file transfers.

Chapter 1

AdvanceNet (OfficeShare)

AdvanceNet support is included in both PLUS and non-PLUS versions of Reflection 1 and 7; it is available only in the PLUS versions of Reflection 2 and 4.

To run Reflection over OfficeShare, install the OfficeShare hardware and software according to the *HP OfficeShare Network Installation and Configuration* manual. Run the USRLOAD program as described in the *HP OfficeShare Network User's Guide.*

1.1 Configuring Reflection

Run Reflection and bring up the Datacomm menu. Use *ADV.NET* in the **Datacomm port** field.

1.2 Using AdvanceNet

Walker Richer & Quinn's Command Interpreter prompt for AdvanceNet appears on the screen as the > character. At the prompt enter the command **CONNECT <nodename>**, where <nodename> is the name assigned by the HP 3000 network manager to the HP 3000 with which you wish to communicate. The Command Interpreter returns a *Connected* message for a successful connection, and an error message for an unsuccessful one. Once the connection is established, proceed as usual.

There are two ways to break a connection:

- Force a disconnect from the PC side.

Reflection 2 and 4
Keystroke: Ctrl-F4
Escape sequence: CSI y
Reflection command: DISCONNECT

Reflection 1 and 7
Keystroke: Ctrl-F10
Escape sequence: ^{E}cf
Reflection command: DISCONNECT

- Force a disconnect from the HP 3000 side with **:BYE.**

When a connection is broken, Reflection displays *Disconnected* and returns to the Command Interpreter. A maximum packet size for file transfer is recommended for better performance.

1.3 Status Messages

Bad character in node name
AdvanceNet message indicating that the node name (name of host computer) has a syntax error. An illegal character (a control character) was entered in the network node name given.

Bad field length in node name
The name entered as the network node name is too long. Check it and reenter the name. The node name has 3 fields, each of which is restricted to 15 characters.

Communications driver software not found
Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Connected
A LAN connection has been established with the requested host computer.

Network Error
The network software reports an unexpected error.

Network Software Missing
AdvanceNet has two error messages for missing software. *Communications driver software not found* means that the software that Reflection talks to is missing. *Network software missing* indicates that the software the communications driver talks to is missing.

Node name missing

AdvanceNet error. You tried to connect to the network without entering a node name. Reenter the connect sequence with the node name.

Unknown node

The name entered for the node is either not a valid node name or the connection request was unsuccessful because the requested node did not respond.

VTCOM Internal Error

The communications driver that Reflection talks to has an internal problem.

VT/DOS Error

This message is displayed when specific error information is not available.

Chapter 2

Office Extend

Support for Fransen/King Ltd.'s Office Extend Network is included in both PLUS and non-PLUS versions of Reflection 1 and 7.

To run Reflection with Office Extend Network, make sure you have version C.0 of the Office Extend software program, and that it has been installed and activated as explained in your Office Extend documentation. After Reflection is run, bring up the Datacomm menu and select *EXTEND* in the **Datacomm port** field. Activate or save this change to your configuration file.

Once the *EXTEND* selection is made, other datacomm parameters— such as baud rate, parity, and pacing—are controlled by Office Extend. Your **Datacomm port** selection *must* be *EXTEND* in order for both Reflection and Office Extend to be running at the same time.

Chapter 3

LAT: Multi-Session Interface

LAT support is included in both PLUS and non-PLUS versions of Reflection 2 and 4; it is available only in the PLUS versions of Reflection 1 and 7.

Reflection's LAT command interpreter lets you create and maintain multiple sessions to one or more LAT services on your local area network with a single copy of Reflection. It also allows you to maintain your sessions while Reflection is not running. For instance, you could log in to the VAX using LAT, start a process on the host, uninstall Reflection (Alt-X), and then later restart Reflection and resume the same session. Commands are similar to those used by the DECserver 200.

3.1 Terminology

A short explanation of the terminology used to describe Reflection's multi-session LAT interface follows.

The *command interpreter session* is the control and data path between Reflection and the LAT command interpreter (or CI). This path is established when the LAT datacomm choice is first activated in Reflection and remains in effect until Reflection is terminated.

The *LAT session* , on the other hand, is the control and data path between the LAT command interpreter and a LAT service (for instance, a VAX or DECserver on the LAN).

LAT: Multi-Session Interface

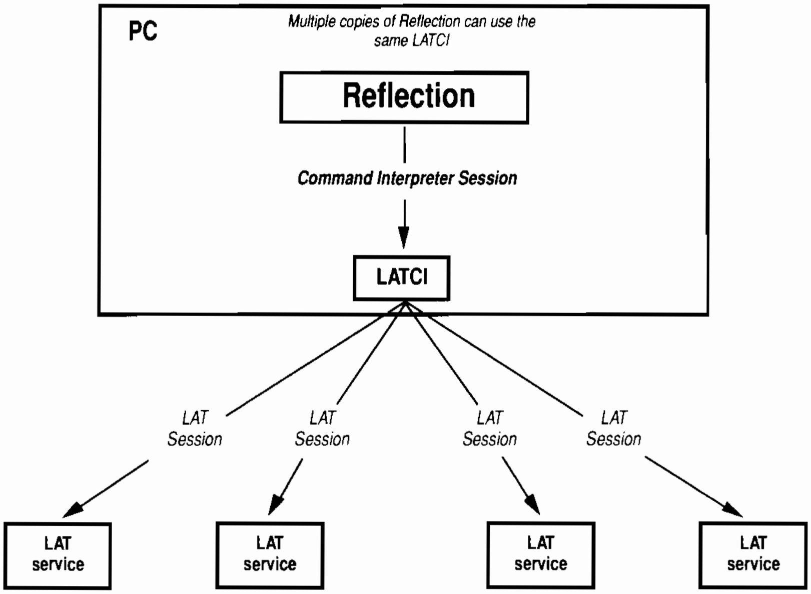

Figure 1: LAT CI Session Support

A LAT session is created when a user connects to a service using the CONNECT command and remains until one of the following happens:

- You log off of the service
- You disconnect the LAT session using the DISCONNECT command
- The service sends a stop slot/message
- The LAT command interpreter is uninstalled using the /U switch

You can switch from your current session to the command interpreter, and switch among existing LAT sessions.

3.2 Installation

To run Reflection over an Ethernet network you must install the Ethernet hardware and do one of the following:

- Install the data link layer and LAT drivers as explained in the Walker Richer & Quinn *R-LAT User Manual.*

- Install the real-time scheduler, data link layer, and LAT drivers as explained in the *DECnet-DOS* or *PCSA/PC* installation guide.

To make use of all of the features provided by the LAT command interpreter, you must invoke it at the DOS prompt before Reflection is started.

If you do not invoke the LAT CI from the DOS prompt, it is automatically installed when you load Reflection, but you cannot change the number of LAT sessions, CI sessions, and receive slots per session. See page 14, *Installing the Default LAT CI*, for more information.

The complete syntax follows:

```
LATCI [/C:<n>] [/L:<n>] [/S:<n>] [/U] [/?]
```

By default, LAT CI supports 4 LAT sessions (each using 4 receive slots) and 2 CI sessions. These parameters can be changed with the following switches:

[/C:<n>]

Sets the number of command interpreter sessions. You need one command interpreter session for each copy of Reflection loaded on your PC. To support two CI sessions, for instance, load Reflection twice—one copy in the background and the other in the foreground. This is useful when loading multiple copies of Reflection in DESQview or Windows.

Values: 1–4
Default: 2

[/L:<n>]

Sets the maximum number of LAT sessions to be supported.

Values: 1–8
Default: 2

[/S:<n>]

Sets the number of receive slots per LAT session.

Values: 2–9
Default: 4

[/U]

Uninstalls LAT CI if it is installed. Cannot be used in combination with any other switch.

[/?]

Displays help information.

If you plan to use both the LAT command interpreter and Reflection's state save feature (Alt-B), LAT CI must be invoked at the DOS prompt— not installed automatically. The commands to load the data link layer, LAT driver, and LATCI.EXE can be included in your AUTOEXEC.BAT file.

Installing the Default LAT CI

If you do not install the LAT command interpreter before running Reflection, you do not receive an error: Reflection loads LAT CI automatically. When loaded in this manner, it functions with some restrictions:

- The following parameters are in effect and cannot be changed:

  ```
  LATCI /C:1 /L:2 /S:4
  ```

  The number of CI sessions is 1, the number of LAT sessions is 2, and the number of receive slots is 4.

- When Reflection is terminated, all LAT sessions are also terminated.

  The stand-alone LAT command interpreter can also be configured to automatically terminate your session on exit. See the discussion of **SET DISCONNECT-ON-EXIT** on page 16.

- You cannot use Reflection's state save feature—Alt-B will not save your session information unless LAT CI has been installed as a TSR.

Configuring Reflection for LAT

1. After running Reflection, configure your Datacomm port field to *LAT*.

Reflection 2 and 4	Press Alt-S to bring up the set-up keys and then select **datacomm set-up**.
Reflection 1 and 7	Press Alt-C to bring up the configuration keys and then press F1, **datacomm config**.

2. Press F1 until *LAT* appears in the **Datacomm port** field. Activate or save this change to your configuration file and press Return.

If you have installed LAT CI as a separate module (from the DOS prompt or from your AUTOEXEC.BAT file), the prompt has upper and lowercase letters:

```
Local>
```

If the LAT CI was installed automatically by Reflection, the prompt is in uppercase and looks like this:

```
LOCAL>
```

See *Installing the Default LAT CI* on page 14 for a comparison of these two methods of initiating the LAT CI.

3.3 Using the Multi-Session LAT Interface

Once the LAT command interpreter has been installed, you can use the CONNECT command to create as many LAT sessions as the command interpreter is configured to support. After a session has been created, you can return to the LAT CI by logging off the service you are connected to, or by entering the following keystroke:

Reflection 2 and 4	Ctrl-F5
Reflection 1 and 7	Ctrl-F8

You can also use the escape sequence ^{E}C*&bR* or the command RESETCOMM. The session you just created remains active.

- To view the available services, use the SHOW SERVICES command.
- To view active sessions, use the SHOW SESSIONS command.
- To continue an active session, use the RESUME command.
- To terminate an active session from LAT CI, use the DISCONNECT command.
- To move among active sessions without returning to the LAT CI, use the keystroke Alt-N. You may need to also press Return to see the prompt of the session that you are resuming. See page 17 for a complete listing of the LAT CI commands.

The keystrokes for moving among active sessions and returning you to the LAT command interpreter are remappable. The following table shows the default keystroke, remapping label, equivalent escape sequence, and Reflection command:

Table 1
LAT Command Interpreter Keystrokes

Keystroke	Remapping Label	Escape Sequence	Reflection Command	Product
Alt-N	next-session	^{E}c&bN	N/A	All
Ctrl-F5	ci-mode	^{E}c&bR	RESETCOMM	Reflection 2 and 4
Ctrl-F8	ci-mode	^{E}c&bR	RESETCOMM	Reflection 1 and 7

See *Keyboard Remapping* in the *Technical Reference Manual.*

3.4 Uninstalling LAT CI

To uninstall LAT CI enter the following command:

```
LATCI /U
```

The LAT CI cannot be uninstalled if you have the command interpreter session still active (for example, if you hot-key out of Reflection).

If you keep an active LAT session after exiting Reflection (you have loaded the command interpreter separately and **DISCONNECT-ON-EXIT** is set to *NO*), you can uninstall LATCI with the /U switch. However, your LAT session will automatically be terminated.

Terminating LAT Sessions

LAT sessions are automatically terminated when you exit Reflection under the following conditions:

- The LAT CI was invoked automatically (not as a separate command at the DOS prompt).
- You installed LATCI separately, but you have **DISCONNECT-ON-EXIT** set to *YES*. See the *Command Language Manual* for more information on this SET command. This is the equivalent of the CI command DISCONNECT ALL.

Otherwise, you can exit Reflection and then resume the same LAT session when you start Reflection again.

3.5 LAT CI Commands

The format and description of each command are listed below. You can abbreviate most of the commands to just one letter; the exceptions are SHOW SERVICES and SHOW SESSIONS. The shortest forms of the commands are S SER and S SES, respectively.

The possible syntax error messages are listed on page 19. The symbol that caused the error is displayed between forward slash characters (/) where applicable.

CONNECT

CONNECT creates a new session to the specified service. You can create a maximum of 4 sessions to the same service.

```
CONNECT {<service>} [<description>]
```

<service>
The name of a valid LAT service; to determine valid services, use SHOW SERVICES

<description>
A string containing no spaces or tabs that is displayed in SHOW SESSIONS output; it can be used to differentiate among multiple sessions to the same service (18 characters maximum)

DISCONNECT

This command lets you disconnect either a single session or all sessions:

```
DISCONNECT {<session number>  | ALL}
```

<session number>
The session number of an existing session; see SHOW SESSIONS on page 19

DISCONNECT terminates the LAT session corresponding to <session number>. If ALL is specified, all LAT sessions not in use by another CI session are terminated. If neither <session number> nor ALL is specified, the session you were most recently using is disconnected.

LAT: Multi-Session Interface

FORWARDS

FORWARDS resumes the session with the next highest session number as shown in the session list.

```
FORWARDS
```

If your current session has the highest session number or you have no current session, FORWARDS resumes the first session in the session list. When the session resumed is in the *stopped* state (see SHOW SESSIONS on page 19), the reason for the stop is displayed, and Reflection returns to CI mode.

HELP

HELP displays documentation on the topic requested. If no topic is specified, a list of valid topics is displayed.

```
HELP [<topic>]
```

Valid topics follow:

CONNECT
DISCONNECT
FORWARDS
HELP
INSTALL
RESUME
SHOW SESSIONS
SHOW SERVICES

RESUME

RESUME resumes the current session if no <session number> is specified. If the session resumed is in the *stopped* state (see SHOW SESSIONS on page 19), the reason for the stop is displayed, and Reflection returns to CI mode.

```
RESUME [<session number>]
```

<session number>
The number of an existing LAT session to be resumed

SHOW SERVICES

SHOW SERVICES lists the names of the services in the LAT service directory. Services are placed in the directory in two ways:

- If you are using DECnet-DOS or PCSA/PC, all LAT services listed in the PC DECnet database are entered in the table when LAT.EXE is installed (no attempt is made to validate the entries).
- The LAT directory service table has a default table size of 10. If you are using R-LAT, a Reflection Complement, the service table size can be changed by using the **/DIR:<n>** switch when installing RLAT.EXE.

```
SHOW SERVICES
```

SHOW SESSIONS

SHOW SESSIONS displays the following information about each active session:

- Session number
- Service name
- Description entered with CONNECT command
- Session status

```
SHOW SESSIONS
```

3.6 Status Messages

Additional parameter(s) required
LAT Command Interpreter syntax error.

Ambiguous command verb
LAT Command Interpreter syntax error. Descriptions of the LAT CI commands begin on page 17.

Ambiguous keyword
LAT Command Interpreter syntax error. Descriptions of the LAT CI commands begin on page 17.

Connected
A LAT Command Interpreter session status response indicating that the connection to the host is active.

Current
LAT Command Interpreter session status response. Indicates that a given session is your current one.

In use
LAT Command Interpreter session status response indicating that the LAT session is the current session of another Command Interpreter user.

Invalid command verb
LAT Command Interpreter syntax error. Descriptions of the LAT CI commands begin on page 17.

Invalid keyword
LAT Command Interpreter syntax error. Descriptions of the LAT CI commands begin on page 17.

Invalid parameter

LAT Command Interpreter syntax error. Descriptions of the LAT CI commands begin on page 17.

LAT session unavailable

LAT Command Interpreter CONNECT command. This message is displayed when the session specified is in use by another Command Interpreter session or is not active. The same message can occur with the RESUME command if the session specified is in use by another Command Interpreter session, or is not active.

No LAT driver installed

LAT Command Interpreter installation. The LAT driver LAT.EXE must be installed prior to installing LAT CI.

No LAT sessions available

LAT Command Interpreter CONNECT command. This message is displayed if you have reached the LAT session quota established when LAT CI was installed.

Parameter too long

LAT Command Interpreter syntax error. Descriptions of the LAT CI commands begin on page 17.

Stopped

LAT Command Interpreter session status response. It indicates that the connection to the host has been terminated. Resuming a stopped session displays the reason the session stopped.

Too many parameters

LAT Command Interpreter syntax error. Descriptions of the LAT CI commands begin on page 17.

LAT connect error

LAT Command Interpreter CONNECT command. This message is displayed if one of the following has occurred:

- The service requested is not in the LAT service directory.
- The LAT driver was unable to allocate resources necessary to create a new slot/circuit.

Chapter 4

CTERM: Multi-Session Interface

CTERM support is included in both PLUS and non-PLUS versions of Reflection 2 and 4; it is available only in the PLUS versions of Reflection 1 and 7.

Reflection's CTERM Command Interpreter lets you create and maintain multiple sessions to one or more DECnet hosts on your DECnet network with a single copy of Reflection. It also allows you to maintain your sessions while Reflection is not installed. For instance, you could log in to the VAX using CTERM, start a process on the host, uninstall Reflection (Alt-X), and then later restart Reflection and resume the same session. Commands are similar to those used by the DECserver 200.

4.1 Terminology

A short explanation of the terminology used to describe Reflection's multi-session CTERM interface follows.

The *Command Interpreter session* is the control and data path between Reflection and the CTERM Command Interpreter (or CI). This path is established when the CTERM datacomm choice is first activated in Reflection and remains in effect until Reflection is terminated (Alt-X).

The *CTERM session*, on the other hand, is the control and data path between the CTERM Command Interpreter and a DECnet host. A CTERM session is created when a user connects to a node using the CONNECT command and remains until one of the following happens:

- You log off of the host node.
- You disconnect the CTERM session using the DISCONNECT command.
- The host terminates the session.
- The CTERM Command Interpreter is uninstalled using the /U switch.

You can switch from your current session to the Command Interpreter, and switch among existing CTERM sessions.

4.2 Installation

To run Reflection over an Ethernet network using the CTERM protocol, you must first install the following:

- The Ethernet hardware
- A series of drivers, which are explained in the *DECnet-DOS* or *PCSA/PC* installation guide (version 2.0 or greater of either product is required)

  SCH.EXE
  DLL.EXE
  DNP.EXE
  CTERM.EXE

To use the CTERM Command Interpreter, it must be installed before running Reflection. The command is entered at the DOS prompt; its syntax follows.

```
CTERMCI [/C:<n>] [/L:<n>] [/U]
```

By default, CTERM CI supports 4 CTERM sessions and 2 CI sessions. These parameters can be changed with the following switches:

[/C:<n>]

Sets the number of command interpreter sessions. You need one command interpreter session for each copy of Reflection loaded on your PC. To support two CI sessions, for instance, load Reflection twice—one copy in the background and the other in the foreground. This is useful when loading multiple copies of Reflection in DESQview or Windows.

