Non Intrusive Realtime Debugger User Guide

Philips Media

PHILIPS

Nird

CD-i Non-Intrusive Realtime Debugger

User Guide

version 1.0

Philips Media Systems B.V.

CONTENTS

LEGAL INFORMATION **L-1**
Copyright L-1
Disclaimer L-1
Reproduction notice L-2
Trademarks L-2

BEFORE YOU BEGIN **B-1**
Manual objectives B-1
Intended audience B-1
Structure of the manual B-2
Manual conventions B-3
Syntax conventions B-4
Icons B-5
Recommended literature B-6

Chapter 1. INTRODUCTION **1-1**
1.1 Overview of the system 1-2
1.1.1 Bugs that make a player reset 1-3
1.1.2 Bugs that make a player freeze 1-4
1.1.3 Obscure bugs 1-4
1.1.4 Design for testability 1-5
1.1.5 Summary 1-6
1.2 Hardware Requirements 1-7
1.3 Delivered system 1-9

Chapter 2. INSTALLATION 2-1

2.1 Installing the software 2-1
2.1.1 Install the host software 2-1
2.1.2 Install the device driver 2-2
2.1.3 Update the search path 2-3

2.2 Installing the hardware 2-4
2.2.1 Inserting the debugging cartridge 2-4
2.2.2 Connecting the PC to the cartridge 2-5

2.3 Testing your installation 2-6

Chapter 3. GETTING STARTED 3-1

3.1 What is a blue-screen crash? 3-2

3.2 Introduction to the NIRD utility disc 3-3

3.3 Finding what caused the crash 3-4
3.3.1 Specify a trigger cycle 3-5
3.3.2 Waiting for a trigger cycle 3-7
3.3.3 Uploading the data 3-9
3.3.4 Disassemble the data 3-11

3.4 Data analysis 3-13
3.4.1 Basic analysis 3-14
3.4.2 Determining player modules 3-17
3.4.3 Determining application modules 3-18
3.4.4 Using symbol tables 3-19
3.4.5 Viewing C-source 3-22

Chapter 4. NON-INTRUSIVE DEBUGGING 4-1

4.1 Host software overview 4-1

4.2 The inquest facility 4-5
4.2.1 Setting a trigger 4-7
4.2.2 Uploading inquest data 4-12
4.2.3 Disassembling inquest data 4-14
4.2.4 Viewing inquest data 4-16

4.3 Modules, symbols and C-source 4-27
4.3.1 Detecting player modules 4-27
4.3.2 Detecting application modules 4-33
4.3.3 Loading symbol tables 4-38
4.3.4 Viewing C-source 4-42

4.4 Minimally intrusive printfs 4-46

4.5 Tracing system calls 4-57

4.6 Hints and tips 4-65
4.6.1 Avoiding null pointers 4-65
4.6.2 Fixing cyclic lists 4-66
4.6.3 Avoiding software rot 4-67

Chapter 5. SOFTWARE GUIDE 5-1

5.1 The 'System' menu 5-1
5.1.1 The 'About' menu item 5-1
5.1.2 The 'Print' menu item 5-2
5.1.3 The 'Statistics' menu item 5-3
5.1.4 The 'DOS shell' menu item 5-3
5.1.5 The 'Quit' menu item 5-3

5.2 The 'Find' menu 5-4
5.2.1 The 'Goto cycle' menu item 5-4
5.2.2 The 'Find cycle' menu item 5-4
5.2.3 The 'Find next' menu item 5-5

5.3 The 'Module' menu 5-6
5.3.1 The 'Detect mdir' menu item.....5-6
5.3.2 The 'Load PMD file' menu item 5-7
5.3.3 The 'Watch loads' menu item 5-7
5.3.4 The 'Load AMD file' menu item 5-8
5.3.5 The 'Load symbols' menu item.....5-9
5.3.6 The 'Load dbg data' menu item 5-10

5.4 The 'Inquest' menu 5-11
5.4.1 The 'Trigger' menu item.....5-11
5.4.2 The 'Upload' menu item.....5-12
5.4.3 The 'Disassemble' menu item 5-12
5.4.4 The 'Load INQ file' menu item 5-12

5.5 The 'Printf' menu 5-13
5.5.1 The 'Start tracing' menu item 5-13
5.5.2 The 'View log file' menu item 5-13

5.6 The 'Call' menu.....5-14
5.6.1 The 'Trace syscalls' menu item 5-14
5.6.2 The 'View log file' menu item 5-15

5.7 The 'Options' menu 5-16
5.7.1 The 'Inquest' menu item.....5-16
5.7.2 The 'Misc' menu item 5-16

5.8 The 'Window' menu 5-17
5.8.1 The 'Size' menu item.....5-17
5.8.2 The 'Move' menu item 5-17
5.8.3 The 'Zoom' menu item 5-17
5.8.4 The 'Next' menu item.....5-17
5.8.5 The 'Close' menu item 5-17

5.9 The 'Help' menu.....5-18

Chapter 6. FAULTFINDING 6-1

Serial communication problems 6-1
Parallel communication problems 6-2
Digital Video problems 6-2
Disassembly problems 6-3
Reporting bugs and getting technical support .. 6-3

Chapter 7. APPENDICES 7-1

A Cartridge connectors 7-1
B Upgrading firmware 7-3
C The DAT file structure 7-4
D The INQ file structure 7-5
E The PMD file structure 7-7
F The AMD file structure 7-9
G Configuration file format 7-11
H Other file structures 7-12
I Instruction set summary 7-13
J System call summary 7-22
K Source for utility disc 7-30

Chapter 8. ALPHABETICAL INDEX 8-1

Tables

I Pin allocations for serial ports on cartridges 7-1
II Pin allocations for parallel port on cartridge 7-2
III Event frame format 7-4
IV Inquest file format 7-5
V Inquest Cycle Frame format 7-6
VI Player Module Directory file format 7-7
VII Application Module Directory file format 7-9
VIII Status register 7-13

Figures

1 Nird and DV cartridges.....1-7
2 Trigger selection dialog box 3-5
3 Upload control dialog box 3-10
4 Disassembly progress dialog box 3-12
5 Inquest data from NIRD utility disc 3-14
6 Analysis for the NIRD Utility Disc crash 3-20
7 The basic workspace screen 4-2
8 Trigger selection dialog box 4-7
9 Upload control dialog box 4-12
10 Viewing inquest data.....4-18
11 Defining the search specification.....4-21
12 Example inquest data 4-23
13 Example processor-internal manifest cycles ...4-25
14 Detect player module directory dialog box 4-28
15 Inquest and module display windows 4-30
16 Detect application modules dialog box.....4-34
17 Scrolling module display in dialog box 4-35
18 Modifying an application module's name 4-37
19 Host software workspace (symbol table) 4-39
20 Program symbols (inquest data window) 4-41
21 Workspace having loaded debug data file.....4-43
22 Inquest data indexing into source file.....4-44
23 Inquest data: write port being created 4-48
24 Displaying minimally intrusive printfs 4-55
25 What happens on a system call 4-58
26 What happens at the end of a system call.....4-59
27 Choosing which system calls to trace 4-61
28 Viewing a log file of traced system calls.....4-63

LEGAL INFORMATION

Copyright

This manual reflects Version 1.0 of the
CD-i Non-Intrusive Realtime Debugger

Publication Editors:	J. Jacobs, W. Reijnaerts, P.Clarke
Contributing Writers:	P. Clarke
Other Contributors:	R. van de Laarschot
Document Revision:	1.0
Publication Date:	March 31st 1995
Product Number:	22ER9480/00

Disclaimer

The information contained herein is believed to be accurate as of the date of publication; however, Philips Electronics N.V. will not be liable for any damages, including indirect or consequential, from use of the CD-i Non-Intrusive Realtime Debugger or reliance on the accuracy of this documentation. The information herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. Distribution of the software or documentation, whole or in part, to any other system or to any other party may constitute a misappropriation of trade secrets and confidential processes which are the property of Philips Electronics N.V., Philips Media Systems B.V., and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

Trademarks

CD-RTOS and CD-i are trademarks of Philips Electronics N.V.

MS-DOS and Windows are trademarks of Microsoft Corporation.

OS/2 is a trademark of International Business Machines Corporation.

Names of products mentioned herein are used for indication purposes only and may be trademarks and/or registered trademarks of their respective companies.

BEFORE YOU BEGIN

Manual objectives

This manual explains how to use the CD-i Non-Intrusive Realtime Debugger. It describes installation of the hardware and software components of the system, and how to use the various debugging facilities that the system provides, namely:

- The inquest facility, where you are shown disassembled bus activity up to a specific cycle, complete with module and label names corresponding to each bus cycle, and can be shown on-screen the associated C-Source.
- Module directory support functions, where you can determine CD-RTOS module positions within a CD-i player, and associate debugging files produced by a compiler with CD-RTOS modules for label and C-source tracing.
- The minimally-intrusive **printf** facility, where you can display debugging messages with very low overhead without using the serial port of a CD-i player.
- System call tracing, where you can monitor system calls and their parameters in real-time.

Intended audience

This manual is intended primarily for experienced programmers who are familiar with software development for the CD-i platform, and who have experience of the CD-RTOS operating system.

Structure of the manual

This manual is divided into eight chapters as follows:

Chapter 1 **Introduction to the Non-Intrusive Realtime Debugger**
An overview of the system is presented and a description is given of its debugging facilities. This chapter also describes what is included in your NIRD debugger pack.

Chapter 2 **Installation**
The installation of the hardware and software components is described, and a test is given to verify successful installation.

Chapter 3 **Getting started**
This chapter provides a quick way to get acquainted with the CD-i Non-Intrusive Realtime Debugger using the NIRD Utility WORM as an example debugging task. Basic operating instructions are described, and examples are given that can lead to quick results.

Chapter 4 **Non-intrusive debugging**
This chapter provides an in-depth step-by-step guide to all the debugging features provided by the system.

Chapter 5 **Software guide**
This chapter describes each option of the menus within the debugging software.

Chapter 6 **Fault-finding**
This chapter provides information on what to do if things do not do what they are supposed to.

Chapter 7 **Appendices A ... K**

Chapter 8 **Alphabetical index**

Manual conventions

The following conventions are used throughout this manual:

- The term "disc" always refers to a compact disc.
  The term "disk" always refers to a hard disk.
  The term "diskette" always refers to a 3.5 inch floppy diskette.

- The names of programs, directories, command lines, and files referred to in the text and the scripts are shown in a **bold** typeface:

  C:\NIRD\BIN
  System

- The names of keys on the keyboard are *italic* when referred to in the text:

  the *Control* key
  the *Return* key

Syntax Conventions

The command line syntax uses the following conventions:

- Command line examples are shown in a **bold** typeface; for example:

  #define NOTHING

- When two keys are to be pressed simultaneously, they are shown connected by a hyphen:

  Control-C

- Variables are in *italic*:

  filename

- Constants, such as keywords, separators and punctuation are shown in a **bold** typeface:

  nird

- Syntax elements within brackets ([]) are optional:

  [option]

- Repeated syntax elements are indicated by an ellipsis (...) following the element to be repeated:

  option ...

