Network General Corporation

Sniffer Network Analyzer®

Protocol Interpreter Development Kit

Errata
Sniffer Network Analyzer: Protocol Interpreter Development Kit
October 30, 1992

The following errata apply to the *Sniffer Network Analyzer: Protocol Interpreter Development Kit.*

1. On page 1–4, under the heading, "Writing a Protocol Interpreter," the manual informs you that you will need to obtain Version 6.00AX of the Microsoft C compiler. This compiler and the necessary library files are now included as part of the Protocol Interpreter Development Kit. You no longer need to obtain them separately.

2. On page 1–6, Figure 1–1 lists the files provided with the Protocol Interpreter Development Kit. In addition to the files listed, the following files are also provided as part of the Microsoft 6.00AX C Compiler:

Files Provided With Microsoft 6.00 AX C Compiler		
CL.EXE	CL.MSG	C1L.EXE
LLIBCE.LIB	C1.ERR	C23.ERR
CL.ERR	C2L.EXE	C3L.EXE

3. On page 1–7, under the heading, "Installing the Microsoft C Compiler," the document gives instructions for installing the compiler. The Protocol Interpreter Development Kit now includes all the tools required to write new protocol interpreters and build new Sniffer analyzers, including the compiler and the library. You will not need to install the Microsoft C Compiler because it is already installed with the Protocol Interpreter Development Kit.

4. On page 1–31, Step 4 of the procedure, "To build a new Sniffer analyzer," lists several changes that should be made to the BUILDSNF.BAT file. The necessary files for Microsoft C 6.00AX are now distributed with the Protocol Interpreter Development Kit. If you are using the compiler supplied with your Protocol Interpreter Development Kit, then the BUILDSNF.BAT file is already set correctly. You will not need to change the BUILDSNF.BAT file unless you are using a custom development environment. If this is the case, you will need to change the lines in BUILDSNF.BAT that set the three variables listed in Step 4 on page 1–31.

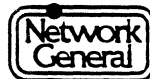
5. To use the Protocol Interpreter Development Kit, you must add the following line to the CONFIG.SYS file of the Sniffer Network Analyzer:

```
DEVICE=C:\DOS\HIMEM.EXE
```

   Note: If your Sniffer Network Analyzer has less than 32MB of RAM, you must remove the above command line before starting the Sniffer Network Analyzer.

6. Before using the Microsoft C compiler, type the following command:

```
SET PATH=C:\DOS;C:\TOOLS;C:\TOOLS\C600AX
```



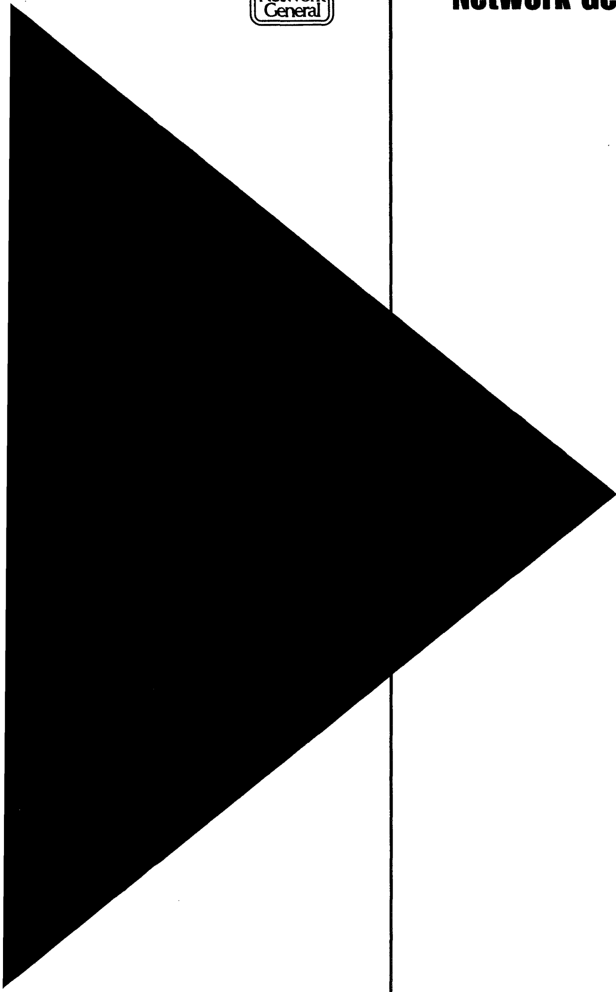
We simplify network complexity.™

Network General Corporation
4200 Bohannon Drive
Menlo Park, CA 94025
TEL: (415) 688-2700
FAX: (415) 321-0855

Network General Europe
Belgicastraat 4
1930 Zaventem, Belgium
TEL: (32-2) 725-6030
FAX: (32-2) 725-6639

Network General Canada, Ltd.
2275 Lakeshore Blvd., West, 5th Floor
Etobicoke, Ontario M8V 3Y3 Canada
TEL: +1 (416) 259-5022
FAX: +1 (416) 259-3727

Network General Corporation



Sniffer Network Analyzer®

Protocol Interpreter Development Kit

DISCLAIMER OF WARRANTIES

The information in this document has been reviewed and is believed to be reliable; nevertheless, Network General Corporation makes no warranties, either expressed or implied, with respect to this manual or with respect to the software and hardware described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. The entire risk as to its quality and performance is with the buyer. The software herein is transferred "AS IS."

Network General Corporation reserves the right to make changes to any products described herein to improve their function or design.

In no event will Network General Corporation be liable for direct, indirect, incidental or consequential damages at law or in equity resulting from any defect in the software, even if Network General Corporation has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

This document is copyrighted and all rights are reserved. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Network General Corporation.

Sniffer Network Analyzer is a trademark of Network General Corporation.
All other registered and unregistered trademarks in this document are the sole property of their respective companies.

©Copyright 1986 – 1992 by Network General Corporation.
All rights reserved. Present copyright law protects not only the actual text, but also the "look and feel" of the product screens, as upheld in the Atari and Broderbund cases.

Document prepared by Jim Mar with contributions from David Trousdale.

April 1992

P/N: PA-7000

Table of Contents

Preface v

About This Manual v
Navigational Aids Used in This Manual v
Manuals for the Sniffer Network Analyzer v
Conventions Used in This Manual vi
Special Notations vi
Terminology vi
Other Sources of Information vi
On-Line Help vii
Technical Support vii

Chapter 1. Extending and Customizing Protocol Interpreters 1-1

Overview 1-3
What Does a Protocol Interpreter Do 1-3
Writing a Protocol Interpreter 1-4
Porting Current Protocol Interpreters 1-4
Installing the PI Development Kit 1-5
Installing the Microsoft C Compiler 1-7
Coding a Protocol Interpreter 1-7
Calling Conventions for Protocol Interpreters 1-7
Protocol Interpreter Data Structure 1-8
Generating Output from Protocol Interpreters 1-9
The Protocol Interpreter Formatting Routines 1-10
Sprintf Function in the PI Development Kit 1-17
Adding Symbolic Names to the Name Table 1-20
Declaring Embedded Addresses 1-20
Displaying Symbolic Names 1-21
Adding Summary Line Flags 1-21
Using Other Protocol Interpreters 1-21
Dependencies on Other Frames 1-24
Integrating into Existing Sniffer Analyzer Code 1-27
Registering Protocol Interpreters 1-28
Building a New Sniffer Analyzer 1-31
Programming and Debugging Hints 1-34

Index

List of Figures

Chapter 1. Extending and Customizing Protocol Interpreters 1-1

Figure 1–1. PI Development Kit files. 1-6
Figure 1–2. Global static variables available to the protocol interpreter. . . . 1-8
Figure 1–3. PIF Routines. 1-12
Figure 1–4. PDK flags.. 1-18
Figure 1–5. Sketch for structure of a new protocol interpreter. 1-23
Figure 1–6. Using the Frame Context.. 1-26
Figure 1–7. Example capture filter code. 1-33

PREFACE

Network General

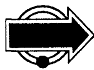
Preface

About This Manual

Welcome to the *Protocol Interpreter Development Kit*. This manual explains the rules and conventions for writing programs that extend the Sniffer Network Analyzer's ability to interpret protocols.

Navigational Aids Used in This Manual

This manual uses icons in the margin to help you locate important information as explained below:



IMPORTANT INFORMATION. Next to this icon is information that is especially important; you should be certain to read it carefully before you proceed.

WARNING. Next to this icon are instructions that you must follow to avoid possible damage to data files, program files, or hardware devices.

PROCEDURE. Next to this icon is a series of steps for accomplishing a particular task.

Manuals for the Sniffer Network Analyzer

Two types of manuals accompany the Sniffer Network Analyzer. The primary manuals, which include this manual, describe the system's normal operations; the supplementary manuals describe the programs that configure and test the system's various hardware and software components for troubleshooting. The actual manuals in your shipment depend on the system configuration.

The following table describes the primary manuals.

For Information On...	Read...
Installing and configuring the Sniffer Network Analyzer.	*Sniffer Network Analyzer: Installation Guide*
Operating the analysis functions on an Ethernet or Token-Ring network.	*Expert Sniffer Network Analyzer Operation*
Operating the monitor functions on an Ethernet network. Using the monitor features effectively to detect network abnormalities.	*Sniffer Network Analyzer: Ethernet Monitor Operations.*

For Information On...	Read...
Operating the monitor functions on a token ring network. Using the monitor features effectively to detect network abnormalities.	*Sniffer Network Analyzer: Token-Ring Monitor Operations*
Various network and protocol types.	*Sniffer Network Analyzer: Network and Protocol Reference*

Conventions Used in This Manual

Special Notations

The following describes the conventions used in this manual:

Bold — Menu options are in bold type. For example:

Move to **Display**, and press Enter.