Values: 1–4
Default: 2

[/L:<n>]

Sets the maximum number of CTERM sessions to be supported.

Values: 1–8
Default: 2

[/U]

Uninstalls CTERM CI if it is installed. Cannot be used in combination with any other switch.

The commands to load SCH.EXE, DLL.EXE, DNP.EXE, CTERM.EXE, and CTERMCI.EXE can be included in your AUTOEXEC.BAT file.

```
SCH
DLL
DNP [<drive>] [<path>]
CTERM
CTERMCI [/C:<n>] [</L:<n>]
```

At Reflection's **Datacomm port** field, use *CTERM*.

The CTERM command interpreter prompt should appear:

```
CTERM>
```

4.3 Using the Multi-Session CTERM Interface

Once the CTERM Command Interpreter has been installed, you can use the CONNECT command to create as many CTERM sessions as CTERM CI is configured to support. After a session has been created, you can return to the CTERM CI by logging off the node you are connected to, or by entering the following keystroke:

Reflection 2 and 4	Ctrl-F5
Reflection 1 and 7	Ctrl-F8

You can also use the escape sequence ^{E}*C&bR* or the command RESETCOMM. The session you just created remains active.

- To view the available nodes, use the SHOW NODES command.
- To view active sessions, use the SHOW SESSIONS command.
- To continue an active session, use the RESUME command.
- To terminate an active session from CTERM CI, use the DISCONNECT command.
- To move among active sessions without returning to the CTERM CI, use the keystroke Alt-N. You may need to also press Return to see the prompt of the session that you are resuming. See page 17 for a complete listing of the CTERM CI commands.

The keystrokes for moving among active sessions and returning you to the CTERM Command Interpreter are remappable. The following table shows the default keystroke, remapping label, equivalent escape sequence, and Reflection command:

Table 2
CTERM Command Interpreter Keystrokes

Keystroke	Remapping Label	Escape Sequence	Reflection Command	Product
Alt-N	next-session	^{E}C&bN	N/A	All
Ctrl-F5	ci-mode	^{E}C&bR	RESETCOMM	Reflection 2 and 4
Ctrl-F8	ci-mode	^{E}C&bR	RESETCOMM	Reflection 1 and 7

See *Keyboard Remapping* in the *Technical Reference Manual.*

4.4 Uninstalling CTERM CI

To uninstall CTERM CI enter the following command:

```
CTERMCI /U
```

The CTERM CI cannot be uninstalled if you have the command interpreter session still active (for example, if you hot-key out of Reflection).

If you keep an active CTERM session after exiting Reflection (you have loaded CTERM CI separately and **DISCONNECT-ON-EXIT** is set to *NO*), you can uninstall CTERM with the /U switch. However, your CTERM session will automatically be terminated.

Terminating CTERM Sessions

CTERM sessions are automatically terminated when you exit Reflection under the following conditions:

- The CTERM CI was invoked automatically (not as a separate command at the DOS prompt).
- You installed CTERMCI separately, but you have **DISCONNECT-ON-EXIT** set to *YES*. See the *Command Language Manual* for more information on this SET command. This is the equivalent of the CI command DISCONNECT ALL.

Otherwise, you can exit Reflection and then resume the same CTERM session when you start Reflection again.

4.5 CTERM CI Commands

The format and description of each command are listed below. You can abbreviate most of the commands to the number of letters that make it unique, in most cases just one letter (for instance, you can use **SHOW N** instead of **SHOW NODES**).

The possible syntax error messages are listed on page 27. The symbol that caused the error is displayed between forward slash characters (/) when applicable.

CONNECT

CONNECT creates a new session to the specified node.

```
CONNECT {<node>} [<description>]
```

<node>
The name of a valid Phase IV DECnet host; to determine valid nodes use SHOW NODES

<description>
A string containing no spaces or tabs that is displayed in SHOW NODES output; it can be used to differentiate among multiple sessions to the same node (18 characters maximum)

DISCONNECT

This command lets you disconnect either a single session or all sessions:

```
DISCONNECT {<session number> | ALL}
```

<session number>

The session number of an existing session; see SHOW SESSIONS on page 27

DISCONNECT terminates the CTERM session corresponding to <session number>. If ALL is specified, all CTERM sessions not in use by another CI session are terminated.

FORWARDS

FORWARDS resumes the session with the next highest session number as shown in the session list.

```
FORWARDS
```

If your current session has the highest session number or you have no current session, FORWARDS resumes the first session in the session list. When the session resumed is in the *stopped* state (see SHOW SESSIONS on page 27), the reason for the stop is displayed, and Reflection returns to CI mode.

HELP

HELP displays documentation on the topic requested. If no topic is specified, a list of valid topics is displayed.

```
HELP [<topic>]
```

Valid topics follow:

CONNECT
DISCONNECT
FORWARDS
HELP
INSTALL
RESUME
SHOW NODES
SHOW SESSIONS

RESUME

RESUME resumes the current session if no <session number> is specified. If the session resumed is in the *stopped* state (see SHOW SESSIONS on page 27), the reason for the stop is displayed, and Reflection returns to CI mode.

```
RESUME [<session number>]
```

<session number>

The number of an existing CTERM session to be resumed

SHOW NODES

SHOW NODES lists the names of the nodes in the local PC DECnet database. To add nodes to this database, use the DECnet-DOS NCP.EXE utility program.

```
SHOW NODES
```

SHOW SESSIONS

SHOW SESSIONS displays the following information about each active session:

- Session number
- Node name
- Description entered with CONNECT command
- Session status

```
SHOW SESSIONS
```

4.6 Status Messages

Additional parameter(s) required
CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

Ambiguous command verb
CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

Ambiguous keyword
CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

Connected
A CTERM Command Interpreter session status response indicating that the connection to the host is active.

CTERM connect error
CTERM Command Interpreter CONNECT command. This message is displayed if one of the following has occurred:

- The service requested is not in the service directory.
- The CTERM driver was unable to allocate resources necessary to create a new connection.

CTERM session unavailable

CTERM Command Interpreter CONNECT command. This message is displayed when the session specified is in use by another Command Interpreter session or is not active. The same message can occur with the RESUME command if the session specified is in use by another Command Interpreter session, or is not active.

Current

CTERM Command Interpreter session status response. Indicates that a given session is your current one.

In use

CTERM Command Interpreter session status response. Indicates that the CTERM session is the current session of another Command Interpreter user.

Invalid command verb

CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

Invalid keyword

CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

Invalid parameter

CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

No CTERM sessions available

CTERM Command Interpreter CONNECT command. This message is displayed if you have reached the CTERM session quota established when the CTERM command interpreter was installed.

Parameter too long

CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

Stopped

CTERM Command Interpreter session status response. It indicates that the connection to the host has been terminated. Resuming a stopped session displays the reason the session stopped.

Too many parameters

CTERM Command Interpreter syntax error. Descriptions of the CTERM CI commands begin on page 21.

4.7 CTERM Limitations

The following restrictions to the CTERM interface are due to limitations in the DECnet-DOS CTERM implementation.

DOS Commands

Executing local DOS commands from the Reflection command line while a CTERM session is active will be slower than usual due to processing overhead caused by CTERM.EXE.

File Transfer

The are several problems with file transfers using the CTERM interface.

- Only 7-bit file transfers are supported. (The default file transfer mode for Reflection 2 and 4 is 8-bit. Change **File transfer link** on the File Transfer Set-Up screen to *7-BIT*.)
- A small packet size (128 bytes or less) should be used. Change the **Packet size** field on the File Transfer screen. File transfers to the host with a larger block size will work correctly, but after the transfer is completed, characters entered in terminal mode will not be echoed on the screen.
- File transfers to the host are slow.

Chapter 5

Other Networks

Only the PLUS versions of Reflection support the networks described in this chapter. The networks appear in alphabetical order.

5.1 AT&T Starlan

Use *AT&T* in the **Datacomm port** field of the Basic or Datacomm menu. A > prompt appears on the screen.

Once you get the > prompt, type **Connect <nodename>**, where **<nodename>** is the Information Systems Network (ISN) node name. If the connection is successful, the message *Connected* appears on the screen.

Status Messages

If the connection is not successful or if the connection is broken, one of the following messages appears:

Communications driver software not found
Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Connected
A LAN connection has been established with the requested host computer.

Disconnected
You have been disconnected from the network software. The LAN connection was either not established or lost.

Feature not included in this version

This message appears if you select *AT&T* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

Network error

An unexpected network error occurred.

No workstation session resources

The workstation network software cannot connect to a node because the workstation cannot support any more virtual circuits.

Unknown node

The ISN name did not respond to the connection request.

Workstation name not found

The workstation node name has not been configured or installed.

To disconnect from the network use the appropriate keystroke:

Reflection 2 and 4	Keystroke: Ctrl - F4 Escape sequence: CSI y Reflection command: DISCONNECT
Reflection 1 and 7	Keystroke: Ctrl - F10 Escape sequence: $^{E}_{C}$f Reflection command: DISCONNECT

Reflection does not break the connection when you exit Reflection unless DISCONNECT-ON-EXIT is set to *YES*. Enter **SET DISCONNECT-ON-EXIT YES** on the command line and then enter **SAVE R<n>.CFG DELETE** to have this value saved to the current configuration file.

The baud rate setting in Reflection has no meaning when the AT&T interface is used.

5.2 Bridge/3-Com

There are currently three 3Com products with which Reflection is compatible: EtherTerm, PCS/XNS, and PCS1.

- EtherTerm uses the XNS protocols and runs on standard 3Com LAN cards.

- PCS/XNS has replaced EtherTerm. It also uses the XNS protocols and runs on standard 3Com LAN cards. PCS/XNS uses the BAPI (Bridge Application Program Interface) to communicate with Reflection.
- PCS1 uses the TCP/IP protocols and runs an the ILANA LAN card, which was developed by Bridge prior to the merger with 3Com. Like PCS/XNS, PCS1 uses the BAPI (Bridge Application Program Interface) to communicate with Reflection.

For all these products, use *BRIDGE* in the **Datacomm port** field.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *BRIDGE* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.3 EICON

EICON network support is available with Reflection. Use *EICON* in the **Datacomm port** field. You will be able to communicate via an EICON network once the EICON software and hardware have been loaded and properly configured according to their documentation.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *EICON* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.4 HP-TELNET (OfficeShare)

OfficeShare from Hewlett-Packard offers a Telnet virtual terminal option. The device driver TELNET.SYS must be on the PC. To run Reflection over HP's Telnet virtual terminal under OfficeShare, install the OfficeShare hardware and software as explained in the *HP OfficeShare Network Installation and Configuration* manual for a Telnet virtual terminal. Run the USRLOAD program as described in the *HP OfficeShare Network User's Guide*. After running Reflection, use *HP-TELNT* in the **Datacomm port** field.

Walker Richer & Quinn's Command Interpreter prompt for Telnet appears in display memory as the > character. At the prompt enter the command **CONNECT <nodename>**, where <nodename> is the name assigned by the OfficeShare network manager to the host with which you wish to communicate. The Command Interpreter returns a *Connected* message for a successful connection, and an error message for an unsuccessful connection. Once the connection is established, proceed as usual.

There are two ways to break a connection:

- Force a disconnect from the PC side.

  Reflection 2 and 4 Keystroke: Ctrl-F4
  Escape sequence: CSI y
  Reflection command: DISCONNECT

  Reflection 1 and 7 Keystroke: Ctrl-F10
  Escape sequence: ^{E}cf
  Reflection command: DISCONNECT

- Force a disconnect from the HP 3000 side with **:BYE.**

When a connection is broken, Reflection displays *Disconnected* and returns to the Command Interpreter.

Status Messages

Communications driver software not found
Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version
This message appears if you select *HP-TELNT* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.5 IBM's LAN ACS

Reflection supports the *IBM Asynchronous Connection Server* via IBM's extended Interrupt 14 interface. This interface, and the workstation software that the interface requires, are documented in the *IBM Local Area Network Asynchronous Connection Server Program Installation and Configuration Guide.*

You must add the DOS device driver EBIOS.SYS to your CONFIG.SYS file. The programs REDIRECT.EXE and ENABLE.EXE are used to make a network connection to a port on the server. DISABLE.EXE is used to break the network connection. After establishing a network connection, run Reflection. Exiting Reflection breaks the connection.

Use Reflection to configure the baud rate and parity for the port on the server. Baud rates from 300 to 19200 are supported. Reflection also sets XON/XOFF pacing based on the datacomm setting. You can maintain up to 4 sessions. Toggle from one session to the next using the **Session# (LAN)** datacomm field. Session number 1 corresponds to COM1, Session# 2 to COM2, Session# 3 to COM3, and Session# 4 to COM4.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *IBM-ACS* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.6 Interrupt-14

The INT-14 connection may be used to connect to any LANs that use this interrupt. It can also be used with custom drivers written for this interrupt. Use *INT-14* in the **Datacomm port** field. See the *IBM Technical Reference* manual for a description. The port number is set according to the **Session# (LAN)** field.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *INT-14* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.7 Mobius

To run Reflection with Mobius you must first install the hardware and software as explained in the Mobius documentation. Use *MOBIUS* in the **Datacomm port** field.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *MOBIUS* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.8 Novell

Novell Netware is a network operating system. There are several ways for it to be connected to the host:

- Network Products has an ACS2 for Novell. Use *NASI* in the **Datacomm port** field.
- Novell's NACS is supported via the *NASI* choice.
- Novell Netware for VMS is supported via *NET-VMS*.
- Eicon's Access/X.25 is supported via *EICON*. See page 33.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *NET-VMS* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.9 RAF

To run Reflection over RAF, you must install the hardware and software as explained in the RAF documentation. Use *RAF* in the **Datacomm port** field.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *RAF* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.10 TelnetManager

TelnetManager is a Walker Richer & Quinn Complement that implements the TELNET protocol on a PC and provides communications over a TCP/IP network. It is memory-resident PC program that provides two functions for Reflection users:

- The ability for your PC to be a virtual terminal over your TCP/IP connection.
- Session management capability that is seamlessly integrated with Reflection. With an easy-to-use set of commands, the session manager lets you maintain and manage one or more terminal sessions to one or more hosts on the network.

Select *TEL-MGR* for the **Datacomm port** field.

See the TelnetManager product documentation for more information on supported TCP/IP networks.

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *TEL-MGR* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

5.11 Ungermann-Bass

To run Reflection over an Ungermann-Bass network, the Ungermann-Bass hardware and software must be installed as explained in the Ungermann-Bass documentation. After Reflection is loaded select *U.B.* in the **Datacomm port** field.

At this point you will be at the Ungermann-Bass Command Interpreter with its >> prompt. Enter **CONNECT <nodename>**, where <nodename> is the node name assigned by the network manager to the host with which you want to communicate. The Command Interpreter returns *Successful* for a successful connection, and an error message for an unsuccessful connection. Once the connection is established, proceed as usual. There are two ways to break a connection:

- Force a disconnect from the PC side.

  Reflection 2 and 4 Keystroke: [Ctrl]-[F4]
  Escape sequence: CSI y
  Reflection command: DISCONNECT

  Reflection 1 and 7 Keystroke: [Ctrl]-[F10]
  Escape sequence: ^{E}Cf
  Reflection command: DISCONNECT

- Put the connection on hold (refer to the Ungermann-Bass documentation).

  Reflection 2 and 4 [Ctrl]-[F5]

  Reflection 1 and 7 [Ctrl]-[F8]

  You can also use the escape sequence *^{E}C&bR* or the command RESETCOMM.

After a disconnect or a hold, you are back at the Ungermann-Bass Command Interpreter prompt. Note the following set-up considerations:

- The Personal NIU for the PC should be configured with **Break Sequence** and **Hold Sequence** set to *NONE*.
- Change your configuration depending on which Reflection you are using:

Reflection 2 and 4

7-bit file transfer is recommended for the Ungermann-Bass network. Set **File transfer link** on the File Transfer Set-Up menu to *7-BIT*.

Reflection 1 and 7

For an HP 3000 running under MPE, the NIU-130 or NIU-180 should be configured with Pace-NIU set to *NO*, Pace-Dev set to *ENQAK*, and Stop Count set to *80*. Receive Breaks should be set to *YES*.

For an HP 3000 running under MPE/XL, the DTC does not do Enq/Ack flow control. Receive Breaks should be set to *YES*.

To optimize file transfer, set Binary Mode on the NIU-130 or NIU-180 to *YES*.

To optimize terminal operation, do the following:

- Set Binary Mode to *NO*
- Set **File transfer link** at the Reflection File Transfer Configuration menu to *7-BIT*.

- The **Session# (LAN)** field on the Datacomm menu corresponds to the Ungermann-Bass Personal NIU Port number. The default value is **1**, which gives you port #1. (A value of **2** gives you port #2, and so on.)

Status Messages

Communications driver software not found

Reflection can't find the LAN software. The Netbios interface must be installed before loading Reflection.

Feature not included in this version

This message appears if you select *U.B.* for **Datacomm port** and you do not have the PLUS version of Reflection. Call Walker Richer & Quinn about an upgrade.

Section 2

Backup and Restore

Chapter 6

Backing Up Files

Because floppy diskettes are not always reliable or particularly convenient, it is helpful to be able to store backups of your PC files on your host computer. The PLUS option of the Reflection Series lets you back up files to HP 3000, VAX/VMS or UNIX systems.

You can send any number of files to a backup file on the host. They are stored in a special backup format, which prevents them from being accidentally altered. When files need to be restored to the PC for any reason, the RESTORE command is used.

6.1 Using the Backup and Restore Interface

A built-in interface lets you backup files to all supported hosts. From the File Transfer menu, press F5, **BACKUP/RESTORE**. Prompts will walk you through either a backup or restore operation; /E exits the process at any time. WRQBACK.RCL can also be run from the command line or another command file. You can back up either hard disks or diskettes, and ASCII and/or binary files. WRQBACK.RCL assumes you are logged on to the host and able to perform file transfers; i.e., it makes no attempt to configure Reflection or establish a host connection.

6.2 The BACKUP Command

The BACKUP command is entered at Reflection's command line. The complete syntax follows:

```
BACKUP [<d:>][<path>]<filename> [hostfile]
            [/S][/C][/D:<date>][/T:<time>][/L:<filespec>]
```

You should be logged on to the host where you have read/write capability, and the appropriate host file transfer program (PCLINK2, VAXLINK2, or UNXLINK2)* should reside on the host. **File transfer protocol** on the File Transfer Configuration or Set-Up menu should be set to *WRQ*.

6.3 BACKUP Options

Each command parameter is discussed in detail; a summary of options is provided on page 51.