- A syntax bar (|) is the equivalent of a boolean OR; one, but not both, of the syntax elements on each side of the bar may be used. The bar is used in conjunction with italic braces (*{}*) which indicate the limits of a syntax group. For example:

  *{***at** | **by***}* ***rsn***
  {element1 | *{***keyword** *element2}* | *element3}*

Icons

This manual uses the following icons to alert you to special information:

Reference: This icon indicates a cross-reference to related information.

Note: Notes contain additional information that is important to understand, but is not procedurally dangerous.

Remark: This icon alerts you to information about unusual behavior of a utility, command, or procedure.

Warning: Warnings contain information that alerts you to possible losses of data.

Recommended literature

To help you fully understand the technical and philosophical aspects of full motion video encoding, the following technical specifications are recommended:

Title ISO Standard 11172:
Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s
(MPEG Standard - current version)
Author ISO/IEC
Publisher ISO/IEC Copyright Office
City Genève
Year August 1993
ISBN –

Article *MPEG: A video Compression Standard for Multimedia Applications*
Author Didier Le Gall
Magazine *Communications of the ACM*
Year April 1991, Volume 34, No. 4

For general CD-i design and production information, refer to:

Title *Discovering CD-i*
Author Eric Miller and Walden Miller
Publisher Microware Systems Corporation
City Des Moines
Year 1991
ISBN –

Title	*CD-i Designer's Guide*
Author	Signe Hoffos, Graham Sharpless, Phillip Smith, Nicholas Lewis
Publisher	McGraw-Hill Book Company
City	Berkshire
Year	1992
ISBN	0-07-707-580-3

Title	*The CD-i Production Handbook*
Author	Philips Interactive Media Systems
Publisher	Addison Wesley Publishing
City	Suffolk
Year	1992
ISBN	0-201-62750-7

Title	*The CD-i Design Handbook*
Author	Philips Interactive Media Systems
Publisher	Addison Wesley Publishing
City	Suffolk
Year	1992
ISBN	0-201-62749-3

Title	*The CD-i Programmer's Handbook*
Author	Philips Interactive Media Systems
Publisher	Addison Wesley Publishing
City	Suffolk
Year	1992
ISBN	–

Chapter 1. INTRODUCTION

Debugging is an inevitable task performed during software development. If a CD-i title is to be produced on time and within budget then it is important that bugs are quickly identified and corrected. The Non-Intrusive Realtime Debugger (hereafter referred to as the NIRD) is an advanced debugging system that has been developed by Philips Media Systems B.V.

An advanced debugging cartridge that plugs into the Digital Video slot of a Philips consumer CD-i player monitors the expansion bus for specific activity. Bugs can be detected in realtime, the loading of CD-RTOS modules can be watched, system calls can be monitored, and developers can use the cartridge to communicate debugging messages with very low overhead to a remote PC. This introduction to the NIRD will include a description of several types of bug, and will provide an overview of the NIRD's debugging facilities.

1.1 Overview of the system

The NIRD consists of a custom cartridge that plugs into the Digital Video slot of a consumer CD-i player, and software that runs under DOS on a debugging host PC (286 processor or above with at least 2MB RAM). A serial link, and if possible a bidirectional parallel link, provide the communication path between the cartridge and the PC. A carrier at the rear of the NIRD cartridge allows through connection to a Digital Video cartridge.

The cartridge has a 9000-gate programmable filter device with 128 KBytes of local buffer memory. This is used to detect particular bus activity and buffer the resulting data. At the heart of the cartridge is a 16MHz Motorola 68340 microprocessor with 1 MByte of memory. Here a second stage of filtering is performed before results are communicated to the debugging host PC for analysis by the programmer.

Different filter designs are stored on the cartridge and are used by the programmable filter to perform different debugging tasks. However, the debugging system software ensures that the programmer only has to concentrate on the particular debugging task and does not need to be aware of the workings of the cartridge.

The host PC software provides a standard-looking Graphical User Interface. Simple menu-driven interaction allows the user to analyse player execution and find bugs quickly and easily. Symbol table and debug data files produced by the title building process can then be used to pin-point the line of source code causing the bug.

1.1.1 Bugs that make a player reset

When a CD-i player resets there is a blue-screen flash and the player returns to the player shell. This occurs whenever an error exception occurs. The most common error exceptions are bus errors (error code 102, caused by accesses to non-valid memory space), address exceptions (error code 103, caused by 16-bit or 32-bit accesses to odd addresses), and illegal instructions (error code 104, where the processor tries to execute invalid instructions).

A bus error will occur if the processor tries to access an invalid memory address. This typically occurs if a program follows a corrupt pointer (such as during traversal of a corrupted linked-list data structure), or if the stack frame of a function becomes corrupted by misuse of local variables which over-write the return program counter.

An address error will occur if a word or long access to an odd address is attempted. This may happen if a pointer is corrupted and points to an odd address.

Illegal instructions are usually caused by the processor executing from non-program memory. This is another bug that may be caused by the function return program counter storage being corrupted. Code that accidentally modifies itself may also produce this bug.

These bugs are easily identified by the NIRD using a debugging function known as the inquest facility. Here, information about each bus cycle is stored in a 16384-cycle circular buffer until an error exception is detected. At this point data saving automatically stops, and the required amount of data up to the error may be uploaded to the PC for disassembly. It is usually very clear what has caused the bug and the host software can show on screen the line of source code that caused the error exception.

1.1.2 Bugs that make a player freeze

Certain types of bug can make a CD-i player freeze. These bugs are traditionally among the hardest to fix as it is usually not clear what the player is actually doing to cause the freeze, and to make things worse the freeze may be hard to reproduce.

There are two main causes of a freeze. The first is that the processor of the CD-i player has halted, and the second is that the processor is stuck in a never ending loop of code. Both of these bugs can be readily detected by the NIRD. The CD-i processor halts if an address error occurs during exception processing of a bus error or address error. This can be investigated using the inquest facility looking for an error exception as described above. Never ending loops of code typically occur when a linked list data structure is corrupted such that the list becomes circular. List traversal code then continuously loops looking for a termination condition that is never met. When the cartridge is looking for an error exception in the inquest facility it is possible to trigger the cartridge by pressing a key on the PC keyboard. This manually stops data being saved in the cartridge's buffer. The data in the buffer can then be uploaded and analysed.

1.1.3 Obscure bugs

Obscure bugs are usually those that are hard to reproduce and may vary from player to player. There are a number of causes that the NIRD can investigate. Null pointers are frequently the culprit, and should always be suspected first as they occur worryingly frequently. These read or write to low memory (in the area of the exception vectors), and show that something in the title is either corrupted or has not been initialised correctly. Fortunately, the fact that they access the exception vectors means that the inquest facility can detect them and show what caused them.

Another cause of obscure title bugs may be timing of delivery of data from real-time plays, or memory management problems. The NIRD allows a user to trace CD-RTOS system calls and returns (with parameters) in real-time so that issues such as these can be investigated.

1.1.4 Design for testability

Programmers frequently use debugging messages sent through the serial port of the CD-i player to aid debugging. However, there are a number of problems with this. Firstly, these messages upset the operating speed of the title due to the system call overhead of the serial communication driver code and the synchronous nature of the communications link. Secondly, due to the synchronous nature of the communication these messages must be removed from the title before release as they would cause the title to crash on a consumer CD-i player if no terminal is attached. However, even the removal of these messages can lead to problems as the programmers have typically developed and tested the title with them included. If the messages are put back in to fix the problem then the problem can disappear, and so on.

The NIRD is supplied with a pair of debugging message libraries. Linking to the first library produces serial debugging messages as before, whilst linking to the second library changes these debugging messages to 'minimally intrusive printfs'. Here the messages are written to a global variable in a shared data module which is monitored by the debugging cartridge. Any writes to this variable are observed and the data is sent to the host PC for displaying on its screen. However, since the overhead of writing messages to a memory address is very small (in the order of microseconds) the disturbance to the title is minimal.

Furthermore, since the data is just written to a global variable in a tiny (80 bytes long) shared data module, the messages can be left in the finished title. This also means that if problems are ever found with the title in the future, on a new CD-i system for example, then the debugging messages will still be available if a debugging cartridge is connected.

Another aspect of design for testability is the ability to trigger the inquest facility on a specific bus cycle. This may be a read or write to a given address, or a given DMA cycle, etc. This allows the user to look at up to 16384 cycles up to a given point of player execution. This is useful for code execution analysis and profiling.

1.1.5 Summary

The Non-Intrusive Realtime Debugger for CD-i is a new approach to debugging. Powerful debugging hardware, concealed beneath a simple to use user interface, allows title developers, system software developers, and test engineers to identify all sorts of bugs, evaluate player performance, and profile sections of code in a manner never before possible. Above all, the NIRD provides a useful tool to help software products to be released on time and in a bug free condition.

1.2 Hardware requirements

The debugging cartridge has been designed for use with all consumer CD-i players capable of accepting a standard (100 pin) Digital Video cartridge. It plugs into the expansion slot, protruding far enough out of the back of the player to provide access to connectors for host PC communication, and has a support at the rear to allow through connection to a standard Digital Video cartridge. The arrangement for players with 100-pin connectors is shown in Figure 1.

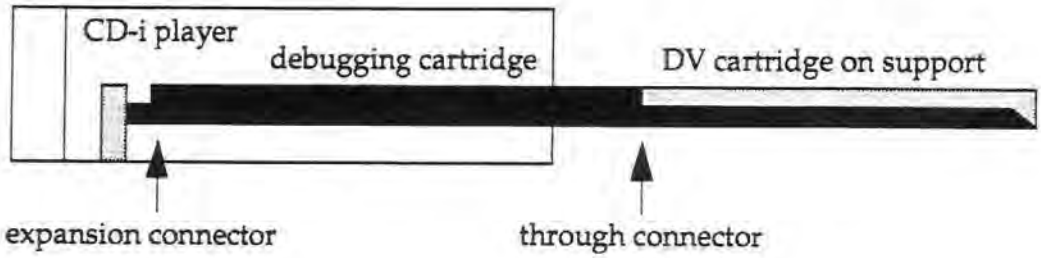

Figure 1: Side view of player with NIRD and DV cartridges installed

Newer players (including models 450, 550, 210/40, 220/60 and later) use a smaller DV cartridge which has a 120-pin connector. An adapter is available which allows connection of a debugging cartridge and a new-style Digital Video cartridge to these players.

The debugging cartridge was not designed for use with a 60X players (601,602,604 and 605(T) models). However, if you remove the back-plate of the player then the cartridge can be inserted provided the 60X has a Digital Video expansion connector. Please note that use of the debugging cartridge in a manner not intended is done at the user's own risk. This includes use of the cartridge PCB outside of its box, or use with CD-i player's that are not manufactured under the Philips or Magnavox brand names.