UPPERCASE — Filenames and commands you type at a DOS prompt are in uppercase. For example:

Modify the AUTOEXEC.BAT file if necessary. To duplicate the file, use the COPY command.

`Screen font` — Screen messages are printed in monospaced font. For example:

If a monitoring session is in progress, the following message appears:

```
You must stop monitoring before you can use this feature.
```

Terminology

Hexadecimal numbers in the manual are followed by "(hex)"; numbers without any notations are decimal. For example, "The maximum number of stations is 75. The default memory address is D8000 (hex)."

The term "analyzer" refer to the software application that runs on the Sniffer Network Analyzer.

Other Sources of Information

Network General Corporation (NGC) provides other sources of information that can help you become familiar with the Sniffer Network Analyzer.

On-Line Help

After highlighting an item in the analyzer menu, you can see a phrase or sentence in a panel near the bottom of the screen. It explains the meaning of the highlighted item. To obtain general information about a particular feature of the Sniffer Network Analyzer, press F1 at any time within the menus of the analyzer application. A window containing a list of topics opens.

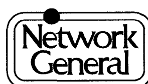
Technical Support

A toll-free number is available to obtain technical support for the Sniffer Network Analyzer.

Phone for Network General's Technical Support Department:	(800) 395-3151
FAX:	(415) 327-9436

CHAPTER ONE: EXTENDING AND CUSTOMIZING PROTOCOL INTERPRETERS

1



Extending and Customizing Protocol Interpreters

Overview

This chapter describes the rules and conventions for writing programs that extend the Sniffer analyzer's ability to interpret protocols.

Network General Corporation (NGC) is constantly expanding the list of available protocol interpreters (PIs). Before writing your own PI, you should check with NGC to see what is currently available.

To use this chapter, you need to be familiar with:

- The general operation of the Sniffer analyzer.
- The C programming language and the MS-DOS programming environment.
- The general structure of network frames and the detailed structure of your protocol.

This chapter is divided into three major sections:

- An overview section that discusses what PIs do and how to install the tools you need to write new PIs.
- A section on writing the C code that performs the necessary PI functions.
- A section on integrating the PI that you have written into the existing Sniffer analyzer code.

You should read this chapter before writing a new PI.

What Does a Protocol Interpreter Do

A PI is a routine (or set of routines) that, when invoked, receives a pointer to data somewhere within a frame. The interpreter has four tasks:

- Generate one or more short text lines to display in the **summary** view for its protocol.
- Generate lines for the **detail** view and highlight the appropriate hex view characters.
- Call the PIs for embedded protocols, if any.
- Supply new symbolic names discovered within the protocol's data.

There are two types of PIs: demultiplexed and embedded. A demultiplexed PI is called directly from the Sniffer analyzer, based on the frame's identifying code. (Depending on the network, that may be its Ethertype, its ARCNET system code, its 802.2 LLC DSAP, and so on.)

An embedded PI is called by another PI to interpret a protocol nested at a higher level. An embedded PI may be shared. It may be called from several other PIs and may be used at different protocol levels.

Writing a Protocol Interpreter

In order to write new protocol interpreters, you need:

- the PI Development Kit from NGC and installed in your Sniffer or in a IBM-PC compatible.
- Version 6.00AX of the Microsoft C compiler from Microsoft Corporation installed with the PI Development Kit.

Porting Current Protocol Interpreters

If you wrote one or more protocol interpreters for previous versions of the Sniffer analyzer and want to re-use them in the new version, refer to this section; otherwise, go to the section "Installing the PI Development Kit" on page 1–5.

To integrate your existing protocol interpreter into the new version of the Sniffer analyzer:

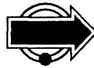
1. Install the PI Development Kit. See the section "Installing the PI Development Kit" on page 1–5.
2. Install version 6.00AX of the Microsoft C compiler. See the section "Installing the Microsoft C Compiler" on page 1–7 for information on suggested options.
3. Move the source code for your PI to the directory where the PI Development Kit is installed.
4. Check that your PI is named `interp_<protocol_name>` and that its data structure is named `pi_data_<protocol_name>`, where `<protocol name>` is the name of the protocol your PI interprets. For more information about this, refer to the section "Registering Protocol Interpreters" on page 1–28.
5. If your protocol interpreter uses the `pi_get_frame` or `pi_invoke_pis` functions, you should be aware that although NGC supports these functions, NGC-written protocol interpreters no longer use them. Read the section "Dependencies on Other Frames" on page 1–24 and modify your protocol interpreter to use the techniques NGC recommends.
6. A sprintf function is delivered with the PI Development Kit. It differs from the sprintf function in the standard C library in a number of ways that simplifies writing protocol interpreters and makes them more robust. NGC recommends that you refer to the section "Sprintf Function in the PI Development Kit" on page 1–17 and use its features.
7. Read the section "Registering Protocol Interpreters" on page 1–28 and modify the new INITPI.C to register your protocol interpreter.
8. Follow the directions in the section "Building a New Sniffer Analyzer" on page 1–31.

Installing the PI Development Kit

You can install the PI Development Kit on your Sniffer analyzer or on a system capable of running version 6.00AX of the Microsoft C compiler. If you chose to install the kit on another system, you will need a 386 or faster CPU with at least 1-Mbyte of extended memory. The advantages of running the PI Development Kit on a system other than your Sniffer analyzer are:

- the compiler and linker may run faster on a different machine.
- the PI Development Kit requires 8 Mb of disk space.

The disadvantage is that you must test your new Sniffer analyzer on the Sniffer analyzer hardware; therefore, you must have some way of moving the linked executable from one machine to another.



Note: If you develop your protocol interpreter on another machine, the linked executable may be larger than a floppy diskette; you may need either a data compression utility or a communications or network connection to transfer the executable.

To install the PI Development Kit:

1. Insert Disk 1 into drive A.
2. Type `a:install` and press Enter.

   The installation process begins and installs all the tools, source files, libraries, and sample files into the *xx*SNIFF\\NEWPI directory, where *xx* is the two-letter network abbreviation.

   If you install to a machine other than the Sniffer, the *xx*SNIFF\\NEWPI directory is created. You can relocate the files to any other directory you chose.
3. Follow the instruction prompts to complete the installation process.

Figure 1–1 lists the files in the PI Development Kit sub-directory.

Files	Description
INITPI1.H INITPI2.H INITPI.C	Files for registering PIs.
MSG.H NETWORK.H PI.H PISWITCH.H PI_LOGIC.H PIFDECL.H	General "include" files.
BUILDSNF.BAT OVLINK.LNK OVLPIMAP.OBJ OVLUTIL.LIB OVPATCH.EXE RTLINK.CFG RTLINK.DAT RTLINK.EXE RTLUTILS.LIB	Tools for building Sniffer analyzer software.
SAMPLE.C	Source code of the sample PI.
SAMPLE.*xx*C	A capture file illustrating the sample PI.
ATALK.H AT_PORTS.C	Files for embedding a PI within AppleTalk.
SMB.H SMBGLOBE.H INTSMBO.C	Files for embedding a PI within SMBs.
TCP.H TCPPORTS.C	Files for embedding a PI within IP, TCP, and UDP.
X25.H X25CALLS.C USERHDLC.C	Files for embedding a PI within X.25 and HDLC.
XWINEXT.C	Files for embedding a PI within XWindows.
SNMP.H NETMGMT.C	Files for embedding new objects in SNMP.
xxSNIFFC.LIB KS.LIB PICODE.LIB PISTRING.LIB RTPI.LIB SN_WFC.LIB XP.LIB	Libraries for building Sniffer analyzer software.

Figure 1–1. PI Development Kit files.

Installing the Microsoft C Compiler

The tools required to write new PIs and build new Sniffer analyzers, except for the Microsoft C compiler, are included in the PI Development Kit. The PI Development Kit only works with version 6.00AX of the Microsoft C Compiler.

Follow the installation instructions provided by Microsoft. The Microsoft installation programs offer a long list of choices to install the compiler. Consider the following:

- Install the DOS or the DOS-and-OS/2 versions (not the Windows version) of the compiler. This is the default option.
- The directory in which you install the "bound executables" must be added to your DOS path so that the compiler can be found.
- Install the LARGE model of the libraries. The default is to install only the SMALL model, which you will not need for PI development.
- Assign the default names to the DOS libraries.
- You do not need the programmer's workbench, mouse support, or on-line documentation.

Coding a Protocol Interpreter

The following sections describe how to write the C code for a protocol interpreter.

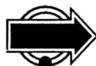
Calling Conventions for Protocol Interpreters

The Sniffer analyzer core code calls PIs using the following calling sequence:

```
int  bytes_interpreted;
char *frame_ptr;         /* Pointer to the frame data */
int frame_length;        /* The length of the frame data starting at frame_ptr */
. . .
bytes_interpreted = interp_protocol (frame_ptr, frame_length);
```

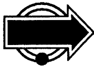
The `frame_ptr` parameter points to the beginning of the data for the protocol within the physical frame. It skips previous protocol fields, including source and destination addresses, type fields, and any previously embedded protocol data.

The Sniffer analyzer permits the user to request that frames be truncated as it captures them. It then records no more than a certain number of bytes for each frame and discards the rest. When the user selects truncation, `frame_length` is the length of the stored (truncated) data, and the global integer `bytes_not_present` indicates how much additional frame data was received but not recorded.