Referencing the Drive [<d:>]

You do not need to specify the drive if the current one is intended. If you want to back up a drive other than the current one, you must indicate it. For example, if you are working from the C: drive and want to back up all the files in the \TRAINING subdirectory on the D: drive, type the following:

```
BACKUP D:\TRAINING\*.* /C
```

(The /C indicates that all files should be backed up, not just the ones that have changed since the last backup.)

Indicating the Directory Path [<path>]

Under the default the current directory is assumed whenever you use the backup command without a path designator. To back up the current directory, simply give the file specification. If you are at a subdirectory level, only that level is backed up.

To have the root directory backed up from a subdirectory location, you must use the root directory designator. The backslash (\) character before the file specification indicates that root directory files are to be backed up.

```
BACKUP \*.*
```

Note that in this example subdirectory files are not included. See the /S option, *Including Subdirectory Files*, on page 47.

If you are below the subdirectory to be backed up, give the path names that precede the subdirectory as follows:

* Note that the Reflection PLUS version of the host file transfer program must be the one installed. The PLUS versions of these programs are compatible with non-PLUS versions of Reflection for file transfer.

```
BACKUP \ level1\ level2\*.*
```

The level2 directory files in the current drive are backed up. Note that none of the files in the root volume directory or in the level1 subdirectory are backed up.

Specifying Files <filename>

You must indicate what file or files are to be backed up whenever you use the BACKUP command. To back up a single file, give the filename and extension. The example assumes the file SECTION1.TXT is in the current drive and directory.

```
BACKUP SECTION1.TXT
```

The complete file specification includes both the filename and extension. DOS file naming conventions should be used and wildcards (* and ?) are acceptable.

The Backup File [<hostfile>]

When files are backed up, a single file is created on the host to store the PC files. The default filename is BACKUP. It is assumed to be the destination file unless another filename is specified. Use the following command to back up files to a different host file:

```
BACKUP *.* <hostfile>
```

Subsequent Backups to the Same File

If you do a complete backup to a host file and then back up the same directory of files again, only the files that have been created or modified since the last backup are transferred to the backup file. Modified files are overwritten with the latest copy; new files are appended to the backup file automatically, as long as there is sufficient room. Files in different directories with the same name will not overwrite each other because directory information is stored with each file.

Note that both PLUS and the DOS backup utility use the archive bit to tell if a file has been modified since the last backup. Any files backed up with DOS and not modified will be ignored when a Reflection backup is done.

Multiple Backup Files

It is possible to maintain several files on the host for backup files, although it is usually unnecessary. Additional files are appended to a backup file automatically and clearly labeled with directory information.

It is usually best and easiest to store all backup files for any one PC in the same file. Reflection does not, however, keep track of drive designations in the host file. If you are backing up more than one drive, create separate files for each one.

Backing Up to an HP 3000

The default host filename is BACKUP. It has a default file size of 160800 records (disc=160800,32,1), which is enough space to back up most hard disks.

Note: All files require at least 2 records on the HP 3000. Because of this, multiple small files require more space than the same number of bytes in a large file. Every record holds 252 bytes. Any *left over* bytes require a complete record. Host and local file sizes on the File Transfer screen have no effect.

To change the name of the default backup file on the HP 3000, use a file equation at the MPE prompt such as:

```
:FILE BACKUP=SAMPLE;DISC=10000,32,8
```

For the HP 3000 any file size that is available on your account can be created. The **32** in the above equation indicates the number of extents into which the file can be broken. The number **8** indicates the initial number of extents opened.

The file BACKUP is retained if it already exists, but files are sent to the file SAMPLE (as long as no host filename is specified in the BACKUP command). SAMPLE in this example could be replaced with any valid MPE filename. It will be the file used in any backups done during this session, unless the default is changed. *LISTEQ* allows you to check the current file equations to see if the default of BACKUP has been directed to another file.

To restore BACKUP as the backup file destination, issue the following command at the MPE prompt:

```
:RESET BACKUP
```

Once the file has been created, you can reference it whenever you want to back up files.

```
BACKUP *.* NEWBACK
```

If you are backing up an entire hard disk, you must create a sufficiently large HP 3000 file before backing up. There are three options:

- Create a new, larger file on the HP 3000, and then specify the new filename for each BACKUP or RESTORE command.
- Create a new, larger file on the HP 3000 and change the default BACKUP filename with a file equation.
- Restore the current BACKUP file to the PC, purge BACKUP and create a new, larger file with the same name.

A 40,000 record file should be adequate for a PC with a 10-megabyte hard disk. To define a new file on the HP 3000 of the desired file size, use the following command at the MPE prompt:

```
FILE NEWBACK;DISC=<numrecords>,32,8
```

Then use NEWBACK as the host filename in the BACKUP command on the Reflection command line.

```
BACKUP C:\*.* NEWBACK /S
```

Lockwords can be used with all host filenames in Reflection, as discussed in the *File Transfer* section of the *Technical Reference Manual.* If the host container file for your backup has a lockword associated with it, you must include it in your backup (or restore) command. For example:

```
BACKUP C:\DOC\*.DOC BACKUP/<lockword>
```

Note that you must always include the host filename, even if, as in this case, you are using the default of BACKUP.

Including Subdirectory Files [/S]

The /S option backs up subdirectory files that exist below the directory being backed up. To back up all subdirectory files below the current directory to the default host file (BACKUP), use the following:

```
BACKUP *.* /S
```

Note that no files above the current directory are included. To back up all the files in the D: drive that have been modified or are new, use the following at any level:

```
BACKUP D:\*.* /S
```

Backing Up Files

Complete Backup [/C]

By default, only files that have been modified or created since the last DOS or PLUS backup are added to the host file when another backup is done. To override this default, use the /C option. *Every* file will be backed up, including ones that have not changed since the last backup.

```
BACKUP *.* /C
```

Note: Both PLUS and the DOS backup utility use the archive bit to tell if a file has been modified since the last backup. Any files backed up with DOS and not modified will be ignored when a Reflection backup is done. Only the /C option ensures that files backed up with DOS will also be backed up with PLUS to the host.

Backing up by Date or Time [/D] [/T]

To streamline the number of files backed up in any single backup operation, use a date or time option. Any files created or modified after the date or time indicated will be backed up. To back up files modified or created after May 29, 1990:

```
BACKUP *.*  /D:5-29-90
```

To back up files modified or created after 1:35 pm:

```
BACKUP *.*  /T:13:35:00
```

The current date is assumed in this example. You can specify both date and time if necessary:

```
BACKUP *.*  /D:5-29-90 /T:13:35:00
```

Log of Backup [/L]

A report of the backup operation is automatically generated whenever a backup is done. The default filename for this report is BACKUP.LOG, and it resides on the PC in the current directory.

Under the default, all backup reports are appended to the same report file. For multiple users of one PC, it might be helpful to have more than one report file. To indicate the name of the file to which the report should be sent, use the /L option with a filename:

```
BACKUP *.*  /L:JONES.LOG
```

A dated log of the backup will be sent to the file JONES.LOG. You can use this file repeatedly: each new report will be appended to the end of the file.

Maintaining the Report File

Because each backup operation log is appended to the report file, you will need to periodically delete some or all of the report file. If you delete the entire file, a new one will be created during the next BACKUP operation.

Printing the Report File

If you would like a hard copy of any backup operation, use the PRN designator* to send the file to the printer directly. No disk file log is kept in this case:

```
BACKUP *.*  /L:PRN
```

6.4 Using Batch Mode for Backups

If backup operations always involve defining several parameters that are fairly consistent, put the command in a command file. Defaults in such a file can include whether the root directory should be included in the backup, what host filename to back up to, whether to log the report in a file or at the printer, and whether to include subdirectory files each time. Then, whenever you need to perform a backup using the defined parameters, you can easily invoke the command file.

1. Go to the Reflection command line.
2. Type in the name of your command file as follows:

```
<filename>.RCL
```

Error codes may be used to indicate what steps should be taken if an error occurs. The IF command provides alternative actions depending on the outcome of a BACKUP operation.

Error codes used for completion of backup jobs are as follows:

```
  0 - Normal completion of backup
101 - No files were found to back up
103 - Backup terminated by user
104 - Backup terminated due to error
```

* Make sure that this refers to a printer on your system.

Backing Up Files

When used in the following command file, the command sequence **LET VI=ERROR-CODE** and **IF V1 > 0** is typical: it allows you to establish responses depending on the result returned. See the *Command Language Manual* for more information.

Reflection 2 and 4

Example using a VAX host:

```
TRANSMIT "^M"
WAIT 0:0:30 FOR "$"
TRANSMIT "<Username>^M^J"
TRANSMIT "<Password>^M^J"
CONTINUE
BACKUP C:\*.*  /S
LET V1=ERROR-CODE
IF V1 > 0
   SET DISABLE-MESSAGES NO
   DISPLAY V1
   ELSE
   DISPLAY "BACKUP COMPLETE^M^J"
;display current directory for
;log file location
   CD
ENDIF
```

Reflection 1 and 7

Example using an HP 3000 host:

```
;CONTINUE
;BACKUP C:\*.* /S /C
;CONTINUE
;BACKUP C:\*.* /S
CONTINUE
BACKUP C:\REFLECT\*.* /S
LET V1 = ERROR-CODE
IF V1 > 0
   SET DISABLE-MESSAGES NO
   DISPLAY V1
   ELSE
   DISPLAY "BACKUP COMPLETE^M^J"
```

```
;display current directory for
;log file location
   CD
ENDIF
```

6.5 Summary of Options

<d:>

Indicates the drive location of files to be backed up (for instance A:, B:, or C:). The default is the current drive.

<path>

Indicates the directory path of the specific directory or subdirectory to be backed up. The default is the current directory.

<filename>

Specifies the name and extension of the file to be backed up. Required parameter. DOS wildcards are acceptable.

<hostfile>

Provides the name of the file on the host that stores the backed up files. The default name is BACKUP. In subsequent backups to this file, new files are appended and existing, modified files are overwritten.

/S

Requests that subdirectory files of the current directory or subdirectory be included in the backup operation. Under the default, files below the directory are not included.

/C

Requests a complete backup of all files in <filename>, whether they have been modified or not. Under the default, only modified or newly created files are backed up.

/D:<date>

Specifies that only files created or modified on or after the date indicated be backed up. DOS format mm-dd-yy is used.

/T:<time>

Specifies that only files created or modified on or after the time indicated be backed up. DOS format hh:mm:ss is used. If no date is given the current date is assumed.

/L:<filespec>

Defines the file where the backup report is to be stored. BACKUP.LOG is the default filename. Logs are appended each time a backup is done. PRN may be used to send the report to a printer.

Chapter 7

HP BACKUP File Directory Information

HPDIR is a utility program provided by Walker Richer & Quinn that runs on the HP 3000. It reads the directory entries in the backup file created by Reflection.

7.1 Installing the HPDIR Utility

The program must be uploaded to your HP 3000; on the command line you can enter UPLHPDIR.RCL after establishing a connection to the HP 3000. To run the program, use the following command at the MPE prompt:

```
RUN HPDIR.PUB.ACCT[;INFO="parameter string"]
```

Running the program without supplying a parameter string will cause the program to display the syntax of the command. Ctrl-Y exits HPDIR and returns to the host prompt.

The INFO= parameter takes the following form:

```
;INFO = "[<path>][<filename>] [<hostfile>] [/S][/N]"
```

<path>

A PC path name that MUST begin with a backslash (\). If omitted, the root directory is assumed.

<filename>

A PC filename. Wildcards (* and ?) are permitted.

<hostfile>

An MPE filename. This is the name of the backup file the program will look for on the HP 3000. This parameter is optional; the default HP filename is BACKUP.

/S

The subdirectory search parameter. If used, all subdirectories will be searched. The default is that subdirectories will not be searched.

/N

The name-only parameter. If used, only the names of the PC files will be displayed. The default is that the name, the size, creation date and time, and backup date and time will be displayed.

The simplest way to list all the PC files in the file BACKUP is as follows:

```
RUN HPDIR.PUB.ACCT;INFO="\*.* /S"
```

This would result in a display of the form:

```
\level1\file1
Size = 99999  Create/Mod = mm/dd/yy hh:mma  Backup = mm/dd/yy hh:mmp

\level1\file2
Size = 99999  Create/Mod = mm/dd/yy hh:mma  Backup = mm/dd/yy hh:mmp

\level1\level2\file3
Size = 99999  Create/Mod = mm/dd/yy hh:mma  Backup = mm/dd/yy hh:mmp

\level1\level2\level3\file4
Size = 99999  Create/Mod = mm/dd/yy hh:mma  Backup = mm/dd/yy hh:mmp
```

The following is an example of a command file that runs the HPDIR program.

```
;save this file under the name HPDIR.RCL
LET V4 = 'RUN HPDIR.PUB.ACCT;INFO="'
LET V4 = V4 & V1 & " " & V2 &  " " & V3 & '"'
TRANSMIT V4
WAIT FOR "^Q"
```

To list all of the files in a default backup file (BACKUP), type the following line from the Reflection command line:

```
HPDIR.RCL /S
```

To list a particular file in a unique backup file (not BACKUP), type:

```
HPDIR.RCL \<path>\<filename> <hostfile>
```

7.2 Extracting Backup Files on the HP 3000

XTRACT is a Reflection utility program for the HP 3000 that extracts a single file from a BACKUP file to a binary, variable record file on the HP 3000. The file is the equivalent of what would be on the HP 3000 after a binary file transfer from the PC.

Uploading XTRACT

A command file is provided that will upload the XTRACT program called *UPXTRACT.RCL*. You should be signed onto the account the program will reside in, such as PUB.SYS, and have save access in that account. Then bring up the command line and enter this command filename. This program file requires the **;Label** option when being uploaded to the HP 3000.

Using XTRACT

The XTRACT program looks for a file named BACKUP on the HP 3000. If your backup file has a different name, enter a file equation using the following format:

```
FILE BACKUP=<hostfile>
```

Note: Overriding the default record size for the extract file (244 characters) with a file equation will lead to serious problems. Don't do it.

The syntax for running the program is as follows:

```
RUN XTRACT;INFO="<path><filename> [<hostfile>]"
```

<path>

A PC path name that MUST begin with a backslash (\) to indicate the root directory. The path is required. (Running the program without supplying any parameters causes the program to display the program syntax.)

HP BACKUP File Directory Information

<filename>

A PC filename. Wildcards (* and ?) are not permitted. The filespec is required.

<hostfile>

An MPE filename. This is the name given to the extracted file on the HP 3000. This parameter is optional; the default filename is EXTRACT.

The XTRACT program can be set up to run as a UDC if desired. Otherwise users can invoke XTRACT manually on the HP 3000 or set up a command file.The following is an example of a command file that runs XTRACT.

```
;save this file under the name XTRACT.RCL
LET V4 = 'RUN XTRACT;INFO="'
LET V4 = V4 & V1 & " " & V2 & '"'
TRANSMIT V4
WAIT FOR "^Q"
```

Invoke this command from the Reflection command line as follows:

```
XTRACT.RCL \<path>\<filename>] [<hostfile>]
```

The path and filespec information is loaded into the variable V1, and the HP filename is loaded into V2 if used.

Chapter 8

VAX and UNIX BACKUP File Directory Information

BACKDIR is a utility that runs on the VAX or UNIX host and lists the files within a Reflection backup file. Backup files are created on the host when you back up your PC using the Reflection PLUS BACKUP command. BACKDIR provides capabilities similar to the DIR command by allowing you to see which files are contained in a given Reflection backup file.

The installation and syntax of BACKDIR differ depending on whether the host is a VAX or UNIX machine.

8.1 Installing the VAX BACKDIR Utility

The VAXLINK2 host file transfer program must be uploaded before BACKDIR can be installed on your host.

A number of files supplied on your Reflection disks are involved in the installation of BACKDIR:

VAXDIR.EXE
VAXDIR.DCL
VAXDIR.RCL

VAXDIR.RCL is a Reflection command file that automatically uploads VAXDIR.EXE and VAXDIR.DCL as BACKDIR.EXE and BACKDIR.COM. Once you have successfully executed the VAXDIR.RCL command file, BACKDIR will be installed and available for use.

VAX and UNIX BACKUP File Directory Information

To perform the installation, do the following:

1. Run Reflection and log in to your VAX host computer.
2. Go to the Reflection command line (press Alt-F10 in Reflection 2 and 4; press Alt-Y in Reflection 1 and 7).
3. Position to a drive and directory that contains all of the VAXDIR.[ext] files. This could be a product diskette or, if you copied these files during Reflection installation, your regular Reflection directory.
4. Type VAXDIR.RCL and Return. The VAXDIR.RCL command file will begin executing and will keep you posted regarding its progress and completion status.

Syntax

The syntax for using the BACKDIR utility follows:

```
@BACKDIR \<filename> <hostfile> [/S] [/N]
```

<filename>

Describes the file(s) to be found in the backup file. It must begin with a backslash, and may contain wildcard characters (* and ?). The default for this parameter is *.*, meaning all files in the root directory.

<hostfile>

The name of the backup file itself. This container file stores all of the individual PC files. The default for this parameter is BACKUP.

/S

Specifies the subdirectories option. It indicates that you want all subdirectories of the directory specified in <filename> to be included on the list of filenames.

/N

The name-only option indicates that you want only the names of those files that match <filename> to be displayed on your screen. The display of file size and creation/backup dates and times will be suppressed.

Examples:

- Gives a condensed listing (filenames only) of all the files in the root directory and all of its subdirectories.

```
@BACKDIR \*.*/S /N
```

- Searches for a file named MYPROG.EXE within a directory named MYDIR.

```
@BACKDIR \MYDIR\MYPROG.EXE
```

- Lists all the files in directory MYDIR, including all of its subdirectories, which have four-character filenames beginning with the letters **MY** and ending with the extension **.BAT**. The backup filename is MYBK0930.RBK.

```
@BACKDIR \MYDIR\MY??.BAT MYBK0930.RBK /S
```

Tips for VAX BACKDIR

Parameters are passed to the BACKDIR.EXE program by defining BACKDIR as a foreign command. This definition is done each time BACKDIR.COM is executed. Then the parameters that were given when invoking the command file are passed to the program.

Another method of implementation is to have your VAX system manager make BACKDIR a global, system-wide, foreign command by including its definition in the SYLOGIN.COM file in the system manager's account.

Copy the BACKDIR.EXE program file into a system account where it will be available for execution by all users. If this is done, all copies of BACKDIR.COM and BACKDIR.EXE in individual user's directories can be deleted. The @ prefix should then be dropped from the BACKDIR syntax when invoking the utility.