It is recommended that the debugging host PC is equipped with a 80286 processor or better (486 or better preferred), with at least 2 MBytes of RAM. The COM1 serial port is required for serial communication with the cartridge, and the provided cable assumes that it is available in the form of a 9-pin D-connector. Suitable adapters are available from peripheral vendors if COM1 is available only as a 25-way D-connector. If a serial mouse is used in COM2 then the mouse driver must be instructed to use the COM2 port. The PC software has been developed to run under MS-DOS Version 5 or above, or PC-DOS Version 6 or above.

The debugging PC communicates with the debugging cartridge through a serial link. However, to speed up data transfer for some debugging functions a bidirectional parallel link can also be used. Most computers support bidirectional parallel communications, and replacement I/O cards with a bidirectional parallel interface cost under $20. The host software will inform you if it finds that the computer only has a unidirectional parallel interface.

The usual environment for using the host debugging software is DOS, although the application can be run in a DOS-box under Windows or OS/2. Beware that the COM port initialisation that these windowing systems perform on starting or exiting may interfere with the debugging system communications device driver. If the serial comms link with the debugging cartridge appears to have failed when trying to run the software after leaving a windowing package then reboot your system and try again.

1.3 Delivered system

The NIRD package consists of the following components:

- This User Guide
- A debugging cartridge for a consumer CD-i player
- A CD-i WORM marked "Non-Intrusive Realtime Debugger Utility Disc"
- A 9-pin female to 9-pin female serial cable
- A 25-pin male to 25-pin female parallel cable
- A 3 1/2" high-density software distribution disc
- An interface to allow connection of the debugging cartridge to a CD-i player with a 450-style 120-pin Digital Video connector.

The distribution disc contains the compressed host software. The installation process allows you to specify the name of the directory under which the software is installed. Assuming the default choice of C:\NIRD the installed files will be in the following directories:

C:\NIRD\BIN:	**COMMS.SYS**	Device driver to manage serial comms
	NIRD.EXE	Main host software
	DPMI16BI.OVL	DOS protected mode overlay file
	RTM.EXE	DOS protected mode run-time manager
	NIRD.CFG	Host software configuration text file
	SYSMENU.HLP	Help file for system menu
	FINDMENU.HLP	Help file for find menu
	MODMENU.HLP	Help file for module menu
	INQMENU.HLP	Help file for inquest menu
	TRCMENU.HLP	Help file for printf menu
	RTOSMENU.HLP	Help file for call menu
	OPTMENU.HLP	Help file for options menu
	WINMENU.HLP	Help file for window menu
	SUPPORT.HLP	Help file for obtaining support
C:\NIRD\LIB:	**TRACE.H**	Include file for minimally intrusive printfs
	TRACE.R	Link library for minimally intrusive printfs
	TRACE_P.R	Link library for mapping to normal printfs
C:\NIRD\EXAMPLE:	**PAULS602.PMD**	Module directory of test CD-i player
	MDIR.AMD	Application module data for example
	CDI_MDIR.STB	Symbol table for module on utility WORM
	CRASH.DAT	Inquest data up to crash
	CRASH.INQ	Disassembled inquest data
	CRASH.TXT	Text printout of inquest data

Chapter 2. INSTALLATION

This section of the user guide shows you how to install the software and hardware components of the NIRD. It is assumed that you are familiar with PC's and DOS. Installation of the software is described step by step and connection to the hardware debugging cartridge is explained. Once the system has been set up, verify the installation using the test procedure given below.

2.1 Installing the software

There are three stages to installing the software:

1. Install the host software on your computer's hard drive.
2. Install the device driver.
3. Update the search path.

Each of these will now be described fully in the following subsections.

2.1.1 Install the host software

The distribution disc contains an installation program and a compressed data file. The installation program checks various parameters of your computer against minimum requirements and then allows you to specify where you would like the NIRD files to be placed on your hard drive. The default location is in a subdirectory tree from the **NIRD** directory on the **C:** drive. References to paths in this documents assume that this is where the files are placed.

To install the software insert the distribution disc, change to that drive, and type **install**. This starts the NIRD System Installation utility. You will first be prompted to choose the drive letter to install to (default is **C**) and then the directory name (default is **NIRD**). The installation utility will then uncompress the NIRD files into the required place on your hard drive.

2.1.2 Install the device driver

The host software needs a device driver to manage serial communications with the debugging cartridge. This file is called **COMMS.SYS** and can be found in the **BIN** subdirectory of your installed NIRD files (**C:\NIRD\BIN** if default file locations were chosen). To install this device driver every time the computer is switched on an entry needs to be made in the **CONFIG.SYS** file found in the C: root directory.

The device driver can be loaded into conventional memory or it can be loaded high. With the editor of your choice (such as MS-DOS "edit"), add the following line after other device driver references:

DEVICE=C:\NIRD\BIN\COMMS.SYS

to load it into conventional memory, or the line:

DEVICEHIGH=C:\NIRD\BIN\COMMS.SYS

to load it into the high memory area. If you do not understand the difference between these two memory areas then please consult your MS-DOS manual.

If the NIRD files were installed into a directory location other than **C:\NIRD** then you will need to make the appropriate changes to the above line.

☞ You must ensure that the mouse driver does not conflict with the NIRD communications device driver. The mouse must be connected to COM2 and the driver must be instructed to use the COM2 port. If the port is not specified using the appropriate command line switch (usually "**/2**") then it may scan the ports, hooking the COM1 interrupt in the process, and preventing the NIRD communications driver from functioning.

2.1.3 Update the search path

It is usual to debug each application in its own directory area. For example, you may wish to debug your "Wonder" title in the directory **C:\NIRD\PROJECTS\WONDER**. In order for DOS to know where to find the host software you need to update the search path of your computer to include the **C:\NIRD\BIN** subdirectory.

With the editor of your choice edit the **PATH** line in the **AUTOEXEC.BAT** file found in the **C:** root directory to include the NIRD binary subdirectory. It should now look something like the following:

PATH=C:\DOS;C:\NIRD\BIN;...etc...

If the NIRD files were installed into a directory location other than **C:\NIRD** then you will need to make the appropriate changes to the above line.

2.2 Installing the hardware

There are two stages to installing the hardware:

1. Inserting the debugging cartridge.
2. Connecting the PC to the cartridge.

2.2.1 Inserting the debugging cartridge

For players which use the older Digital Video cartridges with 100-pin connectors proceed as follows. With the CD-i player turned off remove the Digital Video cartridge if one is fitted and insert the debugging cartridge. The final mating of the two connectors can be felt to confirm that the cartridge is fully inserted. The cartridge extends beyond the back of the player to allow access to connectors. For use with a Digital Video cartridge slide the Digital Video cartridge onto the support bracket at the rear of the debugging cartridge until the it fully mates with the through connector at the rear of the debugging cartridge.

For newer CD-i players which use the new small Digital Video cartridge with the 120 pin connector proceed as follows. With the CD-i player turned off remove the Digital Video cartridge if one is fitted and insert the interface supplied with the NIRD system. This has a 120-pin connector on the top so that a Digital Video cartridge can be connected, and an old-style 100-pin Digital Video connector for connection to the debugging cartridge. The user must provide a support to raise the debugging cartridge to the required height. For use with a Digital Video cartridge use the new-style Digital Video cartridge connected to the top of the interface, not an old style Digital Video cartridge in the debugging cartridge's rear caddy.

Do not connect an old-style Digital Video cartridge to a CD-i player which has the new-style 120-pin Digital Video connector.

2.2.2 Connecting the PC to the cartridge

The host PC communicates with the debugging cartridge using a serial link. A bidirectional parallel link can also be used for faster uploading of data. However, not all PC's have a bidirectional parallel port and so this form of communication is optional. The host software will inform you if it detects that the parallel port is not bidirectional if you attempt to use it for an upload. Due to the way in which parallel communication speeds up the debugging process it is recommended that user's purchase an I/O card from a peripheral supplier if their existing parallel port is not bidirectional. Bidirectional I/O cards cost under $20 from peripheral vendors.

To provide the mandatory serial communications link use the supplied 9-pin to 9-pin serial cable to connect the "PC (serial)" connector on the cartridge with the COM1 port of the host PC. If the COM1 serial port on the PC is in the form of a 25-way connector then an adapter must be used which is available from peripheral vendors.

To provide the optional parallel communications link use the supplied 25-pin to 25-pin parallel cable to connect the "PC (parallel)" connector on the cartridge with a bidirectional parallel port of the host PC.

The "Terminal" connector on the debugging cartridge is used only for development of the debugging system. If you wish to be nosey then you may connect a terminal to this port (9600 baud, no parity). This is not required for use of the debugging system.

2.3 Testing your installation

Re-boot your PC and verify that the communications device driver displays an installation message during boot-up. To start the host software type nird at the DOS prompt. Make sure that the CD-i player which is connected to the debugging cartridge has power applied. On starting the host software the screen will change to show the host utility workspace area with a menu at the top. The simplest way to verify that the installation has been successful is to select the **System | Statistics** menu item. Use of this software is described fully later in this document but for now just do one of the following:

Mouse Click on the **System** menu option. This will display the system sub-menu. Click on the **Statistics** menu-item.

Keyboard Type **Alt+S** to display the **System** menu. Type **S** for the **Statistics** menu-item or use the down cursor arrow key to highlight the **Statistics** menu-item, and then press **RETURN**.

This displays the dynamic memory usage of the PC, and also the cartridge operating system software revision. This information is obtained from the powered cartridge over the communications link. This should display a value such as "**1.0**". If "**n/a**" is displayed ("not available") then there is a problem. Check if:

- The cartridge is fully inserted in the CD-i player.
- The player is switched on.
- The "PC(serial)" port of the cartridge is connected to the COM1 port of the PC.
- The communications device driver is installed on the PC.
- The mouse driver is not trying to access the COM1 port of the PC.

To quit the host software enter **Alt+X** which is the hot-key method, or select the **System | Quit** menu item from the menu bar.

Chapter 3. GETTING STARTED

This section shows you how to get started with using the NIRD system.

In this chapter the NIRD utility disc is used as a debugging example (you will find sample files in the EXAMPLE subdirectory of the installed software). However, if you have a bug in your own title that causes a blue-screen crash as the NIRD utility disc does then this sequence of debugging steps can be used to try to find your own bug.

After using the NIRD system to fix the example bug as described in this chapter, it is recommended that you thoroughly read the following chapter which covers non-intrusive debugging in greater detail.

3.1 What is a blue-screen crash?

Most CD-i programmers will have seen a blue-screen crash. The application is proceeding smoothly when suddenly the screen flashes to blue for half a second or so and then the player returns to the player shell.