The following rule applies only to Ethernet, StarLAN, and token ring. For a demultiplexed PI that is called for a particular LLC DSAP, `frame_ptr` is the

address of the first byte of the information field. The information field immediately follows the source and destination SAPs and the control field. The SAP protocol interpreter is called for any frame with an information field, such as I, UI, or XID. It is not called for frames that do not contain information, such as RR or that do not contain higher-level protocol information, such as TEST or FRMR.



The following rule applies only to Ethernet, StarLAN, and PC Network. For a demultiplexed PI that is called for a particular Ethertype, `frame_ptr` is the address of the first byte of the information field. The information field immediately follows the 2-byte Ethertype.

The value returned from the PI is the number of bytes of frame data that it interpreted. The calling interpreter may use the returned value to decide whether any subsequent interpreters are to be called. If there is no more data left to interpret, simply return the initial value of `frame_length`.

If the PI needs access to DLC or LLC header fields, or to the frame number, it may refer to the values of certain global static variables in which these are recorded. They are listed in Figure 1–2.

Global Static Variable	Purpose
`long pi_frame;`	The current frame number.
`char *dlc_header;`	Pointer to DLC header starting with the first byte of the frame.
For 802.3 networks, Ethernet, StarLAN, and Token ring:	
`char *llc_header;`	A pointer to the LLC header of the frame, starting with the DSAP field.
`int llc_type;`	The type of the LLC frame; see the llc_xxx macros in pi.h.

Figure 1–2. Global static variables available to the protocol interpreter.

Any PI your PI calls to interpret embedded protocols should be invoked using the same calling convention.

Protocol Interpreter Data Structure

This structure is set up during the protocol interpreter registration process described later in this chapter. You should declare the data structure as an external in your protocol interpreter:

```
extern struct *pi_data pi_data_your_protocol;
```

The complete definition of this structure is contained in the file PI.H. The fields described below are the only ones relevant to your protocol interpreters. Each

field below is a "boolean," in which the value 0 indicates "false" and 1 indicates "true."

```
struct  pi_data {
int      do_sum;       Generate summary lines?
int      do_int;       Generate detail interpretation lines?
int      do_count;     Generate only count of summary lines?
int      do_names;     Add symbolic station names?
int      recursive;    Recursive call to get information for
                       another frame?
```

If do_sum is true, your protocol interpreter must generate a **summary** line. If do_count is also true, then the Sniffer analyzer only needs a count of summary lines. Your protocol interpreter must allocate a summary line but need not actually write to it.

If do_int is true, your protocol interpreter must generate one or more **detail** lines. Note that if both summary and detail views are enabled, both do_sum and do_int are true at the same time.

If do_names is true, the user has selected the **Search for names** menu option. Examine the frame data for embedded station names defined by the protocol. If any are found, call the add_station_name function, described in the section "Adding Symbolic Names to the Name Table" on page 1–20 to enter the names into the name table.

The recursive flag is supported only for compatibility with previously written protocol interpreters. It is still supported, but it should not be used in new protocol interpreters.

Your PI should call interpreters for any embedded protocols regardless of the flag settings.

Generating Output from Protocol Interpreters

Protocol interpreters generate output by asking the Sniffer analyzer core code for a buffer to write a summary or a detail line and then writing a string into that buffer. The following sections discuss enhancements to the sprintf function that allow you to generate output and a series of "PIF routines" that do most of the work.

To generate a line for the **summary** view from within a PI, get the address of a line buffer by calling get_sum_line:

```
char  *get_sum_line (pi_data_your_protocol); Returns a pointer to the line buffer
struct pi_data  *pi_data_your_protocol;      Your  protocol interpreter data structure
```

Then move a character string (ending with a null) into the buffer provided using sprintf, strcpy, strcat or any other mechanism available in C. The length of the

string including the null must not exceed MAX_SUM_LINE. For visual consistency of the displayed output, the summary line should begin with a 3-character identification of the protocol and a blank.

Generating a line for the detail view is similar. However, the function to get the address of a line buffer also provides an optional offset and length that show where in the frame the information was found. This is used to highlight the hex field when the user selects the corresponding line of the detail view.

```
char *get_int_line (pi_data_your_protocol, offset, length); /*Returns a pointer to the line */
                                                            /*buffer */
struct pi_data *pi_data_your_protocol;                      /*Your PI's data structure. */
```

`int offset;` The offset from the DLC header of the field that generated the interpretation line. You can calculate this using the global variable dlc_header that points to the DLC header as shown in Figure 1–2, or you can use pif_offset, another global variable, discussed in the next section.

int length; The length of the field, or 0 if no highlighting is desired.

The string should be built in the supplied buffer. Its length, including the final null, must not exceed MAX_INT_LINE.

The version of sprintf included with the PI Development Kit will never write beyond the end of summary or detail line buffers. If you only use sprintf to write to these lines, you never need to check the length of your lines.

In general, the Sniffer analyzer automatically scrolls the **detail** view so that the top line is the first line for the protocol that the user selected in the **summary** view. If you wish some other line of your protocol's **detail** lines to scroll to the top, you may specify a negative offset when calling get_int_line.

Your protocol interpreter should generate summary lines only when do_sum is set to true in the PI data structure and should generate detail lines only when do_int is set to true. An error occurs if this is not done.

The Protocol Interpreter Formatting Routines

PIs may be written with no functions used other than those already described and those available in the standard C library. For fixed-format data, the simplest approach is often to write structure declarations and simply format summary and detail lines using the sprintf function in the standard C library.

When data contains many variable-length or optional fields, however, using C-language structures becomes awkward. To address this problem, NGC provides a series of stream-oriented formatting utilities:

- Protocol interpreter formatting (PIF) routines discussed in this section.
- Sprintf function discussed in the section "Sprintf Function in the PI Development Kit" on page 1–17.

The use of PIF routines and the sprintf function is optional. You may write interpreters without them if you choose.

A PI that wants to use the PIF routines or the sprintf function makes one call to the initialization function pif_init each time it is called for a new frame. The PIF routines then maintain an offset from the DLC header, called the PIF offset, which is used to extract data in various forms. The PIF offset is stored in the global variable pif_offset. You can use pif_offset as the offset to pass to get_int_line if it is convenient. The offset is declared in the file PI.H.

There are three general classes of PIF routines:

- Routines that return a data item to the caller begin with pif_get_. The PIF offset is not updated. These routines may be used to extract data for either the **summary** line or the **detail** view.
- Routines that display a data item on a line in the detail view begin with pif_show_. The PIF offset is incremented by the length of the data item and thus points to the next item.
- Other miscellaneous PIF functions.

Figure 1–3 presents a summary of the available PIF routines.

pif_init	Initialize PIF global variables.
pif_save	Save PIF info before calling an embedded PI.
pif_restore	Restore PIF info after calling an embedded PI.
pif_get_byte	Get value of a single byte.
pif_get_word	Get value of 2-byte word in low-high order.
pif_get_word_hl	Get value of 2-byte word in high-low order.
pif_get_long	Get value of 4-byte word in low-high order.
pif_get_long_hl	Get value of 4-byte word in high-low order.
pif_get_ascii	Get ASCII characters into a C string.
pif_get_ebcdic	Get EBCDIC characters into a C string.
pif_get_lstring	Get a length/string into a C string.
pif_get_addr	Get the address of the current field.
pif_show_byte	Display a single byte.
pif_show_word	Display 2-byte word in low-high order.
pif_show_word_hl	Display 2-byte word in high-low order.
pif_show_long	Display 4-byte word in low-high order.
pif_show_long_hl	Display 4-byte word in high-low order.
pif_show_2byte	Display 2 one-byte fields.
pif_show_4byte	Display 4 one-byte fields.
pif_show_6byte	Display 6 one-byte fields.
pif_show_nbytes_hex	Display an n-byte field in hexadecimal.
pif_show_ascii	Display a string of ASCII characters.
pif_show_ebcdic	Display a string of EBCDIC characters.
pif_show_lstring	Display a length/string of ASCII characters.
pif_show_flag	Initialize to display bits of a flag byte.
pif_show_flagbit	Display flag bit values.
pif_show_flagmask	Display flag bit value if it matches the mask.
pif_show_date	Display a date and a time.
pif_show_space	Display a blank line
pif_header	Display a **detail** view header message.
pif_trailer	Display a **detail** view trailer message.
pif_autoscroll	Set **detail** view autoscroll point to be the next header.
pif_line	Return a **detail** view line buffer and advance the pointer.
pif_set	Set the PIF offset.
pif_skip	Move the PIF offset backwards or forwards.

Figure 1–3. PIF Routines.

Calling the PIF Routines

This section describes the calling sequences of the functions listed in Figure 1–3.

The following initializes the PIF variables. pif_init must be called by the PI before any other pif_xxx routines or the sprintf function can be used.

```
void    pif_init (pi_data_your_protocol, p, len)   /* Initialize PIF globals */
struct  pi_data *pi_data_your_protocol;  /* Protocol interpreter's control block ptr */
void    *p;                              /* Start of frame data to interpret        */
int     len;                             /* Length of frame data                    */
```

The following saves the PIF variables before calling an embedded PI. The caller supplies a "pif_info" area (defined in the file PI.H) into which the current state information is saved. The pif_restore routine restores the state information.

```
void    pif_save (&pd)          /* save pif variables                    */
struct  pif_info  pd;           /* Area used to save pif state           */
```

The following restores the PIF variables. The caller supplied a "pif_info" area that was previously supplied to pif_save.

```
void    pif_restore (&pd)        /* restore pif variables           */
struct  pif_info pd;             /* Area used to restore pif state  */
```

It is necessary to use pif_save and pif_restore only if you are calling embedded PIs and you wish to generate more **detail** view lines after the embedded interpreter has returned.