8.2 Installing the UNIX BACKDIR Utility

BACKDIR also runs on the UNIX operating system and lists the PC files within a Reflection backup file. The UNXLINK2 host file transfer program must be uploaded before BACKDIR can be installed on the host.

Two files supplied on your Reflection disks are involved in the installation: UNIXDIR.C and UNIXDIR.RCL. The latter is a Reflection command file that automatically uploads the source code UNIXDIR.C and compiles and links the program. Once you have successfully executed the UNIXDIR.RCL command file, BACKDIR will be installed and available for use.

VAX and UNIX BACKUP File Directory Information

To perform the installation do the following:

1. Run Reflection, and log in to your UNIX host computer.
2. Bring up the Reflection command line (Alt-F10).
3. Position to a drive and directory that contains the UNIXDIR.[ext] files. This could be a product diskette or, if you copied these files during Reflection installation, your regular Reflection directory.
4. Type UNIXDIR.RCL and press Return. The UNIXDIR.RCL command file will begin executing and will keep you posted regarding its progress and completion status.

Syntax

The syntax for using the BACKDIR utility follows:

```
BACKDIR "<filename>" <hostfile> [/S] [/N]
```

"<filename>"

Describes the file(s) to be found in the backup file and may contain wildcard characters (* and ?). The PC file specification must be enclosed in quotations. The default for this parameter is "*.*", meaning all files in the root directory.

<hostfile>

The name of the Reflection backup file. This is the container file in which all the individual PC files are stored. The default for this parameter is BACKUP.

/S

Specifies the subdirectories option. It indicates that you want all subdirectories of the directory specified in **<filename>** to be searched.

/N

The name-only option indicates that you want only the names of those files that match **<filename>** to be displayed The display of file size and creation/backup dates and times will be suppressed.

Examples of valid BACKDIR commands:

- Gives condensed listing (filenames only) of all the files in the root directory and all its subdirectories.

```
BACKDIR /S /M
```

- Searches for a file named PROG.EXE within a directory named PRGRM.

```
BACKDIR "\PRGRM\PROG.EXE"
```

- Lists all the files in directory BATCH and all of its subdirectories that have four-character filenames beginning with the letters **MY** and having the extension **.BAT**. The backup filename is FULL.RBK

```
BACKDIR "\BATCH\MY??.BAT" FULL.RBK /S
```

Chapter 9

Restoring Files to the PC

RESTORE is used to return files to the PC from a backup file on the host. Because backup files are stored in a special format, RESTORE is needed to return them to a PC directory in a usable format.

A file that has been backed up contains path directory information, filename, and modification (or creation) date and time. When the backup file is restored to the PC, all of these characteristics are checked, and specified files are restored. If the backup copy of the file exists on the PC, the host file overwrites the PC version under the default.

You may use the supplied back up and restore command file if you want to be prompted through the restore process. Press F5 from the File Transfer menu, or enter WRQBACK.RCL directly on the command line. This backup and restore interface is explained on page 43.

9.1 The RESTORE Command

The RESTORE command should be entered as a single line. The entire syntax follows:

```
RESTORE [<hostfile>] [<d:>][<path>]<filename>
        [/S][/K][/H][/L:<filespec>]
```

It is important to make sure that enough space exists on the PC for all of the files being restored. It is possible that as more files are appended to the backup file from this directory, they will become too numerous to restore easily. (Note that no subdirectory files are included in the above restore operation.)

Each file is restored, regardless of whether the file on the PC has been modified since the backup. See below for more on how to save the most current files when restoring.

9.2 RESTORE Options

Each command parameter is discussed in detail. See page 67 for a summary of the options.

The Backup File [<hostfile>]

The host file BACKUP is the default source from which files are to be restored. If another host file contains the backed up files, use that filename in the RESTORE command as follows:

```
RESTORE NEWBACK *.*
```

This restores files from the host file NEWBACK to the current directory on the current drive.

HP 3000 users should note that if the host container file for your backup has a lockword associated with it, you must include it in your restore command. For example:

```
RESTORE C:\DOC\*.DOC BACKUP/<lockword>
```

Note that you must always include the host filename, even if, as in this case, you are using the default of BACKUP.

Referencing the Drive [<d:>]

You do not need to specify the drive if the current one is intended. Files will be restored to the current drive by default.

If you want to restore files to a drive other than the current one, you must indicate the drive. For example, if you want files on the host to be restored to a diskette use a drive designator as follows:

```
RESTORE B:*.*
```

Indicating the Directory Path [<path>]

Under the default the current directory is assumed. To restore to the current directory, simply give the filespec. If you are at a subdirectory level, only that level is restored.

To have the root directory restored from a subdirectory location, you must use the root directory designator (\).

```
RESTORE \*.*
```

Note that in this example no subdirectory files will be included. See the /S option below.

If you are above the subdirectory being restored, give the path names that precede the subdirectory as follows:

```
RESTORE \ level1\ level2\*.*
```

The level2 directory files in the current drive will be restored. Note that none of the files in the root volume or in the level1 subdirectory are restored. If you restore files to another PC, subdirectories are created for you in the new location, if they do not exist already.

Specifying Files <filename>

The PC file specification is the only required parameter in the RESTORE command. DOS wildcards (* and ?) are acceptable. To restore a single file, give the filename and extension. The example assumes the file SECTION1.TXT is located in the current drive and directory.

```
RESTORE SECTION1.TXT
```

Including Subdirectory Files [/S]

The /S option allows you to restore any subdirectory files that exist below the directory being restored. For example, to restore all subdirectory files below the current directory use the following:

```
RESTORE *.* /S
```

Note that no files above the current directory are included. To restore a complete hard disk from any location use the following:

```
RESTORE \*.* /S
```

Preserving Modified Files [/K]

Unless otherwise specified, RESTORE will overwrite any existing file on the PC with the restored file if it has the same path and filename. To prevent this, use the /K (keep) option. This parameter preserves the most recent version of any file. If an existing file on the PC is more current than the backup version, the backup version of the file will not be restored.

Overwriting Hidden Files [/H]

Under the default, RESTORE does not restore any hidden files from the backup file. If you want to restore all hidden files in the selected directory, use /H. Use this option with caution; hidden and read-only files on the PC are overwritten when /H is used, which may damage some copy-protected programs.

Log of Restore [/L]

A report of the restore operation is automatically generated whenever a restore is done. The default filename for this report is RESTORE.LOG. This file resides on the PC.

Maintaining the Report File

Because each restore operation log is appended to the report file, you will need to periodically delete some or all of the report file. If you delete the entire file, a new one will be created during the next RESTORE operation.

Printing the Report File

If you would like a hard copy of a restore operation, use the PRN designator* to send the file to the printer directly. No disk file log is kept in this case:

```
RESTORE *.*  /L:PRN
```

9.3 Using Batch Mode for Restore

Reflection command language allows you to perform restores in batch mode. This is helpful if restores are routine. Commands necessary for doing a typical restore are entered in a command file with any needed parameters defined. To invoke a command file, use the following steps:

1. Go to the Reflection command line (press Alt-F10 in Reflection 2 and 4; press Alt-Y in Reflection 1 and 7).
2. Type in the name of your command file as follows:

   ```
   RESTORE.RCL
   ```

   where you can replace RESTORE.RCL with your command filename.

See the *Command Language Manual* for more information. See page 49 for information about backup completion codes.

* Make sure that this refers to a printer on your system.

9.4 Summary of Options

<hostfile>

Names the file on the host that contains the backed up files.

<d:>

Indicates the drive that is to receive the files (for instance A:, B:, or C:). The current drive is the default.

<path>

Indicates the directory path for the root or subdirectory to be restored. The default is the current directory.

<filename>

Identifies the specific file to be restored as defined by the filename and extension. DOS wildcards are acceptable. This is a required parameter.

/S

Restores subdirectory files of the current directory as well. Under the default, subdirectory files are not included.

/K

Requests that any files modified since the last DOS or PLUS backup not be overwritten.

/H

Restores hidden files. Use this option with caution; hidden and read-only files are overwritten when /H is used.

/L:<filespec>

Defines the file where the restore report is to be stored. RESTORE.LOG is the default filename. Logs are appended each time a restore is done. PRN may be used to send the report to a printer.

Section 3

Application Program Interface

Reflection's multitasking feature lets you run two programs simultaneously on your PC. You can run a program in foreground, and at the same time be transferring files from the PC to a host computer in the background.

Reflection's Application Program Interface (API) provides a way for foreground programs to *hook* into Reflection so that they can initiate file transfers, log on to a host computer, dial a modem, or perform some other communications function. The foreground program controls a background copy of Reflection, and makes it do anything that a user normally does:

- Command language
- Keystroke entry
- File transfers
- Screen display
- Popping Reflection to the foreground

9.1 Support Files

Application Program Interface support is included in all Reflection PC products. However, only PLUS versions of Reflection contain libraries for C, Pascal, and BASIC: the libraries are on a disk labeled *Application Program Interface.* The API disk also includes a full description and examples of each

API function call. The following is an overview of the support files and demo programs that are included on the API disk:

File or directory name	Description
API.DOC	Contains a full description of each API function call with examples
<APIDEMO> Directory	*Utility programs that use the API feature*
HPCMD.EXE	Run HP commands at the DOS prompt and see the results displayed on the PC screen
HPCMD.C	Source for HPCMD.EXE
DOAPI.EXE	Send a command to Reflection in background to transfer a file or run a command script
DOAPI.C	Source for DOAPI.EXE
$.EXE	Run VAX commands from the DOS prompt with results displayed on PC screen
HP.BAT	Sample DOS batch file that demonstrates how HPCMD.EXE (named LISTF.EXE) can be used to check for a file on the HP 3000 from DOS
APIDEMO.DOC	Documentation on the above files
<Pascal> Directory	*Turbo Pascal library support for API*
APIUNIT.PAS	Source of API UNIT file
APIUNIT.TPU	TPU Unit file–created by compiling APIUNIT.PAS: Code can be included into your Pascal program via the 'uses' statement
P_APILIB.OBJ	API library in OBJ format. Used to create new APIUNIT.TPU. You may customize APIUNIT.PAS as needed and recompile via the TPC command.
SAMPLE.PAS	Sample Pascal program which does API calls
SAMPLE.EXE	Compiled version of SAMPLE.PAS

<BASIC> Directory	***Microsoft QuickBASIC library support for API***
API.INC	Include file listing all function declarations. Use the $INCLUDE metacommand to include this file in your BASIC source.
SAMPLE.BAS	Sample BASIC program using API calls
SAMPLE.EXE	Compiled version of SAMPLE.BAS
B_APILIB.QLB	QuickBASIC Library with API calls. Use with QuickBASIC environment
B_APILIB.LIB	Standard API LIB for use with command line BC compiler and linker
BAS_API.DOC	List of API function declarations

<C> Directory	***C library support for API***
API.H	Include file with API structure definitions
SAMPLE.C	Sample program that uses API calls
SAMPLE.EXE	Compiled version of SAMPLE.C
C_SAPI.LIB	Small model API library
C_CAPI.LIB	Compact model API library
C_MAPI.LIB	Medium model API library
C_LAPI.LIB	Large model API library

Hooks are provided for API applications written in assembler, C, Borland's Turbo Pascal, and Microsoft's QuickBASIC.

While Reflection contains API support, Reflection must be loaded with the /W switch in order to activate it. Using this switch adds about 4000 bytes to Reflection. When loaded and in the background, Reflection can be used via a foreground API application. If Reflection is in the foreground, the API application freezes and the user has direct control over Reflection operations.

Application
Program Interface

Chapter 10

Summary of API Function Calls

The API function calls are summarized here. If you have the Reflection PLUS option, the support disks contain a full library with examples. A total of 33 function calls are available in this release of Reflection. A full description of each function call with examples follows and is also provided on your *Application Program Interface* disk. Currently C, Turbo Pascal, and Microsoft QuickBASIC libraries (on disk) are available for linking with API applications.

10.1 Status Calls

Determine Reflection and API status.

Function call	*Description*	*See page*
api_rcheck	See if Reflection installed	131
api_instchk	See if API present	117
api_getinfo	Get static information	103
api_getstatus	Get dynamic information	109

10.2 Command Language

Synchronous—commands that must complete before control is returned to the API application.

Function call	*Description*	*See page*
api_startcommands	Start a command sequence	154
api_docommand	Do a command	92

api_getvar	Return a command language variable	113
api_setvar	Set a command language variable	151
api_found	Return value of the FOUND boolean	97
api_endcommands	End a command sequence	95

Asynchronous—commands that are queued by Reflection so that the API application does not have to wait for completion.

Function call	*Description*	*See page*
api_cmdstatus	How much free space is in the API queue	90
api_qcmd	Queue a new command	126
api_clrcmdq	Flush the queue of commands	86

10.3 Keyboard Support

Queue keystrokes.

Function call	*Description*	*See page*
api_keystatus	How much free space is in the API queue	119
api_qkeys	Queue some keystrokes	128
api_clrkeybd	Flush the keyboard queue	88
api_getfkey	Get a function key	99
api_setfkey	Set a function key	149

10.4 Screen Support

Read and search Reflection's screen.

Function call	*Description*	*See page*
api_screenread	Read the text screen (minus function keys)	141
api_atrbscreen	Read screen with character attributes	80
api_searchscreen	Search the screen for a string	145

10.5 Datacomm Support

Read and write to datacomm from the API application.

Function call	*Description*	*See page*
api_rdchar	Read a character from com port (may be lan)	133
api_writeasync	Write character to com port (may be lan)	158

api_xmitstatus	Check status of transmit buffer	160
api_releasedc	Release the datacomm port	137
api_assertdc	Re-assert control over the datacomm port	79

10.6 Session Support

Start, control, and end API sessions.

Function call	*Description*	*See page*
api_wait	Wait until Reflection is free	156
api_block	Give Reflection some CPU time	83
api_popup	Pop-up Reflection	124

10.7 Miscellaneous

Create buffers, reset Reflection, and pop Reflection into foreground.

Function call	*Description*	*See page*
api_reset	Do a hard reset of Reflection	139
api_offerbuf	Provide buffers for queuing keys/commands	121
api_cancelbuf	Cancel use of buffer	85

Chapter 11

API Function Calls

This chapter consists of a description of each of the API function calls with short examples for assembler, C, Pascal and QuickBASIC. The Pascal interface requires version 5.0 of Turbo Pascal or greater. The BASIC interface requires Microsoft QuickBASIC version 4.50 or greater.

The following table shows the assembly language registers used by Reflection's Application Program Interface.

Table 3
Assembly Language Interface

AH =	0DE52H	
DX =	0	
CH =	0	(reserved for future use)
CL =	API function code	
ES	Used to pass parameters	
BX	Used to pass parameters	
SI	Used to pass parameters	
DI	Used to pass parameters	

INT 21H (use MS-DOS Interrupt 21H to access API)

api_assertdc

Tells Reflection to re-grab the serial hardware away from a foreground task that may have taken it. Reflection re-initializes datacomm back to its normal baud rate, parity, etc. Reflection will continually re-grab the hardware from another program if necessary.

INPUT

```
CX = 19
AX = DE52H
DX = 0
```

OUTPUT

Should not return an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,19
int  21h
```

QuickBASIC, C, Pascal Examples

For examples, see api_releasedc on page 137.

api_atrbscreen

Read text and color attributes from Reflection's background screen. Read can extend to function key display. Color attributes are IBM PC attribute bytes. Client program may read Reflection screen and re-display with same color attributes.

INPUT

```
CX = 31
AX = DE52H
DX = 0
BH = ROW  0 relative
BL = COL  0 relative
ES:DI    points to buffer to contain screen text
SI = number of characters to read.  Note that each
     character is 2 bytes with this function.  For each
     character, the first byte is the character from
     the IBM character set and the second byte is
     the attribute which determines how the byte is
     displayed.
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = 103H  read extends off of screen
```

Assembler and C

See example for api_screenread on page 141.

QuickBASIC

EXAMPLE—Read the Reflection screen and print characters with correct color attributes.

```
' $INCLUDE: 'api.inc'
' Allocate space for reading 80 columns by 25 rows by two bytes per char.
x$ = SPACE$(80 * 25 * 2)
'
' Ask for 80*25 characters starting at row 0, column 0
'
i% = api.atrbscreen%(x$, 0, 0, 80 * 25)
'
' Make default segment the segment of the video display (use B000 for mono)
'
DEF SEG = &HB800
FOR i = 1 TO 80 * 25 * 2
POKE i-1, ASC(MID$(x$, i, 1))
NEXT i
END
```

TURBO PASCAL

function api_atrbscreen(length,col,row:integer;var x:buffer):integer;

See example for api_screenread on page 141.

EXAMPLE—Read 100 characters off of screen and put ASCII bytes only into string variable. Read starting row 10 column 1.

```
program ReadScrnAttr (input,output);
{$V-}
uses  apiunit;
var
        scrn     : buffer;
        row      : integer;
        col      : integer;
        i        : integer;
        scrtext  : string[100];
begin
        row := 10;
        col := 1;
        i   := api_atrbscreen (100, col, row, scrn );
        i   := 0;
        while i < 100 do
             begin
```

```
                scrtext[i+1] := scrn[ i*2 ];
                i := i+1;
            end;
        scrtext[0] := chr(100);
        writeln( scrtext );
end.
```

api_block

Give Reflection some CPU time. This should be done if the API client program is waiting for Reflection to complete some queued commands or keystrokes but wants to maintain CPU control (api_block returns quickly).

INPUT

```
CX = 27
AX = DE52H
DX = 0
```

OUTPUT

```
AX  = 0   implies Reflection idle.  Command or key
          queues have been completed.

AX <> 0   implies Reflection is busy processing
          commands or queued keys.
```

Assembler

EXAMPLE—Give Reflection CPU time while waiting for user input.

```
Idle_Loop:
mov  ah,1           ; Check keyboard status
int  16h            ; BIOS keyboard interrupt
jnz  Key_waiting    ; Exit if user has typed key
mov  ax,0DE52H
xor  dx,dx
mov  cx,27          ; Do api_block
int  21h
or  ax,ax           ; Is Reflection finished?
jnz  idle_loop      ; No.  See if keyboard input
```

api_block

C

```
int      api_block();
```

EXAMPLE

```
main()
{
        /* queue two commands and block until complete */

        api_qcmd("wait 0:0:3");
        api_qcmd("display '^g'");
        while (api_block())
             if (kbhit())  /* If key has been pressed */
                break;  /* stop waiting */
}
```

QuickBASIC

```
DECLARE FUNCTION api.block% CDECL ( )
```

EXAMPLE—Give Reflection CPU time until file transfer completes or user hits a key.