A blue-screen crash is caused by the player's error exception code executing because the processor has detected a fatal error. To effectively debug a crash you need to know the common errors that occur and what usually leads to them.

Bus errors This is the error with CD-RTOS error code 102, and is caused by the processor trying to access invalid memory space. Most CD-i players have a 68070 processor which has a 24-bit address space, but the address registers are 32-bits wide, thus bus errors frequently happen when an address register is loaded with a corrupt value and then dereferenced. Another common cause is the processor branching to non-existing memory because the return program counter is corrupted on a function's stack.

Address errors This is the error with CD-RTOS error code 103, and is caused by a word or long word access to an odd address (bit #0 of the address set). This is usually caused by a corrupted address register or return program counter on the stack just as in the case of a bus error.

Illegal instructions This is the error with CD-RTOS error code 104, and occurs if the processor tries to execute an invalid instruction, and is usually caused by either a bug causing program space to be overwritten with corrupting data, or a corrupted return program counter on the stack forcing a branch into valid memory space which contains garbage. Be aware that some "green" instructions (such as **push ccr** to save the condition code register on the stack) are not implemented on the 68070 microprocessor within most CD-i players. This is managed correctly through an illegal instruction trap. Thus not all illegal instruction exceptions cause crashes or should be considered bugs.

3.2 Introduction to the NIRD utility disc

The NIRD system includes a utility WORM which is used for a number of debugging functions as is explained later in this chapter. The disc contains a very small application, the source for which is given in Appendix K of this guide.

The main function of the application is to communicate the module directory of the CD-i player through the cartridge to the host PC. The mechanism for this is described in the next chapter in this guide. The application is very small and executes quickly so it does not set up the screen or display any messages. Once the application has finished sending the module directory data to the cartridge it crashes with an address error to act as a debugging example.

3.3 Finding what caused the crash

The NIRD's most useful debugging facility is called the inquest facility, and this shows you exactly what a CD-i player was doing up until a particular bus cycle. If this 'trigger' bus cycle is an access to the error exception vector area of memory then the system can show you what lead to a crash.

When using the inquest debugging facility, you define a trigger condition that the cartridge looks for in real time on the expansion bus of the CD-i player. While waiting for this trigger condition the cartridge stores information about every bus cycle in a circular buffer large enough to hold data about 16384 bus cycles. When the trigger cycle is detected data saving stops and you can upload the required amount of data from the cartridge to the host PC for disassembly and analysis.

The sequence for using the inquest debugging facility is thus as follows:

- Specify a trigger cycle.
- Play your title disc and wait for the trigger to be detected.
- Upload the required amount of data.
- Disassemble the uploaded data.
- Analyse the data.

Start by apply power to the CD-i player and insert the NIRD utility disc. On the host PC make a new directory for the debugging session and copy into this the symbol table (**.STB**) file for the application. This is the **CDI_MDIR.STB** file found in the **EXAMPLE** directory of the installed software. Now start up the host software by typing nird at the DOS prompt and verify that the debugging system is ready for use by selecting the **System** | **Statistics** menu item and checking that the cartridge supplies information about the version of the cartridge operating system.

3.3.1 Specify a Trigger cycle

To specify an inquest trigger, select the **Inquest | Trigger** menu-item. This will produce a trigger selection dialog box as shown in Figure 2.

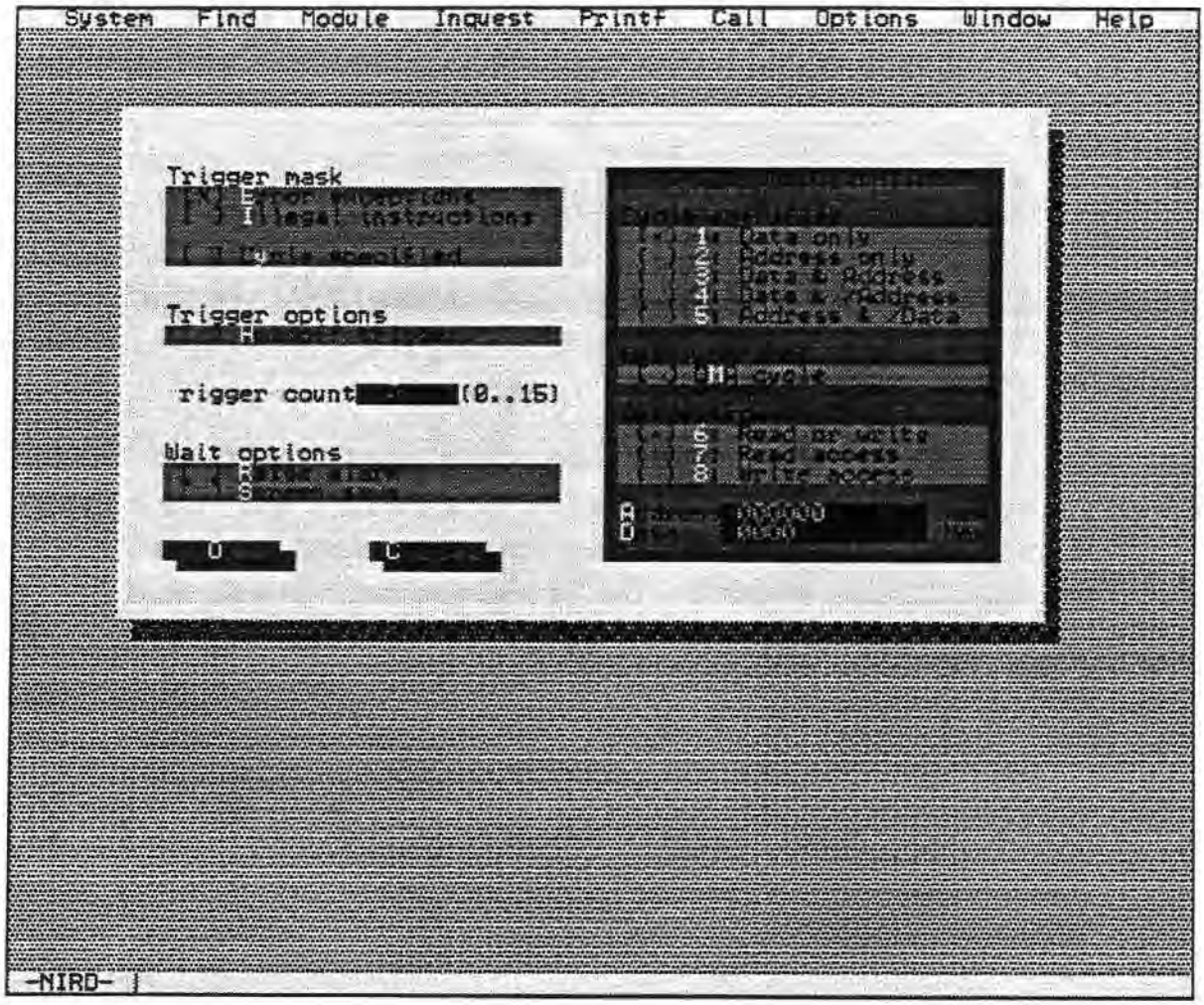

Figure 2: Trigger selection dialog box

The trigger mask area at the top left corner allows you to specify up to three possible trigger sources. If **Error exceptions** is selected then the cartridge will trigger on an access to the error exception vector area except for an access to the illegal instruction exception vector area. Accesses to this area can also be used to trigger the cartridge if you select **Illegal instructions**. This trigger is separate as some people use "green" illegal instructions such as **push ccr**. The third possible trigger is enabled by selecting **Cycle specified**, and this allows you to specify a particular bus cycle in the **CYCLE SPECIFICATION** area on the right hand side of the dialog box. You may specify a particular cycle by giving the address and/or data being accessed, whether it was a DMA cycle (for module loading or real-time plays) or not, and whether it is a read, write or any access.

The dialog box also allows you to enter a trigger count which must be reached before the trigger takes effect, or to hold the trigger off for 128 cycles after it is detected. An audible alarm can be set to sound when the trigger is detected, and you may choose to have a screen saver whilst waiting for the trigger.

We are trying to fix a crash, so select just **Error exceptions**, no trigger holdoff, and a trigger count of zero. Select **OK** when the trigger condition is set.

3.3.2 Waiting for a trigger cycle

After you have selected **OK** the cartridge erases its circular buffer memory and then starts checking each player bus cycle against the trigger conditions. When the cartridge has finished initialising, play the application disc to wait for the trigger. For every bus cycle until the trigger is detected, information about the address and data is saved in a circular buffer large enough to hold 16384 bus cycles.

There are several type of bus cycle, and these are described below:

Instruction fetch Every instruction consists of a 16-bit instruction word that the processor reads from memory.

Operand fetch Some instructions need more than one 16-bit instruction word. Here one or more operand words are used to describe the required instruction. An example is the assembly code instruction **move.w MyGlobal(a6),d0**, where the value of **MyGlobal** is a 16-bit word fetched after the **move** instruction fetch.

Manifest cycle Many instructions read or write to memory when they execute. For example, the move instruction above reads a 16-bit value from a memory address where a global variable is stored into a data register. These accesses to memory resulting from the execution of instructions are known as manifest cycles.

DMA cycle The CD-i player uses the DMA mechanism to quickly load data from disc. DMA bursts interrupt normal program execution to quickly move a block of data into or out of memory.

Exception cycle Exception cycles are the reading or writing of exception stack frames. These occur on error exceptions, system calls, and on device interrupts (timer, pointer, CD-drive, etc).

As well as storing the address and data values for each bus cycle, the cartridge also stores control information to identify whether the cycle is a read or write, if it is a DMA cycle, and so on. Finally, it saves a timestamp for each cycle with 50ns granularity so that you can use the captured data to show you how long a section of code took to execute.

When the cartridge detects the trigger cycle it stops saving data. If you have selected to hold off the trigger then data saving stops 128 cycles after the trigger is detected. This allows you to see exactly what happens immediately after a given cycle. After data saving stops the cartridge's buffer holds information about the 16384 bus cycles up to the point that data saving stopped. This number of cycles is equivalent to about 12ms of bus activity, depending on the mix of instructions during this period.

Before the data can be analysed, it must be uploaded to the host PC to be disassembled by the host software.

3.3.3 Uploading the data

When the cartridge detects the trigger condition the cartridge's buffer holds information about the last 16384 bus cycles up to the cycle where data saving stopped.

Analysis of the data involves uploading data from the cartridge to the host PC, and disassembling the data. The more data that is uploaded the further back in time before the trigger cycle you can analyse, and the greater the size of the raw data and disassembled data files. There is thus a trade-off between the amount of data uploaded, and the time and space required for analysis.

To give you maximum flexibility with data analysis it is possible to upload different amounts of data. It is recommended that you upload just the last 1024 cycles as this consists of just 8K of raw data which disassembles to a file about 25K in size. If you decide that you need more data then you can upload a larger data size and disassemble that. The captured data will remain intact on the cartridge until either another debugging function is performed or power to the CD-i player is switched off.