The sequence below returns a byte, word (2 bytes), or longword (4 bytes) from the frame data at the current PIF offset plus the (signed) value of delta. The PIF offset is not changed. The "_hl" versions are for multibyte fields stored with the most significant byte first. These are macros defined in the file PI.H.

```
char    pif_get_byte (delta)
int     pif_get_word (delta)
int     pif_get_word_hl (delta)
long    pif_get_long (delta)
long    pif_get_long_hl (delta)
int     delta;
```

This sequence moves a printable ASCII null-terminated string at the current offset in the frame to a C string. Unprintable characters are replaced with the character ".". The destination string ends with a "null" even when the source does not. The returned value is a pointer to the end (null) of the destination string. The PIF offset is not changed.

```
char   *pif_get_ascii (offset, len, result_str)   /* get asciiz string    */
int    offset;                /* offset to string from current pif offset */
int    len;                   /* maximum number of source bytes           */
char   result_str[];          /* destination string                       */
```

The following translates a printable EBCDIC null-terminated string at the current offset in the packet into ASCII and moves it to a C string. Unprintable characters are replaced with the character ".". The destination string ends with a "null" even when the source does not. The returned value is a pointer to the end (null) of the destination string. The PIF offset is not changed. If the user forces ASCII display, this calls pif_get_ascii instead.

```
char    *pif_get_ebcdic (offset, len, result_str)   /* Get ebcdic string */
int     offset;               /* offset to string from current pif offset */
int     len;                  /* maximum number of source bytes           */
char    result_str[];         /* destination string                       */
```

The sequence below moves an ASCII null-terminated lstring from the current offset in the packet to a C string. An lstring starts with a length byte followed by that number of characters. Unprintable characters are replaced with the character ".". The destination string ends with a "null" even when the source does not. The returned value is a pointer to the end (null) of the destination string.

```
char    *pif_get_lstring (offset, result_str)     /* Get Lstring        */
int     offset;               /* Signed offset from current position    */
char    result_str[];         /* Return ASCIIz string here              */
```

The following line returns a pointer to the field that is at the current PIF offset. This is a macro defined in the file PI.H.

```
char    *pif_get_addr ()        /* return the data address                 */
```

The following sequence creates new lines in the **detail** view with the text given in prstr and the indicated data from the frame. The text should contain a sprintf formatting code, such as %d or %x indicating where the value should be printed.

For the longword displays, the formatting code should be %ld or %lx. For pif_show_2byte, pif_show_4byte, and pif_show_6byte there should be 2, 4, and 6 formatting codes, respectively. For pif_show_nbytes_hex, there should be a single %s formatting code. The value of byte_count specifies the number of bytes to display, from 1 to 99. The PIF offset is updated. The byte, word, and long routines return the displayed value.

```
char    pif_show_byte (prstr)
int     pif_show_word (prstr)
int     pif_show_word_hl( prstr)
long    pif_show_long (prstr)
long    pif_show_long_hl (prstr)
void    pif_show_2byte (prstr)
void    pif_show_4byte (prstr)
void    pif_show_6byte (prstr)
void    pif_show_nbytes_hex (prstr, byte_count)
char    *prstr;               /* A sprintf control string             */
int     n;
```

The following creates a new detail line from ASCII text starting at the current offset. The caller provides a sprintf control string whose embedded %s is replaced with the ASCII string copied from frame data using pif_get_ascii. The PIF offset is updated.

```
void    pif_show_ascii (len, prstr)   /* Show ASCII text                      */
int     len;                          /* Number of bytes to display           */
char    *prstr;                       /* control string with embedded %s      */
```

The following creates a new detail line from EBCDIC text, starting at the current offset. The caller provides a sprintf control string whose embedded %s is replaced with the EBCDIC string copied from frame data using pif_get_ebcdic. (If you force ASCII translation, this calls pif_show_ascii instead.) The PIF offset is updated.

```
void    pif_show_ebcdic (len, prstr)   /* show ebcdic text                     */
int     len;                           /* number of bytes to display           */
char    *prstr;                        /* control string with embedded %s      */
```

The following creates a new detail line from an ASCII lstring, starting at the current offset. An lstring starts with a length byte followed by that number of characters. The caller provides a sprintf control string whose embedded %s is replaced with the string copied from frame data using pif_get_lstring. The PIF offset is updated.

```
void    pif_show_lstring (prstr)    /* show lstring                         */
char    *prstr;                     /* control string with an embedded %s   */
```

The routine below displays the value of a byte with bit flags and sets up the correct information for subsequent calls to show_flagbit and show_flagmask. The PIF offset is incremented by 1.

```
void    pif_show_flag (prstr, mask)         /* show flag byte              */
char    *prstr;          /* title string: " = %d" is automatically added   */
char    mask;            /* mask value indicating which bits to display    */
```

The routine below writes a field in the form "....1... <string>," indented as appropriate for the previous pif_show_flag call. If the falsestr is NULLP, the truestr is used for both cases. This returns TRUE if any of the specified bits were on.

```
boolean pif_show_flagbit (bit, truestr, falsestr)     /* show flag bits */
char    bit;                /* Bit mask for 1 or more bits              */
char    truestr[];          /* string to show if any masked bits are on */
char    falsestr[];         /* string to show if all bits are off       */

See Also
pif_flag_w (char*, ushort)
pif_flag_w_hl (char*, ushort)
```

The sequence below writes a **detail** line for a bit field only if the masked bits are a specified value. The line is written in the same format as for pif_show_flagbit. This returns TRUE if the flag bits were the specified value and the line was written.

```
boolean pif_show_flagmask (maskbits, value, prstr) /* conditional show flags  */
char    maskbits;                      /* Only check these bits              */
int     value;                         /* Check for this value               */
char    *prstr;                        /* Write this string if matched       */
See also
pif_flagbit_w
pif_flagbit_w_hl
```

The following creates a new detail line from the text given in prstr, with the %s replaced with a readable date and time such as "13-May-90 11:47:13". The date and time are taken from a 4-byte integer at the current PIF offset representing the number of seconds since 1/1/90 at midnight. The integer should be stored with the most significant byte first for pif_show_date_hl and with the least significant byte first for pif_show_date. The PIF offset is incremented by 4.

```
void  pif_show_date (prstr)               /* show Unix-style date            */
void  pif_show_date_hl (prstr)            /* show Unix-style date            */
char  *prstr;                             /* control string with embedded %s */
```

This call writes a blank line to the **detail** view.

```
void    pif_show_space ()                 /* display a blank line            */
```

The following outputs a header line to the **detail** view in the format "----- header_text -----", followed by a blank line, and highlights data starting at the current PIF offset for the length specified. This routine does not update the PIF offset but saves the header string in the global called header_msg so other routines can use it.

```
void    pif_header (len, header_string)    /* Write a header line           */
int      len;                              /* Length of area to highlight   */
char    *header_string;                    /* Header string                 */
```

This call outputs a trailer line to the **detail** view that reports on how much of the frame data was used by the interpreter, based on the final position of the PIF offset. This routine uses the header string saved by pif_header.

```
void   pif_trailer ()                       /* write a trailer line         */
```

The call below makes the next header line written to the **detail** view with pif_header be the one that is scrolled to the top of the detail view when the user highlights the PI's summary line. This is necessary only if the next header line is not the first for this PI.

```
void  pif_autoscroll ()                     /* set autoscroll position      */
```

The following calls get_int_line to return a pointer to a **detail** line area. The caller passes a length that causes the specified number of bytes (at the current

PIF offset) to be highlighted in the hex view. The PIF offset is advanced by the number of bytes specified. Possible side effects may result if you use this as the first argument to sprintf; no other arguments should depend on the PIF offset.

```
char    *pif_line (len)      /* get detail line pointer                  */
int     len;                 /* Number of bytes to highlight and advance */
```

The following sets the PIF offset to point to the specified address. This is a macro defined in the file PI.H.

```
void    pif_set (address)       /* set the PIF offset                   */
char    *address;
```

The following adds the signed delta to the current PIF offset. This is a macro defined in the file PI.H.

```
void    pif_skip (delta)        /* move the PIF offset                  */
int     delta;
```

The call below returns a pointer to the data item at the current PIF offset. This is a macro defined in the file PI.H.

```
char    *pif_get_addr ()        /* return frame data address             */
```

Sprintf Function in the PI Development Kit

The sprintf function delivered with the PI Development Kit ("PDK sprintf") has a number of enhancements. This section assumes that you are familiar with the standard sprintf routine.

PDK sprintf will not write beyond the end of detail and summary lines. If PDK sprintf comes to the end of a summary or detail line and has more to print, it ends the line with an ellipsis and sets the value of the global variable `sprintf_error` to `LINE_OVERRUN`. (If this has not happened, then the value of `sprintf_error` is FALSE.) If you use the PDK sprintf routine to write to summary and detail lines, you need not concern yourself with overrunning these lines.

PDK sprintf understands the following format conversions:

```
                                                            {d}
                                                            {b}
      {<}                                                   {X}
% [~] [{^}] [-] [+] [ ] [#] [0] [{n}] [.{n}] [r] [{l}]      {c}
      {>}                         {*}    {*}       {k}      {s}
                                                            {u}
                                                            {o}
                                                            {x}
```

The flags must appear in the order shown above. The flags and format conversions are listed in Figure 1–4, along with an indication of whether each is

new with PDK sprintf or whether it is standard. The new PDK flags are discussed in the sections that follow.