```
' $INCLUDE: 'api.inc'
i% = api.qcmd("send bigfile to xyz")
DO
LOOP WHILE  inkey$="" and api.block <> 0
```

TURBO PASCAL

```
function api_block :integer;
```

For an example see api_offerbuf on page 121.

api_cancelbuf

This function causes Reflection to return to use of default keyboard and command buffers. Original buffer is reclaimed by API client program. Issue this call prior to terminating.

INPUT

```
CX = 25
AX = DE52H
DX = 0
```

OUTPUT

Should not return an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,25
int  21h
```

C

```
int     api_cancelbuf();
```

QuickBASIC

DECLARE FUNCTION api.cancelbuf% CDECL ( )

For an example see api_offerbuf on page 121.

TURBO PASCAL

function api_cancelbuf :integer;

For an example see api_offerbuf on page 121.

api_clrcmdq

Clears the command queue.

INPUT

```
CX = 17
AX = DE52H
DX = 0
```

OUTPUT

```
AX = 0
```

Should not encounter an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,17
int  21h
```

C

```
int        api_clrcmdq( );
```

QuickBASIC

DECLARE FUNCTION api.clrcmdq% CDECL ( )

EXAMPLE—Clear the command queue.

```
' $INCLUDE: 'API.INC'

i% = api.clrcmdq
```

TURBO PASCAL

function api_clrcmdq :integer;

EXAMPLE—Clear the command queue.

```
uses   apiunit;
var    i : integer;
begin
       i := api_clrcmdq;
end.
```

api_clrkeybd

Clear any remaining unprocessed keys out of the keyboard buffer.

INPUT

```
CX = 14
AX = DE52H
DX = 0
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = error code
```

Should not return an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,14
int  21h
```

C

```
int      api_clrkeybd( );
```

QuickBASIC

DECLARE FUNCTION api.clrkeybd% CDECL ( )

EXAMPLE—Clear the keyboard buffer.

```
' $INCLUDE: 'API.INC'
i% = api.clrkeybd
```

TURBO PASCAL

function api_clrkeybd :integer;

EXAMPLE

```
uses    apiunit;
var     i : integer;
begin
        i := api_clrkeybd
end.
```

api_cmdstatus

Returns amount of free space available in command queue buffer.

INPUT

```
CX = 15
AX = DE52H
DX = 0
```

OUTPUT

```
AX = amount of free space in bytes
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
mov     cx,15
int     21h
or      ax,ax
jz      No_free_space
```

C

int api_cmdstatus( ); returns free space in command queue

EXAMPLE

```
main()
{
        /* Queue a command if there is room in the buffer */

        char  *send_command = "SEND ABC TO XYZ ASCII DELETE";
        if (api_cmdstatus() >= strlen( send_command ) ) {
                api_qcmd( send_command );
                printf("Command Queued\n");
        }
        else
```

```
                printf("Queue full\n");
        }
```

QuickBASIC

DECLARE FUNCTION api.cmdstatus% CDECL ( )

EXAMPLE—Queue a command if there is room in the buffer.

```
' $INCLUDE: 'api.inc'

        cm$= "SEND ABC TO XYZ ASCII DELETE";
        if (api.cmdstatus > len(cm$)) then  ' if there is room then
            api.qcmd( cm$ )                 ' queue the command
            print "Command Queued"
        else
            print "Queue full"
        end if
```

TURBO PASCAL

function api_cmdstatus :integer;

EXAMPLE

```
uses    apiunit;
var     i : integer;
begin
        i := api_cmdstatus;
        writeln ( 'bytes free in command buffer = ', i );
end.
```

api_docommand

Perform a Reflection command synchronously. The call does not return to the caller until the command has been completed or an error is encountered.

During the wait for a Reflection command to complete, you cannot use the HOT-KEY. Reflection will just beep at you. "ALERT" messages will not blink on and off as normal. The calling program will be frozen until the command completes. Make sure that all commands issued will time out after a while, otherwise your API program can crash waiting for a condition that may never be met. See note on Synchronous File Transfers, i.e., use:

```
WAIT 0:0:10 FOR "^Q"
```

Do not use:

```
WAIT FOR "^Q"
```

INPUT

```
CX = 6
AX = DE52H
DX = 0
ES:BX points to buffer containing null-terminated
command string.
```

OUTPUT

No error:

```
AX = 0 Command performed satisfactorily
```

Error:

```
AX = Standard Reflection error code for command
     language.
AX = 102H Service not open - need api_startcommands()
```

Assembler

EXAMPLE

```
        mov     ax,0DE52H
        xor     dx,dx
        mov     cx,6        ; Function call 6
        push    ds
        pop     es
        ASSUME  ES:DATA
        mov     bx,offset Reflection_command
        int     21h
        or      ax,ax
        jnz     command_error

Reflection_command db "SEND ABC TO XYZ ASCII REC=80",0
```

C

```
int     api_docommand( command_string )
                        returns error code
                        returns non-zero if error
char    *command string;command_stringpoints to null-terminated
                        Reflection command
```

EXAMPLE

```
main()
{
  /* do a synch file transfer - print error code if failed */

  int     error;
  char    *command_string = "SEND ABC TO XYZ ASCII REC=80";
  api_startcommands();
  if (error = api_docommand(command_string))
          printf( "File transfer failed, error code = %d\n",error);
  api_endcommands();
}
```

api_docommand

QuickBASIC

DECLARE FUNCTION api.docommand% CDECL (SEG rcommand$)

EXAMPLE—Transfer a file.

```
' $include: 'api.inc'

i% = api.startcommands%
cm$ = "SEND ABC TO XYZ ASCII REC=80"
error% = api.docommand%(cm$)
if error% > 0 then print "File transfer failed error code = ";error%
i% = api.endcommands%
end
```

TURBO PASCAL

function api_docommand(var x: string) :integer;

EXAMPLE—Send a file.

```
{$V-}
uses     apiunit;
var
         i  : integer;
         cm : string[80];
begin
         cm := 'SEND ABC TO XYZ ASCII REC=80';
         if api_startcommands <> 0 then
                writeln ( 'Reflection busy' );
         if api_docommand( cm ) <> 0 then
                writeln ( 'error transferring file' );
         i := api_endcommands;
end.
```

api_endcommands

This function call stops a series of synchronous commands, and turns datacomm back on.

INPUT

```
CX = 10
AX = DE52H
DX = 0
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = error code
```

Should not return error if Reflection is present.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,10
int  21h
or   ax,ax
jnz  Serious_error
```

C

```
int       api_endcommands( );
```

For a C programming example, see api_startcommands on page 154.

api_endcommands

QuickBASIC

```
DECLARE FUNCTION api.endcommands% CDECL ( )
```

For an example see api_startcommands on page 154.

TURBO PASCAL

```
function api_endcommands :integer;
```

For an example see api_startcommands on page 154.

api_found

Return the value of the command language FOUND boolean.

INPUT

```
CX = 9
AX = DE52H
DX = 0
```

OUTPUT

```
AX = value of FOUND boolean
AX =  0   Not found
AX <> 0   Found
```

Assembler

EXAMPLE

```
mov   ax,0DE52H
xor   dx,dx
mov   cx,9
int   21h                        ; call API
or    ax,ax
jz    String_Not_Found
```

C

int api_found( );   returns value of FOUND boolean

EXAMPLE

```
main()
{
  /* return the state of the found boolean */

  api_startcommands();
  api_docommand("transmit  'john^m'");          /* send username */
  api_docommand("wait 0:0:8 for 'Password:'");
  if (!api_found()) {
        printf("Timed out waiting for password\n");
```

```
          printf("Try again ?:");
    }
    api_endcommands();
}
```

QuickBASIC

DECLARE FUNCTION api.found% CDECL ( )

EXAMPLE—Print a message if successful connect to remote modem.

```
' $INCLUDE: 'api.inc'
i% = api.startcommands
i% = api.docommand("transmit 'ATDT 555-1212^m'")
i% = api.docommand("WAIT 0:0:45 for 'CONNECT'")
if api.found then print "Connected to REMOTE MODEM"
i% = api.endcommands
end
```

TURBO PASCAL

function api_found :integer;

EXAMPLE—Print a message if successful connect to remote modem.

```
{$V-}
uses        apiunit;
var         cm    : string[80];
            i     : integer;
begin
            i  := api_startcommands;
            cm := 'transmit "ATDT 555-1212^m"';
            i  := api_docommand( cm );
            cm := 'WAIT 0:0:30 for "CONNECT"';
            i  := api_docommand( cm );
            if api_found <> 0 then
                    writeln ( 'Connected to REMOTE MODEM' )
            else
                    writeln ( 'No connection' );
            i  := api_endcommands;
end.
```

api_getfkey

Return the current setting of a Reflection softkey to calling program. This function returns value of R2 and R4 softkeys, R2 and R4 user-defined keys (UDKs) and R1 and R7 HP user keys.

INPUT

```
CX = 22
AX = DE52H
DX = 0
ES:BX point to buffer to return key.
SI = Key number (1 relative)
    To return R2/R4 user-defined keys  (UDKs)
  UDK # 6  = keynumber 17
  ...     ...
  UDK # 20 = keynumber 34
```

OUTPUT

No error:

```
AX = 0
key record copied to buffer
```

Error:

```
AX = 105H Badkey
```

HP User Key Label Structure: R1/R7

```
struct ukey {
  char   unsigned  ukey_attr;   /* 0=normal, 1=local only, 2=transmit only*/
  char   unsigned  ukey_lablen; /* length of the label  */
  char   unsigned  ukey_deflen; /* length of the definition string */
  int              ukey_reserved;
  char             ukey_text[160]; /*label string followed by definition*/
};
```

api_getfkey

DEC Softkey Structure: R2/R4

```
struct softkey {
   char     unsigned   action;  /* 0 = normal, 1 = local only */
   char     unsigned   label_length;
   char     label_text[8];
   char     unsigned   defn_length;
   char     defn_text[80];
};
```

DEC UDK Structure: R2/R4

```
struct udk {
   char  unsigned          udk_length;
   char                    udk_text[255];
};
```

Assembler

EXAMPLE

```
                                     ; Get HP softkey # 3
mov      ax,0DE52H
xor      dx,dx
push     ds
pop      es
mov      si,3                        ; User key # 3
assume   es:data
mov      bx,offset ukey_buffer       ; where to put the ukey info
mov      cx,22                       ; get user key
int      21h
or       ax,ax                       ; Error?
jnz      getkey_error                ;  YES


key_buffer label byte
ukey_attr   db ?           ; 0 = normal, 1 = local, 2 = transmit only
ukey_lablen db ?           ; length of label text
ukey_deflen db ?           ; length of definition string
reserved    dw ?
ukey_text   db 160 dup (0); label text and definition start
```

C

```
int     api_getfkey( key_buffer, keynumber );
struct  ukey                    *key_buffer;
int     keynumber;
```

EXAMPLE

```
main()
        /* Retrieve a function key and print the label */



{
struct  ukey   key_buffer;

int     i;
int     keynumber=3;
struct  ukey  *kptr;
        kptr = &key_buffer;
        api_getfkey( kptr, keynumber );
        i = kptr->ukey_lablen;
        kptr->ukey_text[i]='\0';  /* null-terminate after label */
        printf("key %d label '%s'\n", keynumber, kptr->ukey_text );
}
```

QuickBASIC

```
DECLARE FUNCTION api.getfkey% CDECL
        (SEG keybuff AS ukeytype, BYVAL keyno%)
```

EXAMPLE—Print the definition part of HP function key #3.

```
' $INCLUDE: 'api.inc'

COMMON x AS ukeytype
       i% = api.getfkey(x, 3)
       y$ = x.ukeytext
       PRINT MID$(y$, ASC(x.ukeylablen) + 1)
       END
```

api_getfkey

TURBO PASCAL

function api_getfkey(k : integer; var fkey: ukeytype) :integer;

TERMS

k

Function key number

fkey

User key record–see apiunit.pas for declaration

EXAMPLE—Print the definition part of HP function key #3.

```
uses    apiunit;
var     x : ukeytype;       (* function key record *)
        i : integer;
        n : integer;        (* label length *)
        keydefn  : string[80];
begin
        i := api_getfkey( 3, x );
        n := ord( x.ukeylablen );
        i := 0;
        keydefn[0] := x.ukeydeflen;  (* definition length *)
        while i < ord( x.ukeydeflen ) do
           begin
              keydefn[i+1] := x.ukeytext[n + i ];
              i := i + 1;
           end;
        writeln ( 'definition of function key 3 = ',keydefn );
end.
```

api_getinfo

Gets information from Reflection. This call returns information of a relatively non-volatile nature, i.e., information that should not change on a second by second basis.

INPUT

```
CX = 3
AX = DE52H
DX = 0
ES:BX points to user buffer
SI = # of words of information to be written to user
buffer. If SI exceeds structure size (25) remaining space
is zeroed.
```

OUTPUT

No error:

```
AX = 0
Buffer initialized
```

Error:

```
AX = error code
```

Should not return error if Reflection present.

Structure Definition

```
struct api_infostruc {
   int        apiinfo_apiversion;
   int        apiinfo_function_key_mode;
   int        apiinfo_local_echo;
   int        apiinfo_remote_mode;
   int        apiinfo_caps_lock;
   int        apiinfo_display_functions;
   int        apiinfo_auto_linefeed;
   int        apiinfo_right_margin;
   int        apiinfo_phys_screen_width;
```

```
            int       apiinfo_memory_response;       /* HP */
            int       apiinfo_xmit_functions;        /* HP */
            int       apiinfo_spow_strap;            /* HP */
            int       apiinfo_inheolwrp;             /* HP */
            int       apiinfo_line_page_mode;        /* HP */
            int       apiinfo_inhhndshk;             /* HP */
            int       apiinfo_inhdc2;                /* HP */
            int       apiinfo_block_mode;            /* HP */
            int       apiinfo_format_mode;           /* HP */
            int       apiinfo_memory_lock;           /* HP */
            int       apiinfo_type_ahead;            /* HP */
            int       apiinfo_normal_cursor_key_mode;/* DEC */
            int       apiinfo_numeric_keypad_mode;   /* DEC */
            int       apiinfo_multipage_mode;        /* DEC */
            int       apiinfo_user_features_locked;  /* DEC */
            int       apiinfo_udks_locked;           /* DEC */
        };
```

apiinfo_apiversion — Version number of API. High byte = major version. Low byte is minor version.

apiinfo_function_key_mode — Number identifying current set of function keys displayed at bottom of Reflection screen.

Identifying Numbers

The identifying numbers for each function key, grouped by product, are listed below.

R2/R4 Keys

Function Key Set	Identifying Number
MAINMENU	0
SOFTKEYS	1
VT102KEYS	2
NOKEYS	3
CONFIGKEYS	4
DEVICECTLKEYS	5
DEVICEMODESKEYS	6
TODEVICEKEYS	7
FILEXFERKEYS	8

TABKEYS	9
TERMINALKEYS	10
GRAPHICSKEYS	11

R1, R7, R1V, & R7V Keys

Function Key Set	Identifying Number
MODESKEYS	0
USERKEYS	1
SYSTEMKEYS	2
CONFIGKEYS	3
MARGINTABKEYS	4
ENHANCEKEYS	5
DEVICECTLKEYS	6
DEVICEMODESKEYS	7
USERMENUKEYS	8
FILEXFERKEYS	9
TODEVICEKEYS	10
VT102KEYS	11*
VMODESKEYS	12*
VMAINMENU	13*
VTABKEYS	14*

∅ and 1 Settings

Command	Set to ∅	Set to 1
apiinfo_local_echo	OFF	ON
apiinfo_remote_mode	OFF	ON
apiinfo_caps_lock	OFF	ON
apiinfo_display_functions	OFF	ON
apiinfo_auto_linefeed	OFF	ON
apiinfo_right_margin	logical right hand margin of screen (0 relative)	
apiinfo_phys_screen_width	actual physical width of screen (1 relative)	
apiinfo_memory_response	4K	8K
	2 = 12K	3 = 15K
apiinfo_xmit_functions	OFF	ON
apiinfo_spow_strap	OFF	ON
apiinfo_inheolwrp	OFF	ON
apiinfo_line_page_mode	LINE	PAGE
apiinfo_inhhndshk	OFF	ON

* Settings when running VT emulation (R1V/R7V).

apiinfo_inhdc2	OFF	ON
apiinfo_block_mode	OFF	ON
apiinfo_format_mode	OFF	ON
apiinfo_memory_lock	OFF	ON
apiinfo_type_ahead	OFF	ON
apiinfo_normal_cursor_key_mode	APPL	NORMAL
apiinfo_numeric_keypad_mode	APPL	NORMAL
apiinfo_multipage_mode	OFF	ON
apiinfo_user_features_locked	UNLOCKED	LOCKED
apiinfo_udks_locked	UNLOCKED	LOCKED

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset infostructure
mov     cx,3
int     21h
or      ax,ax
jnz      info_error
```

C

int api_getinfo(infobuffer, count);

struct api_infostruc *infobuffer; — infobuffer points to structure defined above

int count; — infobuffer size in words

EXAMPLE

```
main()
{
      /* find the physical screen width */

  struct  api_infostruc infobuffer;
```

```
   struct  api_infostruc *sptr;
   sptr    = &infobuffer;
   api_getinfo( sptr , 25);
  printf( "Physical Screen Width = %d\n",sptr->apiinfo_phys_screen_width);
 }
```

QuickBASIC

DECLARE FUNCTION api.getinfo% CDECL
        (SEG buffer AS RINFO, BYVAL rvarlen%)

TERMS

rvarlen%
    Size of buffer

See api.inc for RINFO user-defined type definition.

EXAMPLE—Get Reflection physical screen width.

```
' $include: 'api.inc'
'
'       See API.INC for description of user-defined type RINFO
'
COMMON X AS RINFO

i% = api.getinfo( X, 25%)
print "Reflection screen width is "; x.rphysscreenwidth
end
```

TURBO PASCAL

function api_getinfo(i:integer; var x:info_array) :integer;

TERMS

i
    Size of info_array buffer

info_array
    See apiunit.pas for info_array type declaration

EXAMPLE—Get screen width.

```
uses  apiunit;
var
      info  : info_array;
      i     : integer;

begin
      if api_getinfo( 25, info )  > 0 then exit;
      writeln( 'screen width is ', info[8] );
end.
```

api_getstatus

Return status of volatile information from Reflection. This information is time critical.

INPUT

```
CX = 4
AX = DE52H
DX = 0
ES:BX point to structure
SI   specifies number of words (12) to be returned to user
     buffer.  If SI greater than structure size, excess
     buffer area will be zeroed.
```

OUTPUT

No error:

```
AX = 0
```

Error:

```
AX = error code
```

Should not return error if Reflection present.