After triggering the software automatically takes you to the **Inquest | Upload** menu item. This produces the dialog box shown in Figure 3.

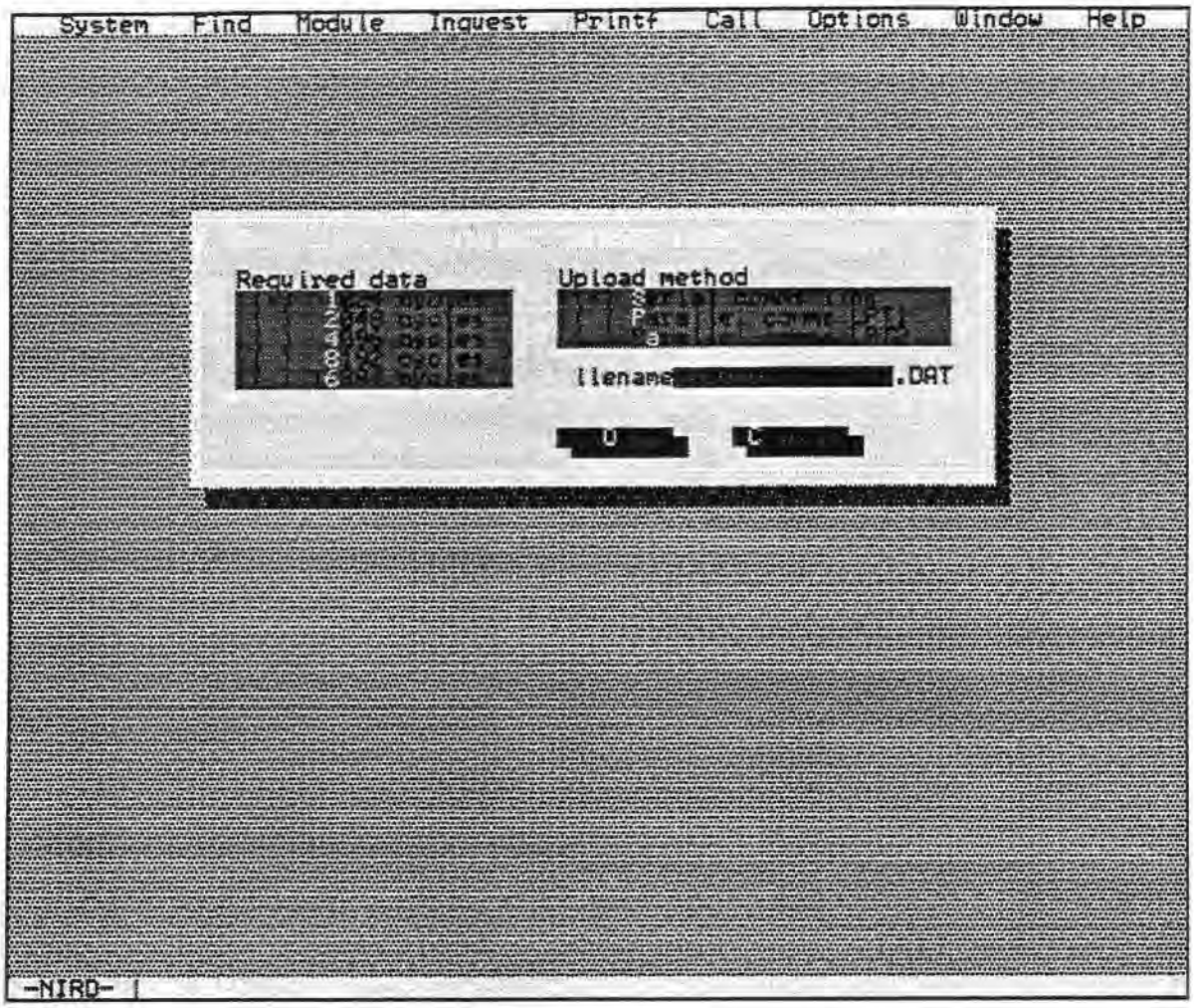

Figure 3: Upload control dialog box

On the left hand side you can select how much data to upload from the cartridge. On the right had side you can specify the method to upload. If your PC has a bidirectional parallel port then use a parallel link as it is much faster than the serial link. Below the upload method selection you may specify the file to upload the data into. Select **OK** to perform the upload.

3.3.4 Disassemble the data

Once the data has been uploaded to the host PC it must be disassembled by the host software before you can analyse it. The disassembly process is very complex as it has to identify each bus cycle as one of the different types described in Section 3.3.2. Furthermore it decodes the processor instructions into assembly language, and identifies the manifest cycles for each instruction. Finally, it is robust so that if a DMA burst, error exception or interrupt occurs during an instruction the disassembler output shows this correctly.

If you have just uploaded data from the cartridge then the host software automatically disassembles the new bus cycle data file. If you wish to disassemble a data file previously uploaded then you may do so using the **Inquest | Disassemble** menu item.

The disassembly process converts a raw bus cycle data file of type **.DAT** into a fully disassembled inquest file of type **.INQ**. A raw data file for 1024 bus cycles disassembles into an inquest file about 25K in size. By way of comparison, a raw data file for the full 16384 bus cycles in the cartridge's buffer disassembles into an inquest file about 425K in size.

Progress of the disassembly operation is shown in a dialog box. This is shown in Figure 4.

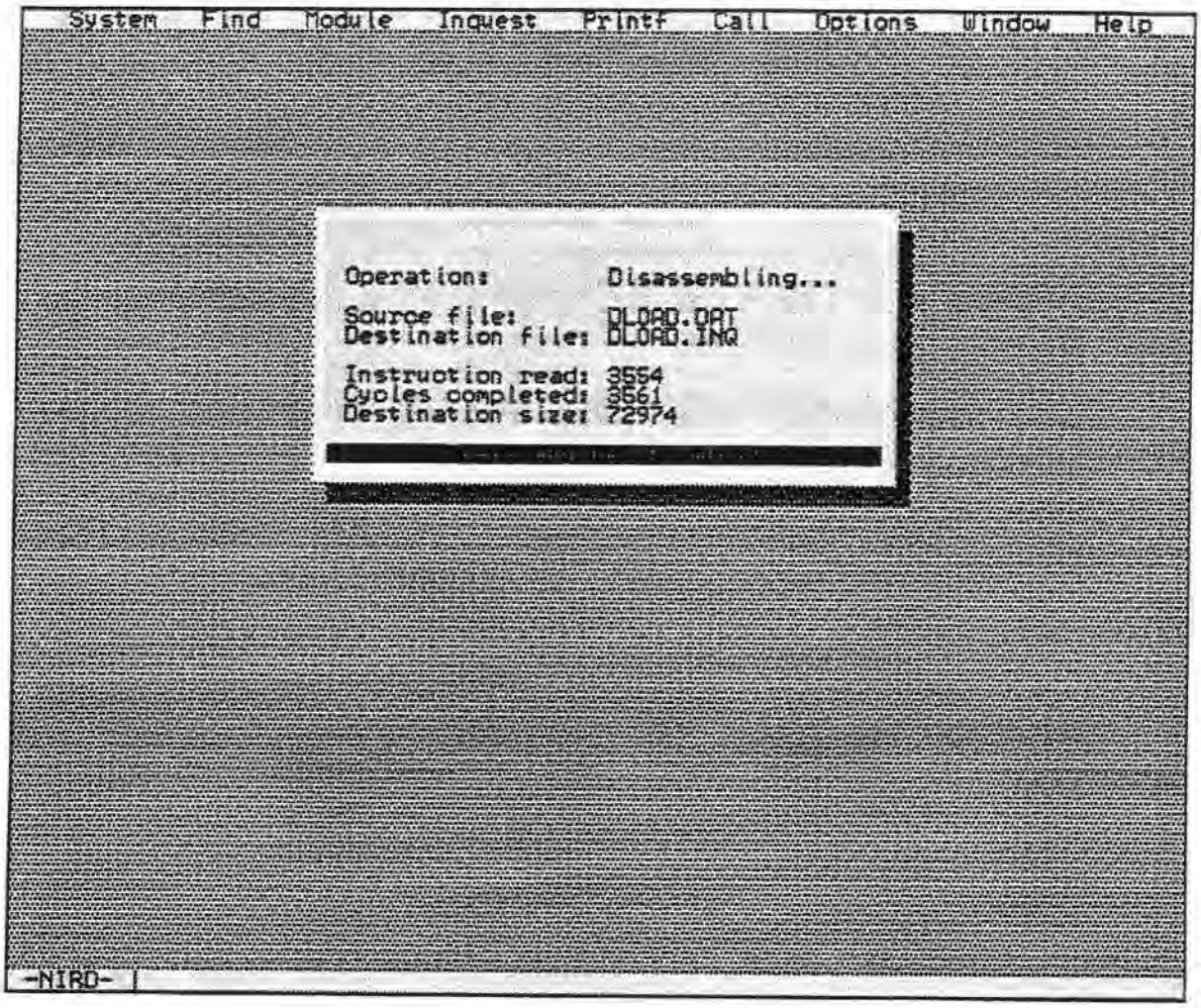

Figure 4: Disassembly progress dialog box

Once disassembly is complete you are ready to look at the disassembled bus cycles to see what caused the bug.

3.4 Data analysis

The disassembled data is in compressed form in an inquest (**.INQ**) file. The host software includes an advanced inquest file viewer which provides colour highlighting of different types of bus cycles and searching facilities. Given information about the position of CD-RTOS modules within the player the host software will show you the module name for each bus cycle. If the symbol table for a module is known (**.STB** file) then it will show you the program label for accesses within that module. Finally, if an application has been produced in C with the appropriate compiler flags to produce a debug (**.DBG**) file then the host software can show you the line of C-source corresponding to an access within that module.

3.4.1 Basic analysis

If you have just disassembled a data file then the host software will automatically load the corresponding inquest file. To load an inquest file at any other time select the **Inquest | Load INQ file** menu item. Figure 5 shows some inquest data for the NIRD Utility Disc (distribution file **CRASH.INQ** in the **EXAMPLE** directory).