~	New	Do not print the format conversion.
<	New	Print a value from the frame; move the PIF offset back
^	New	Print a value from the frame; leave the PIF offset unchanged
>	New	Print a value from the frame; advance the PIF offset
-	Std	Left justify in the print field
+	Std	Precede a positive numeric value with a +
<sp>	Std	Precede a positive numeric value with a space
#	Std	Precede hexadecimal and octal values with 0x and 0, respectively
0	Std	Fill leading areas with 0's rather than spaces
<no>	Std	"Width"
<no>	Std	"Precision"
r	New	The value to print is in high-to-low byte order.
l	Std	The value to print is four bytes long.
k	New	The value to print is one byte long.
u	Std	Unsigned value
o	Std	Octal value
x	Std	Hexadecimal value (use lower case a...f)
d	Std	Decimal value
b	Std	Binary value
X	Std	Hexadecimal value (use upper case A...F)
c	Std	Character
s	Std	Null-terminated string

Figure 1–4. PDK flags.

Sprintf: Printing Values From the Frame (<, ^, and > Flags)

The ^, >, and < flags tell PDK sprintf to print a value from the location in the captured frame pointed to by the PIF offset (rather than from its own parameters). See the section "The Protocol Interpreter Formatting Routines" on page 1–10 for information about the PIF offset. This saves copying data from the frame to temporary variables before calling sprintf or using pointers into the frame.

Sprintf checks that values fetched from the frame were actually captured. If the PIF offset points off the end of a captured frame, then PDK sprintf prints "--- Frame too short" in place of the formatted value. If this happens, sprintf returns to the Sniffer analyzer core code instead of to your PI. You must call pif_init to initialize the PIF offset before using these flags.

The > flag reads a value from the frame and advances the PIF offset (similar to a pif_get_ routine). For example:

```
summary_line += sprintf (summary_line, "NR=%>d");
```

reads two bytes from the frame, prints the integer value represented by the two bytes, and adds two bytes to the PIF offset. If NR, NS, and Length follow one another in the frame, then

```
summary_line += sprintf (summary_line, "NR=%>d NS=%>d Len=%>d");
```

prints each of them. The width and precision specifiers are ignored for the purposes of advancing the PIF offset.

The ^ flag prints the value from the location in the frame but does not change the value of the PIF offset. This is useful for printing a value more than once. For example:

```
detail_line += sprintf (detail_line, "User key = %^d (%>04x)");
```

prints a line such as

```
User key = 79 (004f)
```

Since the %^d format does not advance the PIF offset, the %>04x format prints the same value in the different format.

The < flag prints the value from the location in the frame and then moves the PIF offset back through the frame. For example, if NS appears before NR in a frame, you can position the PIF offset to point at NR and then use this statement:

```
summary_line += sprintf (summary_line, "NR=%<d NS=%^d");
```

Sprintf: Skipping a Value in the Frame (~ Flag)

The ~ flag causes PDK sprintf not to print the output from the format specification. This is equivalent to the pif_skip function. For example:

```
summary_line += sprintf (summary_line, "NR=%>d %~>d NS=%^d");
```

prints the first two-byte integer in the frame, skips the second, and then prints the third, and

```
sprintf (summary_line, "%~>s");
```

skips past the null that terminates a string without printing the string.

Sprintf: Controlling Numeric Values (r and k Flags)

The k ("kharacter") flag tells PDK sprintf that the next value is only a byte. For example:

```
summary_line += sprintf (summary_line,
        "Dst port=%>02kx Src port = %>02kx");
```

prints port numbers from the frame that are only one byte long.

The r ("reverse") flag tells PDK sprintf that the bytes in a two- or four-byte value are in "reverse" order (high-order byte first, low-order byte last, the reverse of the normal Intel format):

```
sprintf (detail_line, "Network number = %^08rlx");
```

prints a network number for which the high byte was transmitted first, and the low byte was transmitted last.

Adding Symbolic Names to the Name Table

When your PI is called and do_names in the PI data structure is true, you must examine the frame for an embedded name. If you find one, add it and its symbolic equivalent to the station name table by calling the function add_station_name, as follows:

```
add_station_name (flagstype, length, addr, name)
```

`int flagstype;` The type of the address plus option flags; see PI.H for definitions. *Example:* ADDRTYPE_DLC plus ASN_NOREPL indicates that the address is a data-link level address and that its symbolic name is not to replace a symbolic equivalent already assigned to that address in the name table.

`int length;` The length of the address, from 1 to 16 bytes.

`char *address;` A pointer to the binary station address for this name.

`char *name;` A pointer to the ASCII name to enter in the table; 1 to 31 characters (including a final null).

Since names discovered in the protocol data may be transitory, you should use ASN_NOREPL. In this way, when the names file already supplied a symbolic equivalent for the address, the name you find does not replace it.

Declaring Embedded Addresses

Each frame has source and destination DLC-level addresses associated with it by the Sniffer analyzer. If a PI knows of other addresses associated with this frame, such as Internet addresses embedded within the frame data, it may announce those addresses by calling the following routine:

```
add_frame_addr (flagstype, length, addr)
```

`int flagstype;` The type of the address plus AFA_XXX option flags; see PI.H for definitions. *Example:* ADDRTYPE_IP plus AFA_SRC for a TCP/IP internet source address of this frame.

`int length;` — The length of the address, from 1 to 16 bytes. *Example*: 4 for TCP/IP internet addresses.

`char *address;` — A pointer to the binary address.

Displaying Symbolic Names

If you want to include the symbolic name corresponding to an address in the text of a summary or detail line, you may use the following routine to look up and format names from the current name table:

```
char *format_addr (line, length, addr, flagstype)
```

`char *line;` — The address of a character buffer into which the formatted name is placed. The function return value is the address of the null at the end of the string.

`int length;` — The length of the address, from 1 to 16 bytes. Example: 6 for DLC-level addresses.

`char *address;` — A pointer to the binary address to be looked up in the name table.

`int flagstype;` — The type of the address plus FMT_XXX option flags; see PI.H for definitions. *Example*: ADDRTYPE_DLC plus FMT_BOTH for a DLC address to be formatted with both the hex value and the symbolic name.

Adding Summary Line Flags

If you want to create special flags and to have the Sniffer analyzer flag frames and display the flags in the **summary** view, you can call the following function:

```
void add_frame_flags(str)
```

`char *str;` — Characters to add to the flags

At most, six flag characters can be displayed for each frame. (See the section "About the Flags Display Option," in the *Sniffer Network Analyzer Operations Manual*.)

Using Other Protocol Interpreters

In general, any protocol interpreter can call any other by using the calling sequence described in the section "Calling Conventions for Protocol Interpreters" on page 1–7. The calling protocol interpreter must supply a pointer to the beginning of the embedded protocol's data and the number of bytes remaining in the frame from that point.

In most cases, the PI that you write will be at the top of the protocol stack, and the only other protocol interpreters you call will be your own. In this case, you can use any calling sequence you want.

You also may call an existing NGC-written protocol interpreter if your protocol interpreter discovers a protocol that it interprets as embedded within your protocol. To find the list of NGC-written protocol interpreters, you can examine the REGISTER lines in the file INITPI.C. You can force any protocol interpreter whose registration includes the flag PITYPE_FORCEE using the pointer to that function. For example, since the line:

```
REGISTER (SMB, smb, "SMB", "SMB: ",
             PITYPE_EMBEDDED+PITYPE_FORCEE+L7, 0, &no_demux)
```

contains the PITYPE_FORCEE flag, you can call the smb protocol interpreter as follows:

```
if (piptr_smb)
   smb_bytes_interpreted = piptr_smb (frame_ptr, frame_length);
else             . . .
```

Testing that piptr_smb is not NULLP ensures that this protocol interpreter has really been registered at run time. The name of this pointer is always "piptr_" followed by the second parameter on the REGISTER line in INITPI.C. What your protocol interpreter does if the embedded protocol has not been registered is up to you.

An Example

Consider the "sample" protocol that has simple fixed-format data:

[1 byte]	Device number
[1 byte]	Command type
[2 bytes]	Number of segments
[20 bytes]	Name of owner (null-terminated ASCII)

In a new module, you would write your PI with a structure similar to Figure 1–5.

Several frames with this sample protocol embedded over SAP 0x91 (or ARCNET system ID 0x91) are in the capture file SAMPLE.xxC that is delivered with the PI Development Kit. You can build a Sniffer analyzer that contains the sample PI and interpret these frames by following the directions in the section "Building a New Sniffer Analyzer" on page 1–31 to integrate the sample PI over a SAP or over an ARCNET system ID.