Structure Definition

```
struct api_statstruc {
   int        apistat_pagetop_row;
   int        apistat_cursor_row;
   int        apistat_cursor_col;
   int        apistat_left_border;
   int        apistat_cursor_physrow;
   int        apistat_cursor_physcol;
   int        apistat_kb_lock_sw;
   int        apistat_batch_flag;
   int        apistat_datacomm_error_flag;   /* HP */
   int        apistat_host_prompt_received;  /* HP */
   int        apistat_xfer_pending_sw;       /* HP */
};
```

apistat_pagetop_row
Ø relative row number of display memory for top of screen, i.e., if 100 lines of text have scrolled off the top of the screen (and are still in display memory), the top of the screen would display row 101 or pagetop_row 100.

apistat_cursor_row
Logical Ø relative cursor row from beginning of display memory.

apistat_cursor_col
Logical Ø relative cursor column from left hand margin of screen.

apistat_left_border
Column position of left border of screen (true left margin may have been scrolled off of screen).

apistat_cursor_physrow
Actual physical row where cursor is located (Ø relative).

apistat_cursor_physcol
Actual physical column where cursor is located (Ø relative).

apistat_kb_lock_sw
Keyboard lock status:
Ø = No lock   <>Ø = Keyboard locked.

apistat_batch_flag
Ø = Reflection IDLE   <>Ø= BUSY.

apistat_datacomm_error_flag
1 = DATACOMM ERROR has occurred since last primary status request.

apistat_host_prompt_received
1 = Host prompt has been received.
Ø = Waiting for host prompt.

apistat_xfer_pending_sw
<>Ø = Transfer of data has been requested from host. Waiting to receive prompt before starting transfer.
Ø = No transfer pending.

Assembler

EXAMPLE

```
mov         ax,0DE52H
xor         dx,dx
push        ds
```

```
        pop         es
        ASSUME      ES:DATA
        mov         bx,offset status_struc
        mov         cx,4
        int         21h
        or          ax,ax
        jnz         info_error

status_struc  dw  22 dup (?)  ; 44 byte buffer to receive data
```

C

```
int         api_getstatus(    statusbuf, siz )
struct api_statstruc *statusbuf;  statusbuf is pointer to structure
int         siz;
```

EXAMPLE

```
main()
{

    /* find which row the cursor is on */

    struct  api_statstruc  buffer;
    struct  api_statstruc  *sptr;
    sptr = &buffer;
    api_getstatus(sptr, 11);  /* returns 11 words of status info */
    printf( "cursor row = %d",sptr->apistat_cursor_row);
}
```

QuickBASIC

```
DECLARE FUNCTION api.getstatus% CDECL
        (SEG buffer AS RSTAT, BYVAL rvarlen%)
```

TERMS

rvarlen%
Size of buffer

See api.inc for RSTAT user-defined type definition.

api_getstatus

EXAMPLE—Get physical cursor and row position.

```
' $include: 'api.inc'
'
'       See API.INC for description of user-defined type RSTAT
'
COMMON Y AS RSTAT

i% = api.getstatus( Y, 11%)
print "Cursor at ROW ";y.Rcursorphysrow ;" COL ";y.Rcursorphyscol
end
```

TURBO PASCAL

function api_getstatus(i:integer; var x:stat_array) :integer;

TERMS

i
Size of stat_array buffer

stat_array
See apiunit.pas for stat_array type declaration

EXAMPLE

```
uses apiunit;
var  status  : stat_array;

begin
     if api_getstatus(11, status )  > 0 then exit;
     writeln ( 'cursor row is ', status[1] , ' column ' ,status[2] );
end.
```

api_getvar

Return contents of variable to user buffer. Note that a Reflection variable may contain characters of any value including nulls.

This command should only be executed as part of a synchronous command sequence, otherwise it can return random values. If it interrupts Reflection's changing of one variable to another, a false result could occur.

Note: Variables cannot always be null-terminated in this way since it is possible for them to contain nulls themselves.

INPUT

```
CX = 7
AX = DE52H
DX = 0
ES:BX points to 80 byte buffer
SI = variable number
```

OUTPUT

No error:

```
AX = 0
SI = # length of variable
```

Error:

```
AX = 10AH bad variable no
```

Assembler

EXAMPLE—Get variable V8 and null-terminate.

```
mov     ax,0DE52H
xor     dx,dx
mov     cx,7                ; Function GetVar
mov     si,8                ; get variable V8
push    ds
pop     es
```

```
ASSUME    ES:DATA
mov       bx,offset Variable_Buffer
int       21h
                                  ; SI is length of variable V0
                                  ; put NULL at end of string
mov       byte ptr [variable_length+SI],0
or        ax,ax                   ; was there an error
jnz       Bad_variable_number     ; yes.

Variable_Buffer db 80 dup (?)
```

C

```
int       api_getvar( var_buffer, varno, varlength )
                         returns non-zero if error
                         returns length of variable
char      *var_buffer;   buffer for 80 byte variable
int       varno;         variable number 0 - 799
int       &varlength;    returned length of variable
```

EXAMPLE

```
main()
{

     /*  Find out if a file is present on the HP3000 */

     int  length;
     char  varbuf[80];
     api_startcommands();
     printf("Enter filename :");
     gets(varbuf);
     length = strlen( varbuf );
     api_setvar(varbuf,1, length);
     api_docommand("transmit 'listf $1^m'");
     api_docommand("readhost 0:0:5 v2");
     api_docommand("readhost 0:0:5 v2");  /* get HP response */
     api_docommand("wait 0:0:8 for '^q'");
     api_getvar( varbuf, 2, &length );    /* get v2 */
     varbuf[length] = '0';              /* null-terminate v2 */
```

```
          if (strstr( varbuf, "CIERR" ))
                printf( "File Not Found on Host\n");
          api_endcommands();
     }
```

QuickBASIC

DECLARE FUNCTION api.getvar% CDECL
(SEG var$, BYVAL varnumber%, SEG rvarlen%)

TERMS

var$
String first initialized to at least 80 characters in size for receiving variable data

varnumber%
Variable number

rvarlen%
Length of variable

EXAMPLE—Find out if a file is present on a VAX.

```
' $INCLUDE: 'api.inc'
LINE INPUT "Enter vax filename"; f$
if f$< "!" then end
i% = api.startcommands
j$ = "transmit 'dir " + f$ + CHR$(13) + "'"
i% = api.docommand(j$)
i% = api.docommand("readhost 0:0:5 v2")
i% = api.docommand("readhost 0:0:5 v2")
i% = api.docommand("readhost 0:0:5 v2")
i% = api.docommand("wait 0:0:8 for '^m$'")
varbuf$ = SPACE$(80)
i% = api.getvar( varbuf$, 2, l% )
varbuf$ = MID$( varbuf$, 1, l% )
IF INSTR(varbuf$, "%DIRECT-W-NOFILES") > 0 THEN
     PRINT "File "; x$; "not found on vax"
ELSE
     PRINT "File "; x$; " on VAX"
END IF
```

```
i% = api.endcommands
END
```

TURBO PASCAL

function api_getvar(i :integer; var x: string) :integer;

TERMS

i Variable number

x String variable to return data to

EXAMPLE—Find out if a file is present on a VAX.

```
uses apiunit;
{$V-}
var      fname   : string[80];
         cm      : string[80];
         i       : integer;
         vlen    : integer;
begin
         write ( 'Enter vax filename' );
         readln( fname );
         if fname < '!' then exit;
         i := api_startcommands;
         cm := 'transmit "dir ' + fname + '^m"';
         i := api_docommand(cm);
         cm := 'readhost 0:0:5 v2';
         i := api_docommand( cm );
         i := api_docommand( cm );
         i := api_docommand( cm );
         cm := 'wait 0:0:0 for "^m$"';
         i := api_docommand( cm );
         i := api_getvar( 2, cm );
         if pos( '%DIRECT-W-NOFILES', cm ) <> 0 then
               writeln ('File ', fname, ' not found on VAX' )
         else
               writeln ('File ', fname, ' on VAX' );
end.
```

api_instchk

See if the API support code has been loaded via the /W switch. Determines Reflection product, version, and serial numbers.

The remaining 17 bytes of the buffer are reserved for future use.

INPUT

```
CX = 0
AX = DE52H
DX = 0
ES:BX point to 32 byte buffer to receive serial number
```

OUTPUT

No error:

```
AX =  0
Null-terminated Reflection 14-byte serial number
copied to buffer.
See Command Language manual for serial number format.
```

Error:

```
AX <> 0
```

Assembler

EXAMPLE

```
mov      ax,0DE52H
xor      dx,dx
mov      cx,0
push     ds
pop      es
ASSUME   ES:DATA
mov      bx,offset serialno_buffer   ;ES:BX -> buffer
int      21h
or       ax,ax
jnz      not_installed  ; Error if AX not zero


serialno_buffer db  32 dup (0)
```

api_instchk

C

int api_instchk(serialbuf) returns Ø if Reflection present
char *serialbuf; serialbuf points to 32 byte buffer

EXAMPLE

```
char  serialbuf[32];
  if ( api_instchk( serialbuf )) {
     printf( "API support not present restart with /W switch\n");
     exit();
  }
  else
     printf( "Reflection serial number %s\n", serialbuf );
```

QuickBASIC

DECLARE FUNCTION api.instchk% CDECL (SEG serno$)
set serno$ to spaces prior to call
serno$= spaces$(32)

EXAMPLE—Get the Reflection serial number if API is present.

```
' $INCLUDE: 'API.INC'

  serno$ = space$(32)
  if api.rcheck%(serno$) <> 0 then
      print "API support not present, restart with /W switch")
  end if
```

TURBO PASCAL

function api_instchk(var s: string) :integer;

EXAMPLE

```
uses    apiunit;
var     serno  : string[32];

begin
        if api_instchk( serno ) ) 0  then exit;
        writeln( 'serial number is ', serno )
end.
```

api_keystatus

Returns amount of free space left in keyboard queue in keys. Each key actually takes two bytes of space in the queue.

INPUT

```
CX = 12
AX = DE52H
DX = 0
```

OUTPUT

```
AX = amount of free space in units of keys
```

Assembler

EXAMPLE

```
mov   ax,0DE52H
xor   dx,dx
mov   cx,12
int   21h
or    ax,ax
jz    No_free_space_remaining
```

C

int api_keystatus; returns free space in key units

EXAMPLE

```
main()
{
if (!api_keystatus)
        printf( "API Key buffer full\n");

}
```

api_keystatus

QuickBASIC

DECLARE FUNCTION api.keystatus% CDECL ( )

EXAMPLE—See if any room left to queue keys.

```
' $INCLUDE: 'api.inc'
if api.keystatus = 0 then print "API key buffer full"
```

TURBO PASCAL

function api_keystatus :integer;

EXAMPLE

```
uses     apiunit;
var      i : integer;
begin
         if api_keystatus <> 0 then
             writeln( 'key buffer full');
end.
```

api_offerbuf

This function provides larger buffers for command or keyboard queues than the default API buffers. The default buffer size is approximately 190 bytes for the command buffer and 32 keys for the keyboard buffer.

Make sure that the current queue is empty before offering a new buffer, otherwise any queued keys or commands are lost. Each offer cancels the previous one. Do not modify the offered buffer space until it has been canceled (see below). Note that the pointer passed to API is a far pointer (32 bit address).

INPUT

```
CX = 24
AX = DE52H
DX = 0
ES:BX points to buffer
SI    size of buffer
DI    buffer type
      0 = keyboard buffer
      1 = command queue buffer
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = 103H  Bad buffer type
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
push    ds
pop     es
```

```
ASSUME  ES:DATA
mov     bx,offset command_buffer
mov     di,1       ; 1 <==> buffer for command queue
mov     si,CBUFFER_LEN    ; length of buffer
mov     cx,24
int     21h

command_buffer  db      2000 dup (0)
CBUFFER_LEN     EQU     $ - command_buffer
```

C

```
int     api_offerbuf( command_queue, bufsize, type );
char    far *command_queue;
int     bufsize;
int     type;
```

EXAMPLE

```
            Offer a 2000 byte command buffer
    #define BUFSIZE  2000
    #define COMMAND_TYPE  1
    char far command_queue[BUFSIZE];

    api_offerbuf(command_queue, BUFSIZE, COMMAND_TYPE );
```

QuickBASIC

```
DECLARE FUNCTION api.offerbuf% CDECL
        (SEG buffer AS bufftype, BYVAL rvarlen%, BYVAL typ%)
```

TERMS

buffer =

Buffer to be used for queuing keys or commands– see api.inc for user-defined type definition

rvarlen%

Size of buffer

EXAMPLE—Set up 512 byte command buffer within QuickBASIC's data segment and queue some commands.

```
' $include: 'api.inc'

COMMON cmdbuf as bufftype  ' see api.inc for bufftype declaration
i% = api.offerbuf( cmdbuf, 512, 1% )
i% = api.qcmd("dir *.*")
i% = api.qcmd("display '^g'")
i% = api.qcmd("wait 0:0:10")
i% = api.wait
i% = api.cancelbuf
end
```

TURBO PASCAL

function api_offerbuf(typ,buflen:integer;varx : buffer):integer;

TERMS

type
0 or 1 for keys or commands

buflen
Size of buffer

x
Buffer–see apiunit.pas for declaration

EXAMPLE—Offer a Keyboard Queue buffer from Pascal's data space.

```
{$V-}
uses  apiunit, crt;
const  maxlength = 256;
var   keys  : string[250];
      i    : integer;
      keybuf   : buffer;
begin
   i := api_offerbuf( 0, 256, keybuf );
   keys := '"enter these keys and hit return" return ';
   i := api_qkeys( keys );
   repeat
   until keypressed or (api_block = 0);
   i := api_cancelbuf;  (* cancel buffer before terminating program *)
end.
```

api_popup

Pop Reflection into the foreground. API client application will be frozen until user presses hot-key or a queued "BACKGROUND" command is processed.

INPUT

```
CX = 20
AX = DE52H
DX = 0
```

OUTPUT

Should not return an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,20
int  21h  ; This call freezes API Client program
```

C

int   api_popup();

EXAMPLE

```
api_popup();
printf("Back from POP COMMAND\n");
```

QuickBASIC

DECLARE FUNCTION api.popup% CDECL ( )

EXAMPLE—Pop up Reflection.

```
' $INCLUDE: 'API.INC'
i% = api.popup
print "BACK from Reflection"
```

TURBO PASCAL

function api_popup :integer;

EXAMPLE—Pop up Reflection.

```
uses   apiunit;
var    i : integer;
begin
       i := api_popup;
end.
```

api_qcmd

Queue a new command in the command queue. This call returns immediately to the caller and the command is processed asynchronously via background multitasking.

INPUT

```
CX = 16
AX = DE52H
DX = 0
ES:BX point to null-terminated command
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = 104H Queue full
   = 106H Bad Length
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
mov     cx,16
push    ds
pop     es
ASSUME  ES:DATA
mov     bx,offset command_string
int     21h
or      ax,ax
jnz     Command_Queue_error


Command_string  db  "Send ABC to XYZ ASCII DELETE",0
```

C

```
int      api_qcmd( command_string );  returns error-code
char     *command_string;
```

For a C programming example, see the api_cmdstatus example on page 90.

QuickBASIC

DECLARE FUNCTION api.qcmd% CDECL (SEG rcommand$)

EXAMPLE—Queue a command and report if there was an error.

```
' $INCLUDE: 'API.INC'

cm$ = "SEND ABC TO DEF ASCII REC = 256"
if api.qcmd( cm$ ) then print "Error queuing command"
```

TURBO PASCAL

function api_qcmd(var cmd : string) :integer;

TERMS

cmd
String containing command

EXAMPLE

```
uses    apiunit;
        var  cm : string[80];
        i  : integer;
begin
        cm := 'SEND ABC TO DEF ASCII REC = 256';
        if api_qcmd( cm ) <> 0 then
             writeln( 'error occurred while queuing command');
end.
```

api_qkeys

This function call queues Reflection keystrokes and returns immediately to the caller. Since this is an asynchronous (queued) call, the caller will not know when the keys are sent. The keys are entered into Reflection's keyboard buffer exactly as though they had been typed, except no remapping takes place.

There are two kinds of keys on your keyboard, those that cause standard ASCII characters to be transmitted and those that perform control functions like resetting an internal modem, bringing up a configuration screen, or displaying the softkeys. The api_qkeys function differentiates among these sets of keys. Keys that cause ASCII values to be transmitted or entered are passed by enclosing the string of keys in quotes—either single or double.

```
"read mail"
"don't exit"
'he said "wait for me!" '
```

C uses quotes to delimit strings, but the quotes are not part of the string. For this reason, you may have to use single quotes within C string constants such as:

```
char  *keyentry = "'find employee miller'"
```

Control keys such as RETURN and VT-ENTER are keywords and must be passed to the api function unquoted. These keyword definitions are the same as those used in Reflection's keyboard mapping utility (KEYMAP or KEYCOMP). Here's an example of entering a string of keys followed by a carriage return:

```
api_qkeys( "'MyPassword'  RETURN ");
```

Notice that the password (MyPassword) is a quoted string, but the keyword RETURN is not quoted. The keynames are documented in the *Keyboard Remapping* section of the Reflection *Technical Reference Manual.*

INPUT

```
CX = 13
AX = DE52H
DX = 0
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = 104H Queue full
AX = 105H Bad Key
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
push    ds
pop     es
mov     bx,offset key_buffer
mov     cx,13
int     21h
or      ax,ax
jnz     key_queue_error


key_buffer  db     '"Hello user.acct' return",0
```

C

```
int     api_qkeys( keybuffer );     returns non-zero if error
char    *keybuffer;
```

EXAMPLE

```
main()
{

/* Queue some keys if there is room in the key queue */
char  *keybuffer = '"Hello user.acct' return";
if ( strlen( keybuffer ) <= api_keystatus())
        api_qkeys( keybuffer );
else
        printf("No room in Reflection Key Queue.\n");
while(!api_block())
        if (kbhit())
              break;
}
```

QuickBASIC

DECLARE FUNCTION api.qkeys% CDECL (SEG keys$)

TERMS

key$
String containing keystrokes to queue

EXAMPLE—Queue keys to log on to an HP 3000.