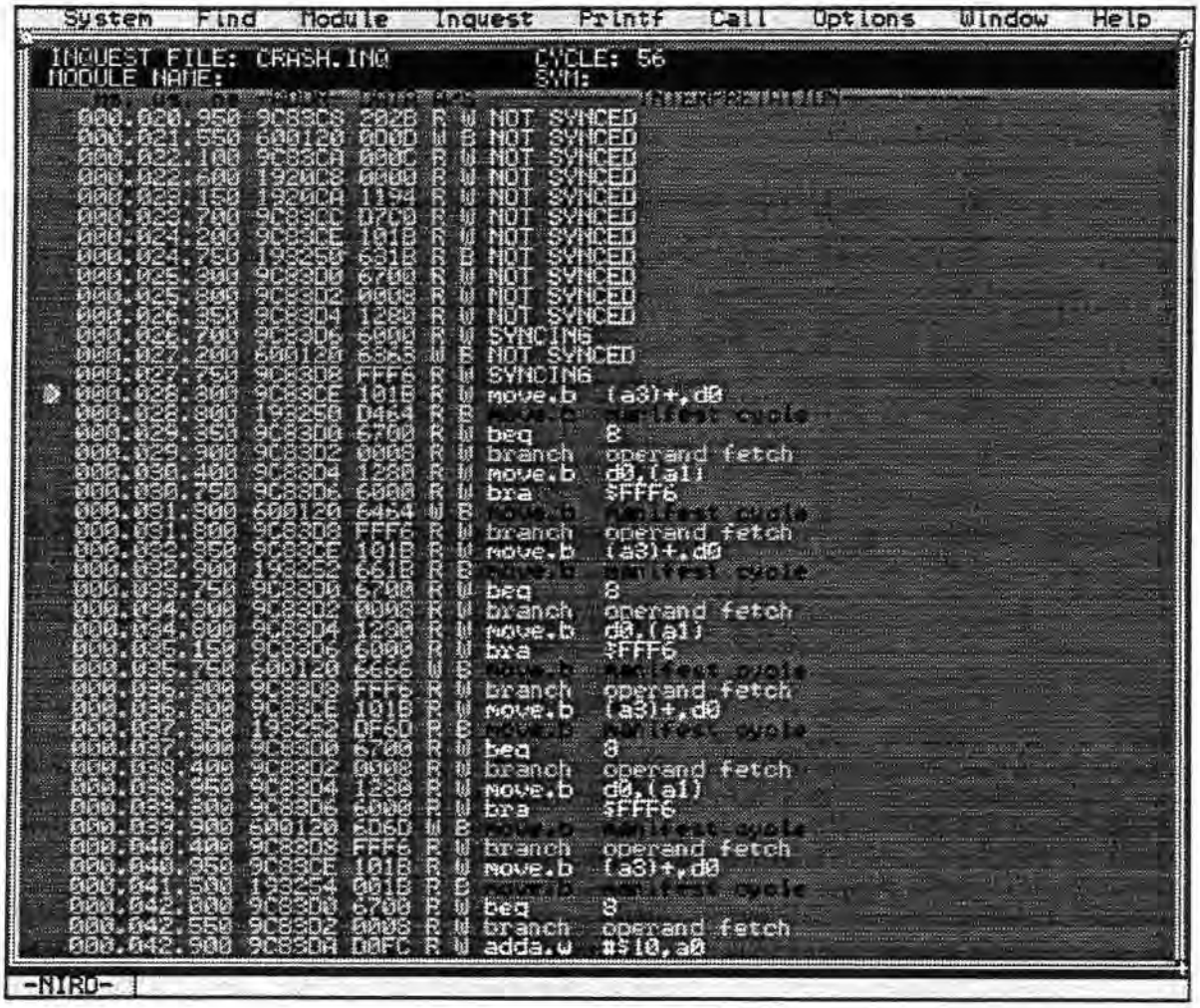

Figure 5: Inquest data from NIRD utility disc

You can move through the inquest data using the cursor up and down keys, the page-up and page-down keys, the home key and the end key. Figure 5 shows inquest data with the cursor at cycle number 56, just as the disassembler has synchronised with the data. Disassembly can only start part-way through the data as it first has to synchronise to determine which cycles are instruction fetches, which are manifest cycles, and so on.

The inquest viewing window shows a row for each bus cycle. Figure 5 shows a number of columns in each row, and you can use the **Options | Inquest** menu item to change what you see. For each bus cycle there are always columns showing the address accessed, the data of the access, the type of access (**R**ead or **W**rite), the size of the access (**W**ord or **B**yte), and the cycle interpretation.

The interpretation column gives you a colour coded explanation of the bus cycle, as follows:

Light gray These are the cycles at the start of the inquest data. The disassembler synchronises on the first change of program flow that it can verify, and so usually synchronises between the first and fiftieth cycle.

White Fully disassembled instructions. This cycle may be a complete instruction word in itself, or may include operands fetched during an associated operand fetch.

Dark gray These are operand fetches. The data of these is used in fully decoded instruction cycles so they can usually be ignored.

Black These are manifest cycles caused by an instruction executing. For example, move instructions could read and/or write to memory locations, subroutine branches and returns save and restore the calling program counter on the stack, and so on.

Blue These are DMA cycles and occur when the CD-i player is using the DMA facility to quickly move data around the system, usually during module loads or real-time plays.

Yellow These are exception cycles and occur whenever an exception stack frame is written to or removed from the stack. This occurs during error exception processing, all system calls, and hardware interrupts.

Red These cycles (usually accompanied by an interpretation of "???") occur if the disassembler becomes confused. This should only ever happen at the end of the data where data saving may cease part way through an instruction.

By looking at the manifest cycles and the instructions that caused them it is possible to determine the contents of registers and memory addresses at any given point in the execution. To find out what triggered the cartridge move to the place in the data where the trigger occurred. This will be either at the end of the data, or 128 cycles before the end of the data, depending on whether you selected to hold off the trigger. It is possible that you will be able to identify the code where the trigger occurs and fix a bug simply using this information. However, just looking at the disassembled data does not show you if the trigger even occurred in your code - it could be in one of the ROMed system modules. The next thing to do is to work out where the various code modules are in the memory map.

3.4.2 Determining player modules

Every CD-i player has many player modules supplied with it in ROM devices. These include the kernel module, device descriptors and drivers, and so on. The NIRD provides a simple way to get information about the module directory of any CD-i player. This uses the program on the NIRD Utility Disc.

To get the player module directory insert the NIRD Utility Disc into the consumer player and select the **Module | Detect mdir** menu item. This presents you with a dialog box where you may enter the name of the Player Module Directory (**.PMD**) file that the module directory data will be stored in. It is usual to make this a name that usefully describes the given player, such as **220NDV40**. This example describes the player as a 220/40 model without a Digital Video cartridge connected. When you are happy with the file name select **OK**. While monitoring the player for module directory data a dialog box is displayed which shows how many modules the host PC has received data about. If you now play the NIRD Utility Disc then the application on the disc will communicate the player's module directory through the cartridge to the host PC. When all the data has been received you will be prompted to press a key. The file will then be loaded into a green module directory window where it can be browsed using the normal navigation keys. A column is shown for each module of its start address, length, type and name. Other columns show the symbol table file name if loaded, and whether compiler debug data has been loaded.

3.4.3 Determining application modules

Every title must consist of at least one application module. These are loaded off the CD or from the emulator at run-time and the operating system decides where to load them. Given application modules should load into the same addresses on subsequent title runs on a given player, but may load in different addresses on different players.

The NIRD system canto watch modules linking into the module directory as this occurs in real time. To do this the cartridge first works out where the module-linking code in your player is, and then watches this execute as the title is run. Module information is shown in a window on the screen, and is saved in an Application Module Directory (**.AMD**) file. To watch modules linking, insert the NIRD Utility Disc into the CD-i player and select the menu item **Module | Watch loads**. This presents you with a dialog box where you may enter the AMD file name. When you are happy with the file name select **OK**. You will be asked to play the NIRD Utility Disc twice. During the first play the cartridge detects the location of the module-linking code in the operating system. During the second play the cartridge detects the first module that is linked after a player reset. This is so that it can detect resets while watching your application. You will then be prompted to play your application disc and any detected modules are shown in the scrolling window.

Detection stops when you press a key or when the player resets. The **AMD** file is then loaded into the module display window with application modules shown in yellow. As the names of application modules cannot be detected they are given default name **appmod1**, **appmod2**, and so on. To correctly name the module move the cursor to the correct module and press return. This presents you with a name changing dialog box. After changing the modules name (in this case to **cdi_mdir**) the AMD file is automatically updated.

3.4.4 Using symbol tables

Once you have player module directory and application module directory information loaded every bus cycle in the inquest data that accesses code space should show the name of the associated module in the **MODULE** column. Remember that the columns you wish to see are selected in the dialog box obtained by choosing the **Options | Inquest** menu item. Module information shows you which module is executing for each instruction/ operand fetch bus cycle. However, to know which function within your code module is being accessed the software needs to use symbol information. If you use the MicroWare linker with the **-g** option then a symbol table (**.STB**) file is produced. This has the format described in the MicroWare OS-9 technical manuals, and describes all of the program and data labels for a module.

To load information about a symbol table file select the **Module | Load symbols** menu item. This presents you with a file chooser dialog box where you can select the STB file that you wish to load. The software determines which module the STB file is for from a field within the header of this file. The file is only loaded if the module is defined in the module data window. While the file is being loaded the names of the labels being read are shown briefly on the screen. Furthermore, the data within the module is also written in readable text form into a temporary file called **SYMBOL.TXT** which is loaded into a browsable window when symbol table reading is completed.

Once the symbol table has loaded a **LABEL** column entry is displayed for each code reference in the inquest data window. For the Utility Disc crash this is shown in Figure 6.