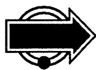
```
#define  USE_PIF   1             /* should we use the PIF routines? */

#include "pi.h"

extern struct pi_data *pi_data_sample;  /* our PI data */

struct  sample_header {                 /* the format of our frame data */
          char  device;
          char  command;
          int   nsegments;
          char  owner [20];
          };

int interp_sample (header, length)      /* our sample interpreter */

struct sample_header  *header;          /* pointer to our protocol header */
int                   length;           /* length of the remaining data */

{
    if (pi_data_sample->do_sum) {       /* summary line wanted? */
        sprintf (
            get_sum_line (pi_data_sample),      /* get a line buffer */
                "SMP Device = %d, Cmd = %02X",
                header->device, header->command);
        } /* end of summary line */

    if (pi_data_sample->do_int) {       /* detail lines wanted? */

#if USE_PIF                             /* show how to use PIF routines */

        /* Set up PIF globals */

        pif_init (pi_data_sample, header, length);

        /* Output the header line and highlight the whole header.
           Note that pif_header() does not alter the PIF offset. */

        pif_header (sizeof (struct sample_header),
            "Sample protocol data area");

        /* Display the fields.  Each routine advances the buffer pointer
           past the data item just displayed. */

        pif_show_byte ("Device number = %d");
        pif_show_byte ("Command type = %02X");
        pif_show_word ("Number of segments = %d");
        pif_show_ascii (20, "Owner = %s");

        /* Write out "End of. ." message & check for excess or missing data */

        pif_trailer ();

#else                                   /* Do detail without PIF routines */
```

Figure 1–5. Sketch for structure of a new protocol interpreter (continued on next page).

```
        sprintf (
            get_int_line (pi_data_sample,              /* get a line buffer */
              (char *)&header->device - dlc_header,    /* highlight offset */
              2),                                      /* highlight length */
            "Device number = %d, command type = %02X",
            header->device, header->command);
        sprintf (
            get_int_line (pi_data_sample,              /* get a line buffer */
              (char *)&header->nsegments - dlc_header, /* highlight offset */
              2),                                      /* highlight length */
            "Number of segments = %d",
            header->nsegments);

        sprintf (
            get_int_line (pi_data_sample,              /* get a line buffer */
                (char *)header->owner - dlc_header,    /* highlight offset */
                20),                                   /* highlight length */
            "Owner = %20s",
            header->owner);
#endif

        } /* end of detail lines */

        /* If there were any embedded protocol after our header,
           we could call the interpreters here.  */

        return length;               /* say that we used up all the data */
        }

/* end of sample.c */
```

Figure 1-5. Sketch for structure of a new protocol interpreter (continued from previous page).

Dependencies on Other Frames



Note: Although the `pi_get_frame` and `pi_invoke_pis` functions are still available for compatibility with protocol interpreters written for previous versions of the Sniffer analyzer, they are no longer documented, as NGC does not encourage their use. The mechanisms NGC recommends for dealing with dependencies on other frames are discussed in this section.

In many protocols, the interpretation of a frame depends upon information found in other frames. Here are two examples:

- When a protocol interpreter sees a frame containing a call to a server, the details of the call may be required to interpret the subsequent return frame.
- When a protocol interpreter sees a client port communicating with a well-known server port or obtains some other clue about what protocols are embedded in frames to and from that client port, this information can help the PI interpret subsequent frames from or to the client port.

After a new capture, when a new data file is loaded, and when the user chooses the "Re-interpret" option on the display menu, the Sniffer analyzer performs a "prescan" during which it interprets all the frames in order. After the prescan,

your PI may be called to interpret the frames in an arbitrary order, depending upon what the user requests to display. The prescan is an opportunity for your protocol interpreter to save any information it may need to be able to interpret the frames later out of order.

The global variable do_prescan is TRUE when the frames are being interpreted in order for the first time and FALSE after this first pass has been completed. When do_prescan is FALSE, protocol interpreters should rely entirely upon the data in the frames and whatever data they stored when prescan was TRUE.

The Sniffer analyzer maintains 16 bytes of "context" with each frame, a permanent data structure attached to the frame. During the prescan, your protocol interpreter should store enough data in the context that it can interpret the frame later without reference to other frames. Since the 16 bytes of context must be shared by all the protocol interpreters working on a frame, you should store as little data as possible.

The Sniffer analyzer core code contains two library functions, one to add data to a frame's context and one to retrieve that data:

```
typedef char * CONTEXT_DATA;
CONTEXT_DATA *allocate_context_data (int pi_number, unsigned context_length)
CONTEXT_DATA *fetch_context_data (int pi_number)
```

The pi_number parameter for both of these functions is a number to identify your protocol interpreter. It is a number between 1 and 31, and it must be different from the numbers used by other protocol interpreters. The file PI.H contains definitions of constants named "CTAG_" followed by a protocol name, which are the pi_numbers that NGC has used for its protocol interpreters. You may use any other pi_numbers for your protocol interpreters.

allocate_context_data returns a pointer to memory allocated for your frame context for the frame currently being interpreted. The buffer's length will equal context_length. If previous context data was stored for this frame with this pi_number, it is deleted by this call. If there is not enough space in the context to satisfy the request, this function returns NULLP. You may write any data you like into the context memory area.

fetch_context_data returns a pointer to the data previously stored by your protocol interpreter for the frame currently being interpreted. If no context data has been stored for the current frame, then this function returns NULLP. It is up to you to interpret the data in the context.

An example of the structure of a protocol interpreter that depends upon other frames is shown in Figure 1–6.

```
/* Protocol interpreter that uses the context */
      /* This array of structures is used during the prescan to store temporary data */
      static struct {
            . . .
      } prescan_data [...];

      /* This variable tells when data should be captured for prescan_data */
      extern int do_prescan;

      /* This is the PI data structure */
      extern struct pi_data* pi_data_user_protocol;

      int interp_user_protocol (frame_ptr, frame_length)

      char *frame_ptr;              /* Frame to interpret */
      int frame_length;             /* Bytes in frame to interpret */

      {
      CONTEXT_DATA *context;

      if (do_prescan)
            {
            /* This is the chance to save data in the array */
            if (<this frame has information we should save>)
                  {
                  <Add information to prescan_data>
                  }
            if (<data in the prescan_data array contains information
                  that will be useful later in interpreting this frame>)
                  {
                  if (context = allocate_context_data (CTYPE_USERPI, <length>)
                        {
                        /* Got the context buffer */
                        <Copy data for later use into context>;
                        }
                  if (<the data item in the prescan_data array will
                        no longer be useful>)
                        <Delete the item in the prescan_data array>;
                  }

            }
      else
            /* The prescan is over. Therefore, the prescan data is */
            /* no longer valid; erase it. */
            /* <Note that prescan_data has no valid data>; */

      if (pi_data_user_protocol->do_sum)
            {
            context = fetch_context_data (CTYPE_USERPI);
            <If context is not NULLP, use the information to help
            write out the summary line.>
            . . .
            }

      if (pi_data_user_protocol->do_int)
            {
            context = fetch_context_data (CTYPE_USERPI);

            <If context is not NULLP, use the information to help
            write out the detail lines.>
            . . .
            }

      }
```

Figure 1–6. Using the Frame Context.

Integrating into Existing Sniffer Analyzer Code

In order for your PI to be of any use, the Sniffer analyzer core code must call your PI at the appropriate times.

- If you have written a protocol interpreter for a new demultiplexed protocol (one that is identified by an Ethertype, DSAP or ARCNET system ID), then registering the protocol as described in the next section will cause your PI to be called whenever your protocol's Ethertype, DSAP or ARCNET system ID appears in a frame.
- If you want to extend the interpretation of Token Ring NetBIOS frames, you can insert the code for your protocol interpreter into the module INTNETBO.C, which is provided in the PI Development Kit. The routine `interp_netbios_other` is called whenever the NetBIOS interpreter finds a frame that is not an SMB frame.
- To extend the Xwindows protocol interpreter, you can add code to the stub functions in the file XWINEXT.C. These routines are called whenever the Xwindows protocol interpreter discovers information that it cannot interpret. Instructions for modifying these routines is found in the comments in the source code.
- To extend the SMB protocol interpreter, you can write new code into the stub interpreter contained in the file INTSMBO.C. This PI is called whenever the standard SMB protocol interpreter discovers information that it cannot interpret. This module contains a real protocol interpreter, and you can write code into it just as you might write a new protocol interpreter. You should also consider registering the `smb_other` protocol interpreter in the file INITPI.C.
- If you have written a protocol interpreter for a new protocol running over AppleTalk's DDP, then adding your protocol's DDP type to the table that the existing DDP protocol interpreter uses will cause your PI to be called whenever your protocol's DDP type appears in a frame. The table is contained in the module AT_PORTS.C, which is provided in the PI Development Kit. Sample additions have been made under the constant `USERPI_OVER_DDP`.
- If you have written a protocol interpreter for a new protocol running over TCP, IP, or UDP, then adding your protocol's port or number to the tables that the existing TCP, IP, and UDP protocol interpreters use will cause your PI to be called appropriately.

  The tables are contained in the module TCPPORTS.C, which is provided in the PI Development Kit. Sample additions have been made to this file under the constants `USERPI_OVER_IP`, `USERPI_OVER_TCP`, and `USERPI_OVER_UDP`. The first two parameters in the TCP table are the minimum and maximum port; for a customer-written PI the last three parameters should always be "&piptr_none, 0, FALSE."

- If you have written a protocol interpreter for a new protocol running over X.25, then adding your protocol to the table in the file X25CALLS.C will cause your PI to be called whenever your protocol's identification number appears in a frame. Sample additions have been made under the constant USERPI_OVER_X25. The definition of the structure in X25CALLS.C is in the file X25.H.
- If you have written a protocol interpreter for a new protocol running over HDLC or HDLC framing, then adding your protocol to the table in the file USERHDLC.C will cause your PI to be called.
- If you want to replace or partly replace a NGC- written demultiplexed protocol, you can arrange that the Sniffer analyzer core code calls your PI instead of NGC's. To do this, find the registration line for NGC's PI. See the section, "Registering Protocol Interpreters" on page 1–28. Change the registration line to register your PI instead.

  If you want, your new PI can interpret only certain kinds of frames. You can re-register NGC's PI as an embedded protocol and call it in situations where you want the NGC PI to interpret the frame. You can also split the work in other ways. For example, your PI can create the summary line, then set do_sum to FALSE in the NGC PI's data structure, and call the NGC PI to produce detail lines and call further embedded PIs. See the example in the section "Example: Changing the TCP Summary Line" on page 1–32.

Registering Protocol Interpreters

Each PI should be registered in the file INITPI.C. Registration serves two purposes:

- For a demultiplexed PI, registration tells the Sniffer analyzer when it is appropriate to call that PI. Registration is required for a demultiplexed PI; otherwise, the PI is not called.
- For any PI, registration causes the name of the PI to appear on the display filter menu so that the user can select frames that contain your protocol. You can choose to have your protocol to appear on the protocol forcing rules menu if you wish.