```
' $INCLUDE: 'api.inc'
i% = api.qkeys( "'hello user.acct' return")
```

TURBO PASCAL

function api_qkeys(var keys: string) :integer;

TERMS

keys
String containing keystrokes

EXAMPLE

```
{$V-}
uses     apiunit;
var      keys   : string[80];
         i      : integer;
begin
         keys := '"hello user.acct" return';
         i := api_qkeys( keys );
end.
```

api_rcheck

See if a copy of Reflection is present in the background.

INPUT

```
AX = 0DE57H
```

OUTPUT

```
AX = "RQ"        - implies Reflection present
AX = other value - implies Reflection not present
```

This function has existed in all versions of Reflection from 2.00 on. It departs from the standard API call format (AX = DE57H instead of DE52H). CX N/A.

This function may be used to detect the presence of any version of Reflection including those prior to version 3.4. It does not imply that the API code is present, only that a copy of Reflection is present in background. This will enable you to tell a user that while Reflection may be installed, either the /W switch was not used or it is a pre-3.40 version.

Assembler

EXAMPLE

```
mov    ah,0DE57H
int    21h
cmp    ax, "RQ"              ; Does AX have correct signature?
jz     Reflection_present    ; Yes - Reflection is present
mov    ah,9                  ; No - print error message and exit
mov    dx,offset error_message
int    21h
mov    ah,4ch
int    21h
error_message db  "Reflection not installed",0dh,0ah,"$"
```

api_rcheck

C

int api_rcheck( ) returns 0 if Reflection is present

EXAMPLE

```
main()
{
  /* See if Reflection is in background */

  if (api_rcheck())
    printf("Reflection not installed\n");
  else
    printf("Reflection in background\n");
}
```

QuickBASIC

DECLARE FUNCTION api.rcheck% CDECL ( )

EXAMPLE—See if Reflection is installed.

```
' $INCLUDE: 'API.INC'

if api.rcheck <> 0 then
        print "Reflection not installed"
else
        print "Reflection in background"
end if
```

TURBO PASCAL

function api_rcheck : integer;

EXAMPLE—See if Reflection is installed.

```
uses  apiunit;
var

begin
      if api_rcheck <> 0 then
             writeln('Reflection not installed')
      else
             writeln('Reflection in background');
```

api_rdchar

Directly read asynchronous datacomm from Reflection's receive buffer. Reflection will not read the characters. api_startcommands( ) must be invoked first to stop Reflection from reading its incoming datacomm.

It is the API client program's responsibility to keep up with the incoming data. Make sure that the appropriate flow control is set. Recommend SET RECEIVE-PACING XON/XOFF in most cases. Character translation takes place from the host character set to the PC character set unless SET DISABLE-TRANSLATION YES is in force.

INPUT

```
CX = 33
AX = DE52H
DX = 0
ES:BX points to  2 byte buffer for storing character.
(Null is second byte)
```

OUTPUT

No error:

```
AX =  0  character read
```

Error:

```
AX <> 0  no characters available
```

Assembler

EXAMPLE

```
mov         ax,0DE52H
xor         dx,dx
push        ds
pop         es
ASSUME      ES:DATA
mov         bx,offset receive_buf
mov         cx,33
```

api_rdchar

```
int         21h
or          ax,ax
jz          received_a_character


receive_buf  db  ?,0
```

C

int   api_rdchar(string) read character and store in *string
char   *string;   returns NZ if character not available

EXAMPLE

```
main()
{

    /* transmit some characters and read their echo */

    char  one_byte[2];
    char  temp[33];
    int  k;
    if (api_startcommands())   {    /* tell Reflect not to read */
      printf("can't start\n");    /* incoming data */
      exit();
    }
    k=0;
    api_docommand("transmit 'Read echo from these characters^m'");
    while(k < 33 )  {
           if (kbhit())
                break;
           if (!api_rdchar(one_byte))
                temp[k++]=*one_byte;
     }
     temp[k]='0';
     printf("%s\n",temp);
     api_endcommands();
}
```

QuickBASIC

DECLARE FUNCTION api.rdchar% CDECL (SEG char$)

TERMS

char$

Must be initialized to a length of 1 prior to call–if found, char$ will contain character that has been read and api.rdchar% = 0

EXAMPLE—Send a character and then read the echo coming back.

```
' $INCLUDE: 'api.inc'
ch$ = space$(1)
i% = api.startcommands
'
' send a character - depending on host, will probably be echoed back
'
i% = api.writeasync( asc("a"))
'
' rdloop waiting for character to come back or user to hit key
'
do
loop while api.rdchar( ch$ ) <>0 and inkey$=""
if ch$ = "a" then print "Successfully transmitted and received character"
i% = api.endcommands
end
```

TURBO PASCAL

function api_rdchar(var x:integer) :integer;

TERMS

x

Integer that will contain character after successful read (api_rdchar returns 0)

EXAMPLE—Write a string and read back the echo and print it out. Requires connection to full duplex host with local echo off.

```
{$V-}
uses  apiunit;
var
```

```
       j  : integer;
       k  : integer;
       i   : integer;
       srctext  : string[255];
       destext  : string[255];
       c   : integer;
begin
      srctext := 'send this string';
      destext := '';
      i := api_startcommands;
      k := 0;
      while k < length( srctext ) do
         begin
                   if api_xmitstatus > 0  then
                         begin
                         j := api_writeasync( ord( srctext[k+1] ));
                         k := succ(k);
                         end;
           end;
      k := 0;
      while k < length( srctext ) do
         if api_Rdchar  (c) = 0  then
            begin
            destext[k+1] := chr(c);
            k := succ(k);
            end;
      destext[0] := chr(k);
      j := api_endcommands;
      writeln( 'received data = ', destext );
end.
```

api_releasedc

The purpose of this command is to stop stealing datacomm hardware away from the foreground process. Normally if Reflection is running a command file or processing a command, it will steal the datacomm hardware (COM1, COM2, etc.) away from a foreground process that may have grabbed it. This call allows a foreground task to initialize the datacomm hardware and do direct datacomm I/O without interference from Reflection in background. Reflection will not be able to do datacomm.

This call will not restore datacomm to the foreground program if it has already been taken away. The foreground program will have to re-initialize the datacomm hardware. api_releasedc should only be required when the hardware serial ports are being used by both Reflection and a foreground program.

INPUT

```
CX = 18
AX = DE52H
DX = 0
```

OUTPUT

Should not return an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,18
int  21h
```

C

```
int     api_releasedc( );
```

api_releasedc

QuickBASIC

DECLARE FUNCTION api.releasedc% CDECL ( )

EXAMPLE—Release datacomm.

```
' $INCLUDE: 'API.INC'
i% = api.releasedc
```

TURBO PASCAL

function api_releasedc :integer;

EXAMPLE

```
i := api_releasedc
```

api_reset

Sends Reflection a hard reset. Check the Reflection *Technical Reference Manual* for a description of the hard reset process. In addition, api_reset performs the following three functions:

- api_flushcmd()
- api_clrkeybd ()
- api_endcommands()

To prevent the host from performing a hard-reset, issue the command "SET EXITS-DISABLED YES".

INPUT

```
CX = 21
AX = DE52H
DX = 0
```

OUTPUT

Should not return an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,21
int  21h
```

C

```
int       api_reset( );
```

api_reset

QuickBASIC

DECLARE FUNCTION api.reset% CDECL ( )

EXAMPLE—Hard reset Reflection.

```
' $INCLUDE: 'API.INC'

i% = api.reset
```

TURBO PASCAL

function api_reset :integer;

EXAMPLE—Hard reset Reflection.

```
uses     apiunit;
var      i : integer;
begin
         i := api_reset;
end.
```

api_screenread

Read text from the Reflection screen in background.

Screen reads do not extend to the function key area. Returns an error if the screen is in graphics mode.

INPUT

```
CX = 11
AX = DE52H
DX = 0
BH = ROW         0 relative
BL = COL         0 relative
ES:DI            points to buffer to contain screen text
SI = number of bytes to read
```

OUTPUT

No error:

```
AX = 0
```

Error:

```
AX = 106H asked for read off of screen
     103H Screen in graphics mode
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
mov     bh,[start_row]
mov     bl,[start_column]
mov     si,[bytes_to_read]
push     ds
pop     es
ASSUME  ES:DATA
mov     di,offset screen_buffer
mov     cx,11
int     21h
or      ax,ax
jnz     screen_read_error
```

C

```
int     api_screenread( buffer, row, col, length );
                                 returns non-zero if error
char    *buffer;
int     row;
int     col;
int     length;
```

EXAMPLE

```
main()
{

/* read a string off of the screen */

char  screen_buffer[30];
api_startcommands();

/*  Home cursor, clear screen go down 4 and right 6   */
/*  NOTE - THESE ARE HP ESCAPE SEQUENCES - !!!!!! */

api_docommand("display '^[H^[J^[B^[B^[B^[B^[C^[C^[C^[C^[C^[C'");

/*  display sample text */

api_docommand("display 'line at row 4 column 6'");

/*  read 22 bytes off of screen at row 4 column 6  */

api_screenread(screen_buffer, 4, 6, 22 );

screen_buffer[22]='0';
if (strcmp(screen_buffer,"line at row 4 column 6"))
  printf("returned string doesn't compare\ n");
else
  printf("Found [%s]\n",screen_buffer);
api_endcommands();
}
```

QuickBASIC

DECLARE FUNCTION api.screenread% CDECL
(SEG scrbuffer$, BYVAL row%, BYVAL col%, BYVAL rvarlen%)

TERMS

scrbuffer$
Buffer to receive screen data–scrbuffer$ must first be initialized to size sufficient to contain data

row%
Row to start read

col%
Column to start read

rvarlen%
Number of bytes to read

EXAMPLE—Print Reflection screen (characters only - no color).

```
' Print the Reflection screen.
' $INCLUDE: 'api.inc'
scbuf$ = space$(80*24)
i% = api.screenread( scbuf$, 0%, 0%, 80%*24% )
cls
print scbuf$
end
```

TURBO PASCAL

function api_screenread(length ,column, row:integer; var x: buffer) :integer;

TERMS

length
Number of bytes to read

column
Column to start read

row

Row to start read

x

Buffer to receive data–see apiunit.pas for declaration

EXAMPLE—Read and print 80 columns from Reflection screen.

```
{$V-}
uses     apiunit;
var      scrbuf  : buffer ;
         j, row, col, k, i  : integer;
         scrtext            : string[255];
begin
   write( 'enter row and column to start screen read :');
   readln( row, col );
   j := 80;
   if api_screenread( j, col, row , scrbuf ) <> 0 then exit;
   k := 1;
   scrtext[0] := chr(j);
   while k <= j do
        begin
             scrtext[k] := scrbuf[k-1];
             k := k+1;
        end;
   writeln( scrtext );
end.
```

api_searchscreen

Search the Reflection screen for a string. If found, return row and column location.

INPUT

```
CX =    32
AX =    DE52H
DX =    0
BH =    Column  (0 relative) where to start search
BL =    Row  (0 relative)
ES:DI = Buffer containing null-terminated string
        to search for
```

OUTPUT

No Error:

```
AX = 0  found string
BH =    Column  (0 relative)
BL =    Row  (0 relative)
```

Error:

```
AX <> 0 String not found
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
push    ds
pop     es
ASSUME  ES:DATA
mov     bl,[row]
mov     bh,[column]
mov     di,offset search_string ; string to search for
mov     cx,32
 int    21h
```

```
or      ax,ax
jnz     string_not_found
mov     [row],bl     ; store location where found
mov     [column],bh

column        db     20
row           db     2
search_string db     "Enter Password:"
```

C

```
int     api_searchscreen( search_string, row, col );
                                    returns NZ if not found
int     &row;
int     &col;
char    *search_string;
```

EXAMPLE

```
main()
{
    Search screen for field. If found, print contents

       /* Search screen for a prompt */

       int  row, col;
       char  *string = "Employee Name:";
       char  employee[30];
       col = 0;
       row = 0;
       if (api_searchscreen( string, &row, &col ))
             printf("Can't find employee field\n");
       else  {
             col = col + 15;    /* skip 'Employee Name:' */
             api_screenread( employee, row, col, 30 );
             employee[30] = '0';
             printf("Employee (%s)\n",employee);
       }
}
```

QuickBASIC

DECLARE FUNCTION api.searchscreen% CDECL
(SEG srchstrng$, SEG row%, SEG col%)

TERMS

srchstrng$
String being searched for

row%
Row where search is to begin

col%
Column where search is to begin–if found, row% and col% are set to the location of string

EXAMPLE—Search screen for field. If found, print contents.

```
' $INCLUDE: 'api.inc'
srchtext$ = "Employee Name:"
namevar$ = SPACE$(40)    ' allow 40 spaces for name variable
column% = 0
row% = 0
IF api.searchscreen(srchtext$, row%, column%) <> 0 THEN
       PRINT "Can't find employee field"
ELSE
       PRINT "found", row%, column%
       column% = column% + 15
       i = api.screenread(namevar$, row%, column%, 40)
       PRINT "Employee name : "; namevar$
END IF
END
```

TURBO PASCAL

function api_searchscreen(var col, row :integer;var x:string):integer;

TERMS

col
Starting column

row
Starting row

x

String to search for–if string is found, col and row contain string position on screen

EXAMPLE—Search screen for field. If found, print contents.

```
{$V-}
uses  apiunit;
var   searchtxt  : string[30];
      row  : integer;
      col  : integer;
      i  : integer;
      namvar  : string[40];
      scrtext : buffer;
begin
   row := 0;
   col := 0;
   searchtxt := 'Employee Name:';
   if api_searchscreen(col, row, searchtxt) <> 0 then
            writeln( 'employee field not found')
   else
      begin
           col := col + 15;
           i := api_screenread( 40, col, row, scrtext);
           i := 0;
           repeat
                namvar[i+1] := scrtext[i];
                i := i + 1;
           until i = 40;
           namvar[0] := chr(40);
           writeln( 'Employee name : ', namvar );
      end;
end.
```

api_setfkey

Directly set an R1/R7 HP user key or an R2/R4 softkey. DEC UDKs may only be set via the DISPLAY command with the appropriate escape sequence.

INPUT

```
CX = 23
AX = DE52H
DX = 0
ES:BX point to  ukey/softkey structure
SI = key number
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = 105H bad key
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
push    ds
pop     es
assume  es:data
mov     bx,offset key_structure
mov     si,3       ; set user key 3
mov     cx,23
int     21h
or      ax,ax
jnz     key_error


key_structure label byte
```

```
ukey_attr      db    0  ; normal (as if typed at keyboard)
ukey_lablen    db    16  ; length of label text  (2 rows )
ukey_deflen    db    23  ; length of definition string
reserved       dw    ?
ukey_text      db    '  LOGON ',' HP ','HELLO George,mgr.sales',0dh
```

C

```
int       api_setfkey( key_buffer, keynumber);
struct    ukey                  *key_buffer;
int       keynumber;
```

EXAMPLE

```
main()
{

        /* change function key 3 to LOCAL */

        int  keynumber = 3;
        struct  ukey  key_buffer;
        struct  ukey  *sptr;
        sptr = &key_buffer;
        api_getfkey( sptr, keynumber );
        sptr->ukey_attr = 1;      /* 1 = local */
        api_setfkey( sptr, keynumber );
        printf("Userkey 3 set to LOCAL  [L]  \n");
}
```

QuickBASIC

See api_getfkey on page 99.

TURBO PASCAL

See api_getfkey on page 99.

api_setvar

Load ASCII string into Reflection variable. Reflection variables are limited in length to 80 characters. The default number of variables is 10, but can be increased via a SET command to 800.

This command should only be executed as part of a synchronous command sequence, otherwise it can return suspect values, i.e., it may interrupt Reflection changing one variable to another so a false result could occur.

Variables cannot always be null-terminated in this way since it is possible for them to contain nulls themselves.

INPUT

```
CX = 8
AX = DE52H
DX = 0
ES:BX points to 80 byte buffer
SI = variable #
DI = length of variable
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX = 10AH bad variable no
     106H bad length
```

Assembler

EXAMPLE—Set variable V2 to value below.

```
mov             ax,0DE52H
xor             dx,dx
mov             cx,8
mov             si,2
mov             di,VARLENGTH
```

```
        push        ds
        pop         es
        ASSUME      ES:DATA
        mov         bx,offset Variable_Buffer
        int         21h

        mov         byte ptr [variable_length+SI],0
        or          ax,ax       ; was there an error
        jnz         Bad_variable_number  ; yes.


    Variable_Buffer db "Enter this string in V2"
    VARLENGTH       EQU  $ - Variable_Buffer
```

C

int	api_setvar(var_buffer, varno, varlength);	returns non-zero if error
char	*var_buffer;	
int	varno;	variable number 0-799
int	varlength;	

For a C programming example, see api_getvar example on page 113.

QuickBASIC

DECLARE FUNCTION api.setvar% CDECL (SEG var$, BYVAL varnumber%)

TERMS

var$
String containing data to load into variable

varnumber$
Variable number

EXAMPLE—Set a command language variable.

```
    ' $INCLUDE: 'API.INC'
     a$ = "abc" + chr$(0) + "def" ' a$ contains a null
       i%=api.setvar(a$, 3 ) ' set V3 to a variable that contains a null
```

TURBO PASCAL

function api_setvar(i :integer; var x: string) :integer;

TERMS

i

Variable number

x

String data to load into Reflection variable

EXAMPLE—Set a command language variable.

```
{$V-}
uses apiunit;
var   cmdvar  : string[80];
      retvar  : string[80];
      i       : integer;
begin
      cmdvar := 'Load this string in V3';
      i := api_setvar( 3, cmdvar );
      i := api_getvar( 3, retvar );
      writeln ( retvar );
end.
```

api_startcommands

Prepares Reflection for a series of synchronous commands. This call returns an error if Reflection is busy, i.e., if it is already performing a command or file transfer. If successful, this call turns off incoming datacomm (characters remain in receive buffer) unless a docommand( ) is actually being processed.

After the api_startcommands function is issued, Reflection no longer reads incoming datacomm out of the receive buffer unless a Reflection command is actually being executed. In most cases, startcommands should not be issued until you are ready to start processing commands. Depending on the type of receive pacing, you may run the risk of overflowing the receive buffer. Note that datacomm is turned back on while Reflection is in foreground. Upon return to background, datacomm is turned back off unless a command is active.

INPUT

```
CX = 5
AX = DE52H
DX = 0
```

OUTPUT

No error:

```
AX = 0
```

Error:

```
AX = 100H  Not available (busy)
```

Assembler

EXAMPLE

```
mov    ax,0DE52H
xor    dx,dx
mov    cx,5
int    21h
or     ax,ax
jnz    Reflection_Busy
```

C

int api_startcommands( ); returns non-zero if error or busy

EXAMPLE

```
if(api_startcommands())
        printf( "Reflection Busy\n");
```

QuickBASIC

DECLARE FUNCTION api.startcommands% CDECL ( )

EXAMPLE—See if we can start docommand sequence.