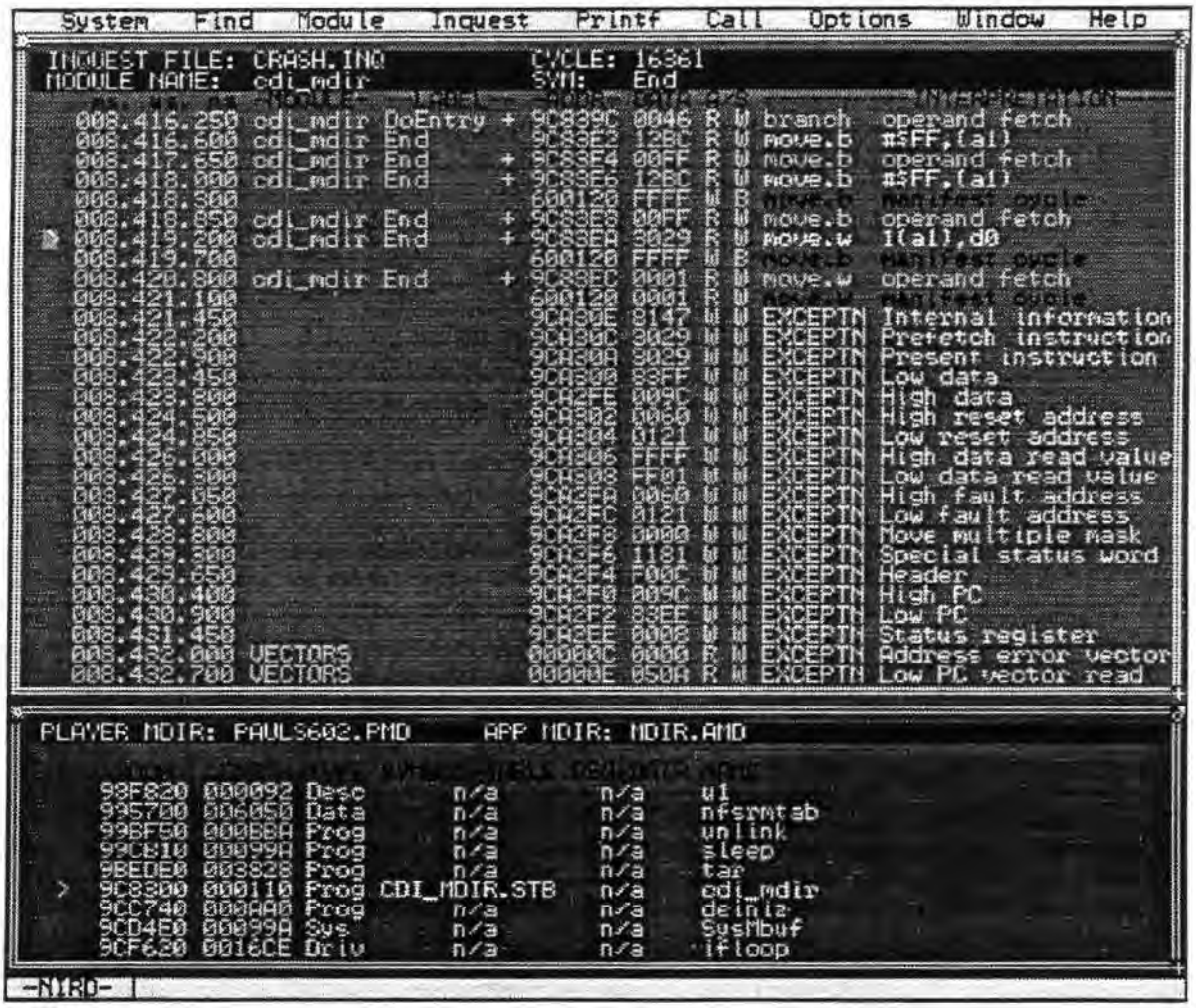

Figure 6: Module and symbol analysis for the NIRD Utility Disc crash

In Figure 6 the host software is showing an inquest data window in the top two-thirds of the screen, and the module data window occupies the bottom one-third of the screen. Note that the module data window shows that the file **CDI_MDIR.STB** has been loaded for the module **cdi_mdir**.

The inquest window has columns for the module and label information.

At the point of the crash (address error shown by error stack frame being written on the stack followed by an access to the address error exception vector) the code was executing from the **cdi_mdir** module so the crash is due to the application on the NIRD Utility Disc. The label information shows the function/ program area currently executing. The NIRD Utility Disc application was written in assembly code as one large function broken up with labels. The '+' character at the end of label names indicates that the code is executing from an area after the label (or within the scope of a function name in C programs). If the address accessed exactly matches a label address (the label position in the code or the start of a C-function) then there is no '+' character.

The NIRD software shows that the crash occurs after the code with the **"End"** label with an address error. Looking closely at the bus cycles it can be seen that the address error occurs because of the word move manifest cycle from the instruction which the cursor is indicating. This can be seen to be because address register **a1** holds value **0x600120** which is an even value, and a word was read from **1(a1)** which is an odd value.

As symbol tables are so useful to debugging, you are strongly recommended to include each code-module's STB file as part of all your disc images.

3.4.5 Viewing C-source

The NIRD Utility Disc was written in assembly language and so viewing inquest data with label information is the most help that the compiler can give us to help pin-point a bug. However, if a module has C-code components, and the MicroWare compiler and linker were both executed with the **-g** option, then a debug data (**.DBG**) file will have been generated as well as the symbol table file. The debug data file relates lines in the C-source code with addresses in the final CD-RTOS module produced by the linker.

To load debug data select the **Module | Load dbg data** menu item. This presents you with a file chooser dialog box where you can select the DBG file that you wish to load. The software determines which module the DBG file is for from the name of this file. For example, if the file **CDI_WONDER.DBG** is loaded then the software will load it for the **cdi_wonder** module. The file is only loaded if the module is defined in the module data window. While the file is being loaded the names of the required source files are shown briefly on the screen. These source files must reside in the current working directory and must adhere to the DOS "8.3" naming convention.

Once the debug data has successfully loaded a C-source window will show you your source code. As you move the cursor through the inquest window a flashing highlight bar in the source window will show you the corresponding line of C-source. If there is no flashing bar then it means that there is no corresponding line of C-source. This usually occurs inside library functions, and so on. C-source tracing is described in more detail later in this User Guide.

Chapter 4. NON-INTRUSIVE DEBUGGING

This section describes how to use the host software and the different debugging facilities that the Non-Intrusive Realtime Debugger provides.

4.1 Host software overview

The "NIRD" utility is the main software which runs on the PC host. Type **nird** at the DOS prompt to start the software. You will be presented with a work-space screen using the 80 column by 50 row text display mode. At the top of the screen is the main menu and at the bottom is a line for help messages. Highlighted letters on the main menu, sub-menus and dialogue boxes show the key to press for selection. The mouse can also be used to click (left button) on screen buttons or select input areas. There are also hot-key options to most areas which allow fast use of the software without a mouse.

Figure 7 shows the basic workspace screen. If you click on a menu word with the mouse, or hold the **ALT** key down with the highlighted letter key, then a menu appears.

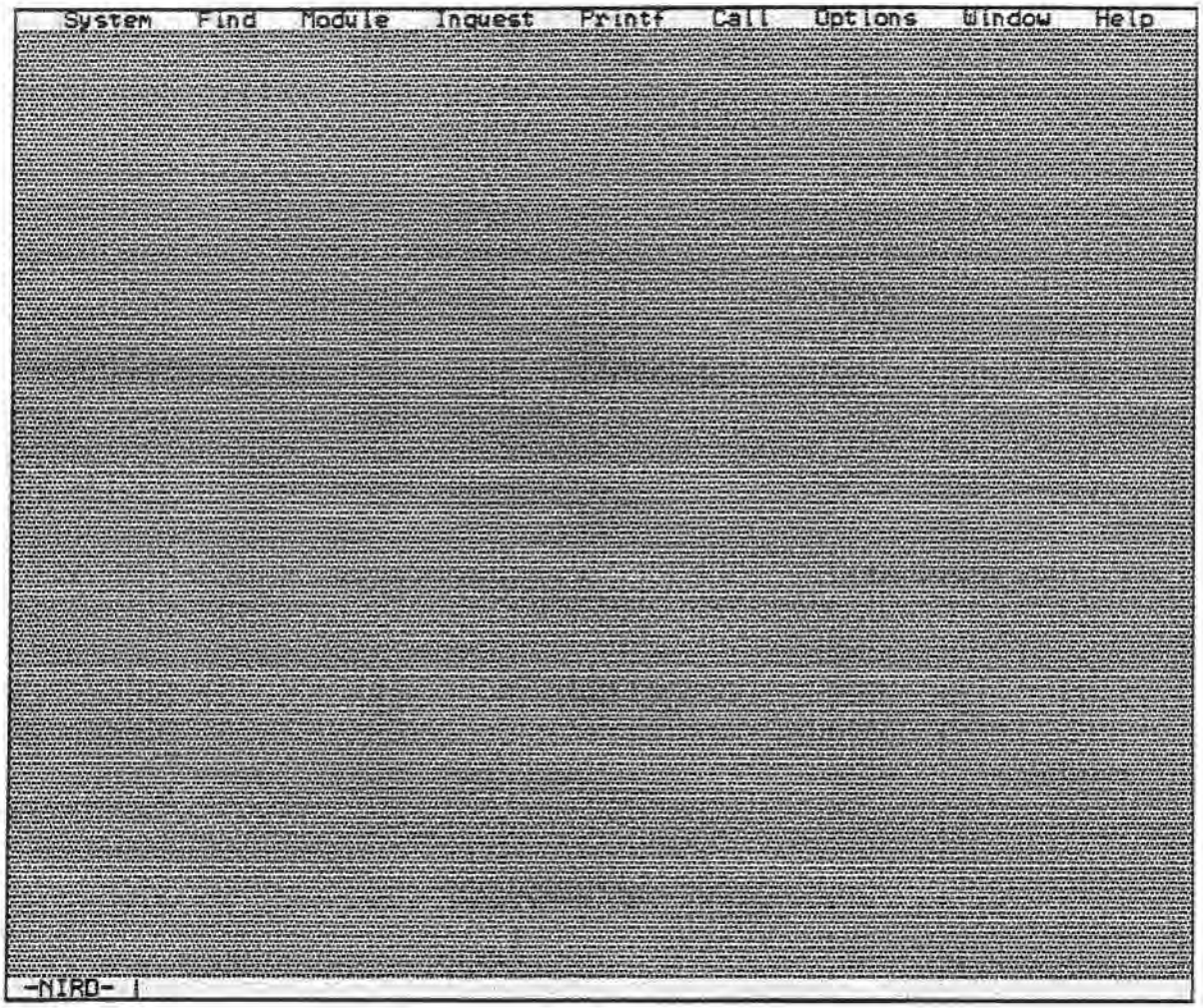

Figure 7: The basic workspace screen

Clicking on another menu word or using another **Alt**+menu letter combination brings up another menu. Clicking in the workspace or pressing **ESC** removes a menu.

The available menus are as follows:

System This menu provides some useful system facilities and allows the user to quit the application.

Find This menu allows you to find specific processor cycles once you have obtained some inquest data.

Module This menu provides facilities concerned with CD-RTOS modules. This includes detecting the module directory list of a given player, watching application modules loading and being created, and defining symbol table files and debug data files for application modules.

Inquest This menu provides access to the inquest facilities. You may specify and wait for a trigger condition, upload data from the cartridge, disassemble inquest data, and load a disassembled data file for analysis.

Printf This menu allows you to either start tracing minimally intrusive printfs into a log file, or to load an existing printf log file.

Call This menu allows you to trace system calls into a log file, or to load an existing system call log file.

Options This menu allows you to change various options within the system.

Window This menu allows you to manipulate windows on the host PC's screen.

Once a menu is visible at the top of the screen you may use the up/down cursor keys to highlight an option within the menu. The left/right cursor keys select the menu to the left or right of the currently active menu respectively. When the desired option is highlighted you may press the **RETURN** key to activate that option. Alternatively, once a menu is visible you may use the highlighted option letter to choose that option. Without even selecting a menu you may choose an option using the associated hot keys as described in each menu box. If you are using a mouse then clicking on the menu word causes the menu to appear, and clicking on an option within a menu activates that option.

A standard file chooser is used to select files to open. The type of file is specific to the debugging task currently being performed. The open file dialog box allows you to see the modification date and size of files before you open them, navigate the directory path, and so on.