All registration occurs in the function INITPI.C, which is called when the Sniffer analyzer is initialized.

The macro that registers a PI appears as follows:

```
REGISTER (DEFAULT, prot_name, menu_name, pi_det, pi_type, ntypes, types)
```

This macro **must not** end with a semi-colon.

The function that registers a PI has the following calling sequence:

`DEFAULT`
A flag to indicate the protocol suite to which this protocol interpreter belongs. All customer-written PIs should be registered with this flag set to "DEFAULT". Setting this to any other value (or changing the values in existing REGISTER statements) may cause the Sniffer analyzer to crash.

`char *prot_name`
The name of the protocol that your PI interprets. The entry point to your PI must be named `interp_<prot_name>`; the protocol interpreter data structure used by your PI must be named `pi_data_<prot_name>`.

`char *menu_name`
A pointer to the string that will appear in Sniffer analyzer menus to refer to your PI. This string must not exceed 18 characters.

`char *pi_det`
A pointer to the string containing an abbreviation identifying your PI on lines in the detail window. To preserve alignment with displays produced by the Sniffer analyzer's other PIs, the string should be in one of the forms "xx:", "xxx:", or "xxxx:", but you may use any string of up to 5 characters.

`int pi_type`
The type of PI; see the `PITYPE_XXX` symbols in PI.H. The most common instance of `pi_type` are:

Network	PI Type	Comment
ARCNET:	PITYPE_ARCID	Indicates that this protocol interpreter is system-code-demultiplexed over Arcnet.
Token ring, Ethernet, StarLAN and PC Network:	PITYPE_SAP	Indicates that this protocol interpreter is SAP-demultiplexed over token ring, Ethernet, StarLAN, or PC Network.
Ethernet, StarLAN or PC Network:	PITYPE_ETYPE	Indicates that this protocol interpreter is Ethertype-demultiplexed over Ethernet, StarLAN, or PC Network.
Any:	PITYPE_EMBEDDED	Indicates that the protocol interpreted by this PI is embedded within other protocols.

Any:	PITYPE_FORCEE	Indicates that this PI should appear on the menu of protocols that can be forced. Note that if you use this PITYPE, your protocol must follow the standard calling sequence described in the section "Calling Conventions for Protocol Interpreters."

`int ntypes;` The number of types that can be processed by this demultiplexed interpreter. For an embedded protocol, specify 0. Ntypes depend upon the network:

ARCNET:	The number of system codes
Token ring:	The number of DSAPs
Ethernet, StarLAN and PC Network:	The number of DSAPs or Ethertypes

`int *types;` An array of "ntypes" integers representing the various system codes, DSAPS, or Ethertypes (as appropriate to the network) that can be processed by this demultiplexed interpreter.

For an embedded protocol, this parameter is ignored and should be specified as a null address.

A number of sample registration lines have been inserted into INITPI.C. You should edit one of them to register your protocol interpreter. For example, if your PI interprets a demultiplexed protocol with an Ethertype, then the registration line following the line:

```
#if USERPI_OVER_ETYPE
```

should be edited to register your protocol (instead of "sample"). For demultiplexed protocols, you must also edit (and should rename) the variables that control the demultiplexing:

`sample_sap` (currently set to 0x91)

`sample_arcid` (currently set to 0x91)

`sample_etypes` (currently set to 0x900 and 0x911)

The REGISTER macro assumes that `pi_data_<prot_name>`, `interp_<prot_name>`, and `rt_interp_<prot_name>` have been declared. Add these declarations to INITPI1.H. You can add a line in that file that declares `pi_data_sample`, etc.

The REGISTER macro assumes that `piptr_<prot_name>` has been declared. Add this declaration to INITPI2.H. You can add a line in that file that declares `piptr_sample`.

Building a New Sniffer Analyzer

To build a new Sniffer analyzer with your new PI integrated into it, follow this process:

To build a new Sniffer analyzer:

1. Write the code for your protocol interpreter or modify the existing stub routines.
2. Make the necessary modifications to INITPI.C, INITPI1.H, and INITPI2.H to register your protocol interpreter.
3. Examine the OVLINK.LNK file and modify it appropriately. Required modifications are:

   a. If you have written a new module, then remove the pound sign from the line that reads "# file sample.obj" and change it to include the object file the compiler produced from your module instead of SAMPLE.OBJ. If you have written several new modules, duplicate that line as required to include them all.

   b. If you have extended an existing protocol interpreter by editing one of the C modules provided in the PI Development Kit, then remove the pound sign from the line that includes the object file produced from that module. Also, insert a pound sign at the beginning of the line that reads module "`<module_name>.cmp`, `<module_name>.cma`."

   Warning: The OVLINK.LNK file is the command file for the linker. Do not change the overlay structure of the Sniffer analyzer code, as this may cause it not to work. Also, do not make miscellaneous changes to this file based on RTLINK documentation; the version of RTLINK distributed with the PI Development Kit is not the commercially available one, and features may differ.
4. Examine the BUILDSNF.BAT file and change the lines that set the following three variables:

tmp	Must contain a directory on a disk that has at least 2.5 Mb of free space.
lib	Must contain the directory on which version 6.00AX of the Microsoft libraries resides.
include	Must contain the directory on which version 6.00AX of the Microsoft include files resides.

5. Run the buildsnf.bat batch file with the command:

```
BUILDSNF <pi_integration> [ < module > ]
```

   The `pi_integration` parameter must be in upper case and must be one of the following:

ARCID	For a demultiplexed PI for Arcnet.
DDP	For an embedded PI over AppleTalk DDP.
ETYPE	For a demultiplexed PI with an Ethertype.
IP	For an embedded PI running over IP.
NETBIOS	For an extension to the NETBIOS PI.
SAP	For an embedded PI with a SAP.
SMB	For an extension to the SMB PI.
TCP	For an embedded PI running over TCP.
UDP	For an embedded PI running over UDP.
X25	For an extension to X.25.
XWIN	For an extension to XWINDOWS.

The module parameter is the name of a new module that you have written (without the .c extension). This module will be compiled. If you have extended existing modules but have not written one of your own, you may omit this parameter.

6. Move the xxSNIFF.EXE file built by this operation to the xxSNIFF directory on your Sniffer analyzer (where xx is a two- letter abbreviation of your network type; EN for ethernet, AR for Arcnet, etc.).

   Note: Rename your existing Sniffer analyzer executable first so that you will still have it if your new one does not work.

Example: Changing the TCP Summary Line

Figure 1–7 shows how to write a capture filter for the TCP interpreter (part of the TCP/IP PI) in order to change the formatting of the **summary line** while generating the lines for the **detail** view with the standard interpreter.

```
/* file: sample2.c */
/*******************************************************************************
SAMPLE2.C
This is an example capture filter to the TCP Protocol Interpreter that changes
the format of the summary line from that generated by the standard PI.

>>>  This is for Sniffer analyzer version 2.30, where interp_tcp() is
>>>  not called indirectly by interp_ip() using piptr_tcp.
>>>  Accordingly, we intercept at the IP level instead of the TCP level;
>>>  all occurrences of "interp_ip" are changed to "interp_ip2" in initpi.c.
*******************************************************************************/
#include "pi.h"

extern  struct pi_data  *pi_data_tcp;   /* ptr to PI data for TCP PI */
extern  int interp_ip (void *, int);    /* the original IP interpreter */

/*--- Our replacement IP interpreter. It gets control because all occurrences
   of "interp_ip" have been replaced by "interp_ip2" in initpi.c---*/

int interp_ip2 (ip_ptr, length)         /* OUR REPLACEMENT IP INTERPRETER */
char    *ip_ptr;                        /* pointer to IP header */
int     length;                         /* length of remaining data */
{
int     tcp_sumflag, bytes_used, flags;
char    *ptr, *tcp_ptr;

    tcp_sumflag = pi_data_tcp->do_sum;          /* save TCP PI's summary flag */
    pi_data_tcp->do_sum = 0;                    /* temporarily turn it off */
    bytes_used = interp_ip (ip_ptr, length);    /* call IP then perhaps TCP */
    pi_data_tcp->do_sum = tcp_sumflag;          /* put back TCP's flag */
    if (tcp_sumflag                             /* if it was on, */
     && ip_ptr [9] == 6) {                   /*  and the embedded protocol was TCP, */
      ptr = get_sum_line (pi_data_tcp);      /*  then generate our TCP summary line. */
        tcp_ptr = ip_ptr +                      /* start of TCP header */
            ((ip_ptr [0] & 0xf) << 2);          /*   based on "IHL" IP field */
        ptr = stradd (ptr, "TCP ");             /* start with protocol name */
         /* Format the summary line here. As an example, we build a line
            that contains only the keywords for the TCP flags that are set.
            If we were serious, we'd define structures for the headers and
            probably format other fields as well. */
        flags = tcp_ptr [13];                   /* the TCP flags byte. */
        if (flags & 0x01)
            ptr = stradd (ptr, "FIN ");         /* data finished flag */
        if (flags & 0x02)
            ptr = stradd (ptr, "SYN ");         /* synchronize flag */
        if (flags & 0x04)
            ptr = stradd (ptr, "RST ");         /* reset flag */
        if (flags & 0x08)
            ptr = stradd (ptr, "PSH ");         /* push flag */
        if (flags & 0x10)
            ptr = stradd (ptr, "ACK ");         /* Acknowledge flag */
        if (flags & 0x20)
            ptr = stradd (ptr, "URG ");         /* urgent flag */
        }
    return bytes_used;
    }
/* end of sample2.c */
```

Figure 1–7. Example capture filter code.