```
' $include: 'api.inc'

if api.startcommands% > 0 then print "Reflection busy"
end
```

TURBO PASCAL

function api_startcommands :integer;

EXAMPLE

```
uses    apiunit;
var
        i : integer;
begin
        if api_startcommands <> 0 then
             writeln ( 'Reflection busy' );
end.
```

api_wait

Wait for Reflection to empty the command and/or keyboard queues. Does not return until Reflection is idle. When a series of commands are queued, but the API client program cannot continue until the commands have been completed, this call could be used. Handle with care: a keyboard lock can make the system hang.

INPUT

```
CX = 26
AX = DE52H
DX = 0
```

OUTPUT

Should not return an error.

Assembler

EXAMPLE

```
mov  ax,0DE52H
xor  dx,dx
mov  cx,26
int  21h
```

C

int   api_wait( );

EXAMPLE

```
main()
{

        /* Queue commands and WAIT till complete */

        api_qcmd("wait 0:0:5");
        api_qcmd("display '^g'");
        api_wait();
}
```

QuickBASIC

DECLARE FUNCTION api.wait% CDECL ( )

EXAMPLE—Queue commands and WAIT until complete.

```
' $INCLUDE: 'api.inc'
i% = api.qcmd("WAIT 0:0:5")
i% = api.qcmd("display '^g'")
i% = api.wait
```

TURBO PASCAL

function api_wait :integer;

EXAMPLE

```
uses apiunit;
var i : integer;
var cm : string[80];
begin
        cm := 'SEND ABC TO XYZ';
        i  := api_qcmd( cm );
        i  := api_wait    (* wait until command completed       *)
end.                      (* note: may cause program to hang if *)
                          (* command can't complete             *)
```

api_writeasync

Write a single character to the Reflection transmit buffer for transmission to the host.

INPUT

```
CX = 34
AX = DE52H
DX = 0
SI = character
```

OUTPUT

No error:

```
AX =  0
```

Error:

```
AX <> 0 buffer full
```

Assembler

EXAMPLE

```
mov     ax,0DE52H
xor     dx,dx
mov     cx,34
mov     si,'a'
int     21h
or      ax,ax
jnz     buffer_full
```

C

```
int     api_writeasync( c );   write character c. return NZ
                               if xmit buffer full
int     unsigned               c;
```

EXAMPLE

```
main()
{
  Transmit the contents of the string

            /* write a string */

int         c;
char        *xmit_string   = "Send this string";
            while (( c =*xmit_string++)  != '0' )  {
                  while(!api_xmitstatus())
                       if (kbhit())
                             break;
                  api_writeasync( c );
            }
}
```

QuickBASIC

DECLARE FUNCTION api.writeasync% CDECL (BYVAL c%)

TERMS

c%

The ordinal value of the character within the ASCII character set. This can be derived via the ASC function. c% = asc(c$).

For an example see api_rdchar on page 133.

TURBO PASCAL

function api_writeasync(c :integer) :integer;

TERMS

c

Integer containing character to be transmitted

For an example see api_rdchar on page 133.

api_xmitstatus

Return the amount of free space in the transmit buffer. Should be used before api_writeasync to prevent accidental overrun of the transmit buffer and loss of data.

INPUT

```
CX = 35
AX = DE52H
DX = 0
```

OUTPUT

```
AX = amount of free space
```

Assembler

EXAMPLE

```
mov   ax,0DE52H
xor   dx,dx
mov   cx,35
int   21h
or    ax,ax                    ; is there any free space
jz    buffer_full              ;  NO
```

C

```
int      api_xmitstatus();   ; returns # bytes of free space
                               if xmit buffer full
```

For a C programming example, see api_writeasync on page 158.

QuickBASIC

DECLARE FUNCTION api.xmitstatus% CDECL ( )

EXAMPLE—Send an ASCII file out the datacomm port without flow control.

```
' $INCLUDE: 'api.inc'
      open "filename" for input as #1
readloop:
  if eof(1) then goto endit
  line input #1, x$
  for charpos = 1 to len(x$)
        do          ' delay until transmitter ready
        loop while api.xmitstatus = 0
        i%=api.writeasync( asc( mid$(x$,charpos, 1) ))
  next charpos

  do          ' delay until transmitter ready
  loop while api.xmitstatus = 0
  i%=api.writeasync( 13% )      ' send a carriage return
  for j% = 1 to 100 : next j%     ' delay after carriage return
  goto readloop
endit:
  close (1)
  end
```

TURBO PASCAL

function api_xmitstatus :integer;

For an example see api_rdchar on page 133.

Application Program Interface

Chapter 12

How to Write an API Application

It is assumed that you are already familiar with Reflection and have written Reflection command language programs. If you haven't, that is probably the best and easiest place to start. Automating user tasks that involve a host computer is complex: your command file has to be completely synchronized with the application on the host. Even simple jobs, such as a command file to log a user on to a host computer, are not necessarily trivial.

API adds another level of complexity because the Reflection screen is no longer visible to indicate timing problems or error messages. When things don't work, it can be difficult to figure out what is wrong. An API application has to communicate with Reflection, which in turn communicates with the host. Creating API applications for Reflection requires knowledge of all of the following:

- Host communication issues
- The host application
- Reflection (especially Reflection command language)
- The programming language in which the API is written

12.1 Queued vs. Synchronous Commands

Some API commands are queued (asynchronous) and some are synchronous. Of the API functions that are provided, the synchronous command support offered by the *api_docommand* function is the easiest to use, and usually the most appropriate for interactive applications.

Synchronous Commands

Synchronous functions require that the API application wait for the function to be completed before proceeding. This is how most programs operate. When a call is made, it must complete before another call is issued. Synchronous commands give the API application direct control over operations and when they occur. When control returns to the API application, the command has either completed, or a return code indicates what went wrong.

Asynchronous Commands

Asynchronous commands capitalize on the multitasking capability of Reflection. A queued function requests Reflection to do something as time permits. The request is processed when Reflection gets to it in the queue, and there's no guarantee exactly when it will be done. The advantage is that the function call returns to the API application immediately (as soon as the request is queued), so the API application can continue while Reflection works on the queued request.

For example, if you transfer a file to the host programmatically using a synchronous call, the following happens:

- The function is called
- The file is transferred
- The function returns with an error code when the transfer is complete

However, if the program must simultaneously interact with a user, you may want to simply issue the transfer request, and then return the API application's attention to the user. File transfers are unlike most system calls (write to file, read character from keyboard, and so on) in that they can take a long time. Because the user gets no visual feedback, it may appear that the PC is hung and needs to be rebooted.

File Transfer Example

By using queued commands, you can get around this problem and pop up the file transfer screen during the transfer so the user will at least know that something is happening. The following code fragment demonstrates this. It queues 3 commands: a command to perform the file transfer, a command to get the error code into a variable, and a command to switch back to background. It then issues the *api_popup* command so the user can view the file transfer. As soon as the file transfer completes, the error-code is captured and Reflection returns to the background.

While these commands are queued, the effect is synchronous since the file transfer is performed in the foreground. It is important to capture the error code since it is possible for the user to stop the transfer via the STOP TRANSFER key. If this happens, capturing the error code will help you find out why the transfer failed.

```
char   error_text[81];
int    length;
api_qcmd( "SEND testall.c TO data1;p ASCII DELETE");
api_qcmd( "LET V9=ERROR-CODE" );
api_qcmd( "BACKGROUND");
api_popup();
api_getvar( error_text, 9 , &length);
error_text[length]='\0';
printf( "File transfer Completed\n");
printf( "Error code %s",error_text );
exit();
```

12.2 Timeouts

If you are using synchronous calls (*api_docommand*), where the API application has to wait for the call to complete, make sure that you specify a timeout on any WAIT, HOLD, READHOST, or other commands. Your application could hang waiting for a response that never arrives.

For example, if you send Reflection a command like WAIT FOR 'Main Menu', be very sure that the text 'Main Menu' is going to come down from the host computer within a reasonable time period. Otherwise, the function call will never return, and the application will hang waiting for the required message to come from the VAX or HP 3000. Rather than risk this, make sure that all commands have a timeout period:

```
WAIT 0:0:2 for 'Main Menu'
```

This command waits up to 2 seconds for the string. If the string has already arrived or arrives in 5 seconds, the call returns quickly to the API application. Another API function lets you find out if the string was actually found: *api_found* .

12.3 Keyboard Reads

You will probably want to avoid *api_docommand("accept v1")* . ACCEPT is a Reflection keyword that reads input from the keyboard. Since Reflection is in background, the user won't be able to type anything (unless you have queued keys) and your program will hang.

You are probably accustomed to using the C *gets* command or some other variant for reading keyboard input into your C program.

Unfortunately, many Microsoft C and Turbo Pascal keyboard reads use DOS calls like function 3FH (read file handle). This is similar to UNIX. When these calls are made to read keyboard input, they don't complete until you enter a whole line terminated by a carriage return. Since DOS is not re-entrant and only one process can make a DOS call at a time, calls of this type cause Reflection to freeze until they complete. No multitasking takes place. This can halt a file transfer and cause the host or Reflection to timeout. This should not be an issue unless you use queued commands.

To avoid this problem, you will need to use different calls to read from the keyboard. The libraries (on disk) include the calls *rgets* for C and *rfreadkeybd* for Pascal.

12.4 Conflicting Commands

When Reflection is running in the background, it is constantly accepting commands from both the host computer and the API application. These commands may occasionally conflict. For instance, the host may lock the keyboard just as the API application is sending keystrokes for the next screen.

The host can also invoke Reflection commands by prefixing Reflection commands with the escape sequence ^{E}C*&oC* . Since some host programs use this facility, Reflection may get a command sent with this escape sequence while it is executing a command or command file begun by the user or the API application. Since Reflection can't process two commands at once, the host commands fail. Because the host program isn't behaving the same way it was when you ran it from the keyboard, the API application calls also fail.

SET HOST-INITIATED-COMMANDS NO turns off host-initiated commands: if any commands come from the host, they will fail. However, this still seriously affects or aborts the host application. There's no solution to problems of this kind except awareness. It is often helpful to turn on display functions (Reflection 1 and 7) or display controls (Reflection 2 and 4) to see what the host is actually doing. Host commands, status requests, etc., are then visible so that you can see what your program has to take into account.

12.5 Preventing User Exits

An unguarded API application can be unintentionally sabotaged by the user. If the user pops up Reflection and hardexits (Alt-X), your application will have nothing to interface with. The only way that your application can detect this is by looking at return codes. If you're programming in Assembler, *AX* returns *0DE00H* if Reflection has been uninstalled. In C or Pascal, the API function returns *-1*. To make sure that the user can't abort Reflection, use the following Reflection SET commands:

SET HOT-KEY NONE
Prevents the user from popping up Reflection

SET EXITS-DISABLED YES
Prevents the user from exiting or resetting Reflection via Alt-X or a hard reset

SET DISABLE-INTERRUPT YES
Prevents the user from stopping a command via Ctrl-Y

These settings will make it more difficult for you to debug your program, so don't set them until the program is working correctly.

12.6 Datacomm in api_docommand Sequences

Once the *api_startcommands* function is performed, datacomm is turned off except during the actual execution of an *api_docommand*. This is essential to allow your application to remain synchronized with the host program. When datacomm is turned off, Reflection will not read any incoming characters from its receive buffer unless a command is being processed. Characters are still received, but they remain in the buffer. Reflection ignores them.

This means that the API application can pause and prompt the user for something without missing any incoming characters. If the API application is logging a user onto a DEC VAX, for example, and has to stop and prompt for a password, there is no way for the password prompt to go undetected. The prompt stays in Reflection's receive buffer waiting to be read by the next command:

```
api_docommand("WAIT 0:0:30 FOR 'Password:'");
```

This command returns immediately since the prompt was already in the receive buffer.

If, instead, Reflection always read datacomm and the above wait command were issued, the WAIT would time out in 30 seconds without finding the 'Password:' prompt since it had already been received. The API application would assume that there was a problem communicating with the VAX and proceed accordingly.

To prevent Reflection's receive buffer from being overrun by the host during delays by the API application, you must set the appropriate type of flow control. For both HP and DEC, this will usually be XON/XOFF receive pacing.

When a Reflection WAIT FOR command is issued, it should be followed with an *api_found* command. This tells your application whether the WAIT actually found the string you were looking for, or whether it timed out without finding it.

```
api_docommand("WAIT 0:0:30 FOR 'HP3000'");
if (api_found())
        printf("Logged on Successfully\n");
else
        .....
```

12.7 Configuration Issues

The *api_getinfo* command returns a structure of configuration items that appear on configuration or set-up screens but have no corresponding SET command. Most of these items will not affect your API application.

You can use the VALUE command to get more information on common configuration items such as baud rate, parity, etc. To find the baud rate, for instance, execute the following sequence:

```
api_docommand( "LET V3=VALUE(BAUD)" );
api_getvar( baudvalue, 3, length );
baudvalue[length]='\0';
printf( "Reflection Baud Rate = %s\n",baudvalue);
```

SET commands can be issued to change most configuration items. Some items cannot be changed except via SET commands or DISPLAY commands that use escape sequences.

Chapter 13

Converting Command Language to API

The following command file dials a modem and attempts to log on to an HP 3000 computer. Following it on page 171 is the same function in a C program using API calls.

13.1 Command Language Dialing Program

```
SET DATACOMM-PORT COM1
SET BAUD 2400
SET CHARACTER-DELAY 80
;
;Need to set big delays and long waits when talking to modems
;
;Initialize retry_count
LET V1 = 0
DISPLAY 'Initializing Modem...^M^J'
WAIT 0:0:1
TRANSMIT '+++'
WAIT 0:0:2 FOR 'OK'
WAIT 0:0:.5
TRANSMIT 'ATH^M'
WAIT 0:0:2 FOR 'OK'
IF NOT FOUND
    LET V4 = 'No Response from modem^M^J'
    GOTO FAIL
ENDIF
WAIT 0:0:1
DISPLAY 'Dialing Modem...^M^J'
TRANSMIT 'ATDT 1234567^M'
WAIT 0:0:45 FOR 'CONNECT'
IF NOT FOUND
    LET V4 = 'Did not receive CONNECT Message^M^J'
    GOTO FAIL
ENDIF
DISPLAY 'Connected to Remote Modem^M^J'
;
; Modem indicates connected - attempt to log on
;
:TRYAGAIN
IF V1 < 6
    LET V1 = V1 + 1
    TRANSMIT '^M'
    WAIT 0:0:1 FOR '^Q'
    IF NOT FOUND
        DISPLAY 'Try $1 Failed^M^J'
        GOTO TRYAGAIN
    ELSE
        GOTO LOGON
    ENDIF
ELSE
    LET V4 = 'Never got host prompt^M^J'
    GOTO FAIL
ENDIF
DISPLAY 'Entering logon and password^M^J'
TRANSMIT 'hello doni,mgr.pc50/joshua^M'
WAIT 0:0:30 FOR 'HP3000'
IF NOT FOUND
    DISPLAY 'Logon rejected - hangup modem^M^J'
    WAIT 0:0:2
    TRANSMIT '+++'
    WAIT 0:0:3 FOR 'OK'
    WAIT 0:0:1
    TRANSMIT 'ATH^M'
    WAIT 0:0:2 FOR 'OK'
    IF NOT_FOUND
        LET V4 = 'Hangup complete^M^J'
        GOTO FAIL
    ELSE
        LET V4 = 'Hangup failed^M^J'
        GOTO FAIL
    ENDIF
ENDIF
WAIT 0:0:30 FOR ':^Q'
SET CHARACTER-DELAY 0
DISPLAY 'Successful Logon^M^J'
STOP
;
;       FAIL 'SUBROUTINE'
;
:FAIL
DISPLAY  V4
STOP
```

13.2 C Version

The C version of the same command file using synchronous (do while you wait) commands follows. Notice that C is used to handle all of the logic and flow control rather than Reflection.

```
main()
{
    int  retry_count;
    char    serbuf[32];
    if (api_instchk(serbuf)) {
            printf("API not present\n");
            exit();
    }
    if (api_startcommands()) {
            printf("API busy\n");
            exit();
    }
```

```
api_docommand("alert 'API - AUTODIAL - LOGON'");
api_docommand("set datacomm-port com1");
api_docommand("set baud 2400");
api_docommand("set character-delay 80");

/* need to set big delays and long waits when talking to modems */

retry_count = 0;
printf("Initializing Modem...\n");
api_docommand("wait 0:0:1");
api_docommand("transmit '+++'");
api_docommand("wait 0:0:2 for 'OK'");
api_docommand("wait 0:0:.5");
api_docommand("transmit 'ATH^m'");
api_docommand("wait 0:0:2 for 'OK'");
if (!api_found())
     fail("No Response from modem\n");

api_docommand("wait 0:0:1");
printf("Dialing Modem...\n");
api_docommand("transmit 'ATDT 1234567^m'");
api_docommand("wait 0:0:45 for 'CONNECT'");
if (!api_found())
   fail("Did not receive CONNECT Message\n");

printf("Connected to Remote Modem\n");

/* Modem says we're connected - see if we can log on */

for ( retry_count = 0 ; retry_count < 6 ; retry_count++) {
     api_docommand("transmit '^m'");
     api_docommand("wait 0:0:1 for '^q'");
     if (!api_found())
          printf("Try %d Failed\n",retry_count);
     else
          break;
}
```

```
    if ( retry_count >= 6)
            fail("Never got host prompt\n");
    else
            printf("Entering logon and password\n");

    api_docommand("transmit 'hello doni,mgr.pc50/joshua^M'");
    api_docommand("wait 0:0:30 for 'HP3000'");
    if (!api_found())  {
        printf("Logon rejected - hangup modem\n");
        api_docommand("wait 0:0:2");
        api_docommand("transmit '+++'");
        api_docommand("wait 0:0:3 for 'OK'");
        api_docommand("wait 0:0:1");
        api_docommand("transmit 'ATH^M'");
        api_docommand("wait 0:0:2 for 'OK'");
        if (!api_found())
           fail("Hangup complete\n");
        else
           fail("Hangup failed\n");
    }
    api_docommand("wait 0:0:30 for ':^q'");
    api_docommand("set character-delay 0");
    printf("Successful Logon\n");
    api_endcommands();
}

fail(s)
char    *s;
{
    printf("%s",s);
    api_endcommands();
    exit();
}
```

13.3 Queuing Keystrokes

Be careful when using single and double quotes with the *api_qkeys* function. When passing keystrokes, keys are either passed as **function names**, (the same names used in Reflection's keyboard remapping) or as **quoted strings**. Function names such as *HARD-RESET*, *HOST-BREAK*, *RETURN*, *F1*, and *F2* are not put in quotes. Keystrokes passed as quoted strings must be in quotes.

The following passes the function name *RETURN* as a non-quoted string to API—it is the same as pressing Return:

```
api_qkeys("RETURN");
```

The example below passes the keystrokes **r**, **e**, **t**, **u**, **r**, and **n** to the API application.

```
api_qkeys("'return'");
```

Index