Programming and Debugging Hints

The LARGE memory model is used throughout the Sniffer analyzer and must be used by your PIs. This means that pointers are four bytes and integers are two bytes so that the symbol NULLP, not zero, should always be used to represent a null pointer.

Extreme care should be taken in writing PIs, especially in the manipulation of pointers used as targets. There is no hardware memory protection and with 4-byte pointers, no segment address limitation; every byte in the machine is vulnerable to a wayward pointer.

Be particularly careful that incorrect or even totally random protocol data will not cause your interpreter to overflow strings, to access invalid memory areas, or to loop.

The Microsoft Codeview debugger will not work because of the size of the Sniffer analyzer executable module, but SYMDEB may work.

Avoid the use of printf for debugging messages since the cursor is off the screen most of the time and the messages will not be seen. You can instead insert messages into the debugging window using the similarly called `debug_msg` function; use **Shift-F1** to pop up the debugging window.

You also can insert debugging messages into the **detail** view by using `get_int_line`. If you must use printf, specify DEBUG as a command line parameter when invoking the Sniffer analyzer. The cursor will be left somewhere on the screen, but your output will destroy the screen formatting.

Be aware of interpreting invalid information beyond the end of the stored data. If frames were captured in "partial frame" mode, the data present may be less than the entire frame; the global `bytes_not_stored` indicates how much data is not present.

The PIF routines and the sprintf function detect when they are about to access locations beyond the end of the frame data. They print a **frame too short** message into the **detail** view, and it does not return to your PI.

Be careful not to store more than `MAX_SUM_LINE` characters in a **summary** line buffer or more than `MAX_INT_LINE` characters in a **detail** line buffer, regardless of what the frame data might be. The PDK sprintf function protects against this possibility.

The 80386 processor that is used in the Sniffer analyzer stores integers in low-high format. Depending on your protocol, you may have to reverse integers for printing or calculation. The PIF routines ending in `_hl` are useful in that case, as is the "r" flag in PDK sprintf.

The Microsoft Version 6.00AX compiler supports argument type checking using ANSI standard function prototyping. The #include files provided with the PI Development Kit have declarations for all the documented functions, and you are encouraged to add declarations of your new functions.

The /J flag has been used to compile all modules to make the default for character variables unsigned.

INDEX

Network General

Index

A

address
- embedded 1-20

B

build
- custom Sniffer PI 1-31

C

calling conventions
- PI 1-7

compiler
- PI 1-34

D

data structure
- PI 1-8

demultiplexed PI 1-3

detail view 1-11, 1-13, 1-32
- synchronization with summary view 1-10
- use of PI 1-3

DLC
- header, pointer 1-8

E

embedded
- address 1-20
- interpreter 1-3

example
- new PI 1-22

F

flags
- options 1-18
- summary line 1-21

format
- PI routines 1-10
- PI utilities 1-10

frame
- number, pointer 1-8

H

higher-level
- protocol 1-3

highlight
- hex corresponding to detail interpretation 1-10

I

installation
- Microsoft C compiler 1-7
- PDK files 1-6
- PI Development Kit 1-5

L

LLC
- header, pointer 1-8

M

Microsoft C compiler 1-34
- installation 1-7

N

name
- symbolic 1-20, 1-21

P

PDK sprintf
- controlling values 1-20
- functions 1-17
- printing values 1-18
- skipping values 1-19

PI
- adding symbolic names 1-20
- build custom Sniffer protocol interpreter 1-31
- calling conventions 1-7
- compiler 1-34
- custom 1-3
- data structure 1-8
- demultiplexed vs. embedded 1-3

dependencies on other frames 1-24
example 1-22
formatting 1-10
formatting utilities 1-10, 1-12
integrating into existing code 1-27
memory model 1-34
output 1-9
porting current 1-4
registering 1-28
return value 1-8
PIF
offset 1-11
routines 1-10, 1-12
protocol
higher-level 1-3

R

registration
PI 1-28
return value
PI 1-8

S

Sniffer analyzer
build new, with custom PI 1-31
sprintf
controlling values 1-20
functions 1-17
printing values 1-18
skipping values 1-19
summary view 1-9, 1-11, 1-32, 1-34
display flags 1-21
flags 1-21
synchronization with detail view 1-10
use of PI 1-3
symbolic equivalent
display by custom PI 1-21

T

technical support vii

NETWORK GENERAL CORPORATION
Marketing Publications Comment Form

At Network General, we are interested in your suggestions to improve this manual. Please take a moment to complete the following survey. Your comments are greatly appreciated.

MANUAL TITLE ______________________________

1 **How do you use this manual?** (check one or more.)

- ☐ To get an overview of the product
- ☐ To learn a task
- ☐ To look up a fact
- ☐ To get out of trouble
- ☐ Other ______________

2 **How often do you use this**

	DAILY	WEEKLY	MONTHLY	LESS THAN MONTHLY
◆ manual? (check one.)	☐	☐	☐	☐
◆ product? (check one.)	☐	☐	☐	☐

3 **Evaluate the manual?**

		YES	NO	COMMENTS
Is information...	◆ accurate?	☐	☐	______
	◆ easy to read?	☐	☐	______
	◆ easy to find?	☐	☐	______
Are examples...	◆ helpful?	☐	☐	______
	◆ realistic?	☐	☐	______
	◆ plentiful?	☐	☐	______
Are illustrations...	◆ helpful?	☐	☐	______
	◆ realistic?	☐	☐	______
	◆ plentiful?	☐	☐	______
Is the index...	◆ complete?	☐	☐	______
	◆ accurate?	☐	☐	______

4 **Did you notice any omissions?** ______________________________

5 **Do you have any general comments or suggestions?** ______________________________

"We solve network problems."™

fold here

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 9 MENLO PARK, CA

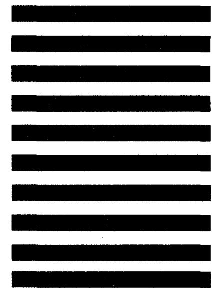
POSTAGE WILL BE PAID BY ADDRESSEE

Network General Corporation

4200 Bohannon Drive

Menlo Park, CA 94025-9791

ATTN: Technical Publications Department



NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

fold here

The following information will help us better evaluate your needs.

Position ____________________

Department ____________________

Optional:

Name ____________________

Company ____________________

Address ____________________

Phone ____________________

Errata

Sniffer Network Analyzer: Protocol Interpreter Development Kit

October 30, 1992

The following errata apply to the *Sniffer Network Analyzer: Protocol Interpreter Development Kit.*

1. On page 1–4, under the heading, "Writing a Protocol Interpreter," the manual informs you that you will need to obtain Version 6.00AX of the Microsoft C compiler. This compiler and the necessary library files are now included as part of the Protocol Interpreter Development Kit. You no longer need to obtain them separately.

2. On page 1–6, Figure 1–1 lists the files provided with the Protocol Interpreter Development Kit. In addition to the files listed, the following files are also provided as part of the Microsoft 6.00AX C Compiler:

Files Provided With Microsoft 6.00 AX C Compiler		
CL.EXE	CL.MSG	C1L.EXE
LLIBCE.LIB	C1.ERR	C23.ERR
CL.ERR	C2L.EXE	C3L.EXE

3. On page 1–7, under the heading, "Installing the Microsoft C Compiler," the document gives instructions for installing the compiler. The Protocol Interpreter Development Kit now includes all the tools required to write new protocol interpreters and build new Sniffer analyzers, including the compiler and the library. You will not need to install the Microsoft C Compiler because it is already installed with the Protocol Interpreter Development Kit.

4. On page 1–31, Step 4 of the procedure, "To build a new Sniffer analyzer," lists several changes that should be made to the BUILDSNF.BAT file. The necessary files for Microsoft C 6.00AX are now distributed with the Protocol Interpreter Development Kit. If you are using the compiler supplied with your Protocol Interpreter Development Kit, then the BUILDSNF.BAT file is already set correctly. You will not need to change the BUILDSNF.BAT file unless you are using a custom development environment. If this is the case, you will need to change the lines in BUILDSNF.BAT that set the three variables listed in Step 4 on page 1–31.

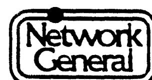
5. To use the Protocol Interpreter Development Kit, you must add the following line to the CONFIG.SYS file of the Sniffer Network Analyzer:

   ```
   DEVICE=C:\DOS\HIMEM.EXE
   ```

   Note: If your Sniffer Network Analyzer has less than 32MB of RAM, you must remove the above command line before starting the Sniffer Network Analyzer.

6. Before using the Microsoft C compiler, type the following command:

   ```
   SET PATH=C:\DOS;C:\TOOLS;C:\TOOLS\C600AX
   ```

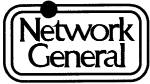


We simplify network complexity.™

Network General Corporation
4200 Bohannon Drive
Menlo Park, CA 94025
TEL: (415) 688-2700
FAX: (415) 321-0855

Network General Europe
Belgicastraat 4
1930 Zaventem, Belgium
TEL: (32-2) 725-6030
FAX: (32-2) 725-6639

Network General Canada, Ltd.
2275 Lakeshore Blvd., West, 5th Floor
Etobicoke, Ontario M8V 3Y3 Canada
TEL: +1 (416) 259-5022
FAX: +1 (416) 259-3727



We solve network problems.™

Network General Corporation
4200 Bohannon Drive
Menlo Park, California 94025
(415) 688-2700

Network General Europe
Belgicastraat 4
1930 Zaventem (Brussels), Belgium
32-2-725-6030

P/N:PA-7000