Windows on the screen can be manipulated under mouse control, or by using the **Window** menu. When manipulating using the mouse, clicking in the top left corner closes a window, clicking in the top right corner allows the window to be moved, and clicking in the bottom right corner of a window allows it to be sized.

Clicking in a window selects it as the one to receive subsequent keyboard input.

4.2 The inquest facility

This is the most useful debugging facility that the system provides and many of the other facilities are provided simply to support this one. The inquest facility allows you to closely examine what the player is doing up until a given point in time. When you are exploring possible causes of a bug this is usually the first place to start.

When using the inquest mode of debugging the cartridge stores information about the last 16384 player bus cycles in a circular buffer until a trigger condition is detected. Saving of information then stops, and you are able to upload this data to the host PC for disassembly and analysis. The trigger condition is set using the host software, and may be an access to the error exception vector area at the bottom of RAM, a specific bus cycle (described by address, data and control bus values), or when the user presses a key on the host PC's keyboard.

Triggering on an access to the error exception vector area of the player's memory map allows data collection to stop when an error occurs. These vectors are read whenever there is a bus error, address error, illegal instruction error, etc. These sorts of errors typically cause a "blue screen crash" which is produced by the reset code after the error exception vectors are read. The vectors are read by the processor when it detects an error, and so analysis of the cycles immediately before this access will show what caused the crash. Many bugs are caused by null pointer references. Here an uninitialised pointer is dereferenced and this causes a read of the bottom of memory. The cartridge will detect these accesses as they will usually access the error exception vector area of memory by mistake. Please refer to the hints and tips advice in Section 4.6.1 for ideas on reducing the number of null pointer references found in titles.

Triggering on a specific bus cycle is frequently used after the cause of a bug has been found to track it down further. For example, if a global variable pointer has an invalid value assigned to it at some point which later causes a bus error when it is used, then you may want to trigger on the address where that global variable is stored in order to identify when the corruption took place.

Finally, manual triggering by pressing a key on the keyboard whilst the cartridge is waiting for a trigger allows you to stop the cartridge at any point in time. This is particularly useful in cases where a title freezes but does not trigger the cartridge through any other trigger condition. When you press a key, data storage on the cartridge stops and the required amount of data can be uploaded and disassembled to see what the CD-i player is doing. This frequently occurs when a linked list data structure gets corrupted such that the list becomes circular. A list traversal routine may then loop continuously searching for the end of the list. Please refer to the hints and tips advice in Section 4.6.2 for some ideas to help identify where the list corruption is taking place.

4.2.1 Setting a trigger

The first stage of debugging using the inquest facility is to set the trigger condition that the cartridge will looking for whilst storing data in its circular buffer.

This function is performed by selecting the **Inquest | Trigger** menu-item. Selecting this menu item produces a trigger selection dialog box as shown in Figure 8.

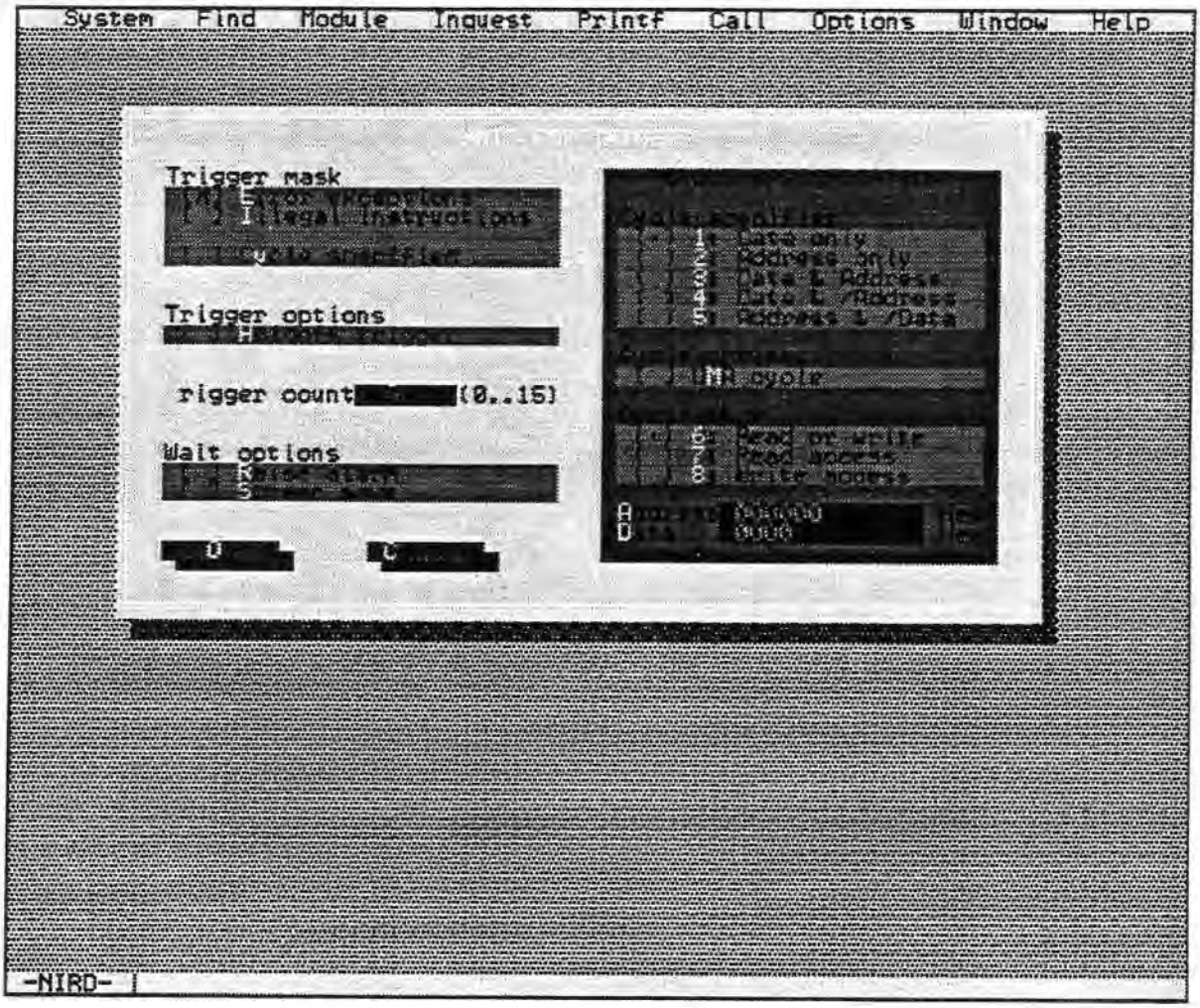

Figure 8: The trigger selection dialog box

At the top left part of the dialog box is the trigger mask selection area. Here you may select which trigger conditions you wish to use. Selecting "Error exceptions" will cause the cartridge to trigger if any of the following conditions are met:

- An access is made to any of the following vector address areas (here partitioned so that you can see their intended use):

0x000008-0x00000B	Bus error
0x00000C-0x00000F	Address error
0x000014-0x000017	Zero divide
0x000018-0x00001B	**CHK** instruction exception
0x00001C-0x00001F	**TRAPV** instruction exception
0x000020-0x000023	Privilege violation
0x000024-0x000027	Trace exception
0x000028-0x00002B	Line 1010 emulation
0x00002C-0x00002F	Line 1111 emulation
0x000030-0x00003B	Reserved
0x00003C-0x00003F	Uninitialised interrupt
0x000040-0x00005F	Reserved
0x000060-0x000063	Spurious interrupt

- A write access is performed to any address in the range **0x000000** to **0x000007**. This address holds the base address of the operating system area, and is used by the operating system every time a system call occurs, for example. If this address is corrupted then the player may not crash until the next system call is made, and so detection of writes to this area is a useful debugging trigger.

Selecting "Illegal instructions" will cause the cartridge to trigger if there is an access to addresses in the range **0x000010** to **0x000013**. This area holds the exception vector for illegal instructions. This range is not included in the standard error exception trigger because some instructions are "green" but are not available on the 68070 processor. An example of this is the **push ccr** instruction.

Instructions such as these are managed by the code vectored-to through the illegal instruction exception. If you are using these instructions then you should not use this trigger condition. This trigger condition is only available if the "Error exceptions" trigger has been selected, otherwise the current "Illegal instructions" option is grayed-out and is not available for toggling.

Selecting "Cycle specified" allows you to trigger on a specific bus cycle using the cycle specification area occupying the right half of the dialog box. You may specify a data and/or address combination using the cycle specifier area to be one of the following:

- Trigger on access of specified data at any address.
- Trigger on access to specified address with any data.
- Trigger on access of specified data at specified address.
- Trigger on access of specified data when not at specified address.
- Trigger on access of specified address when not specified data.

Below the "Cycle specifier" area is a check box describing whether the cartridge should trigger on the cycle if it is a DMA cycle, or not a DMA cycle. This allows you to trigger on specific data being loaded by DMA off disc.

The "Cycle type" area below the "Cycle options" area allows you to specify whether the trigger should occur for a read access, for a write access, or for any access.

At the bottom of the cycle specification area are the data and address specification areas. In these areas you may enter the particular hexadecimal values for the address and/or data buses for the cycle you wish to trigger on.

Having specified the particular trigger condition that you are interested in, you may select the "Holdoff trigger" option on the left hand side of the dialog box to delay the trigger for 128 bus cycles. This enables you to see what happens just after the trigger.

The "Trigger count" area allows you to specify a trigger count before data storage stops. The default value is zero which makes storage stop as soon as the trigger is detected.

The "Wait options" allow selection of screen blanking while waiting for the trigger, and selection of an audible alarm on trigger detection.

When you are happy with your trigger options select **OK**. Trigger detection can start at any point provided the CD-i player has power applied. In other words, you may choose to start trigger detection when the player is at the player shell screen in order to be sure to check all of a title for bugs, or you may choose to start trigger detection once the title has reached a certain point of execution. Note that trigger detection starts after a short reset period lasting about half a second or so. This reset is shown on the host PCs screen after you select **OK**.

While waiting for a trigger condition to be detected a small status dialog box is displayed. If you press a key at any time then the cartridge is manually triggered and you will automatically proceed to the upload data facility. If you have selected the savescreen option in the trigger selection dialog then the screen will be black except for the status dialog box which will slowly bounce around the screen. This is useful, for example, if the title only ever crashes after long periods of time and you wish to preserve the phosphor in your monitor. The status message on the small status dialog box will inform you when the cartridge has triggered. If you have selected the alarm option in the trigger selection dialog then an audible alarm will now sound. Pressing a key cancels the alarm if one is sounding and leads you to the upload data facility.

4.2.2 Uploading inquest data

When the cartridge has triggered you can upload inquest data to the PC for analysis. You are led to this option automatically if the cartridge has just triggered, or you may choose to upload data by selecting the **Inquest** | **Upload** menu-item. This verifies the inquest data and produces an upload control dialog box as shown in Figure 9.

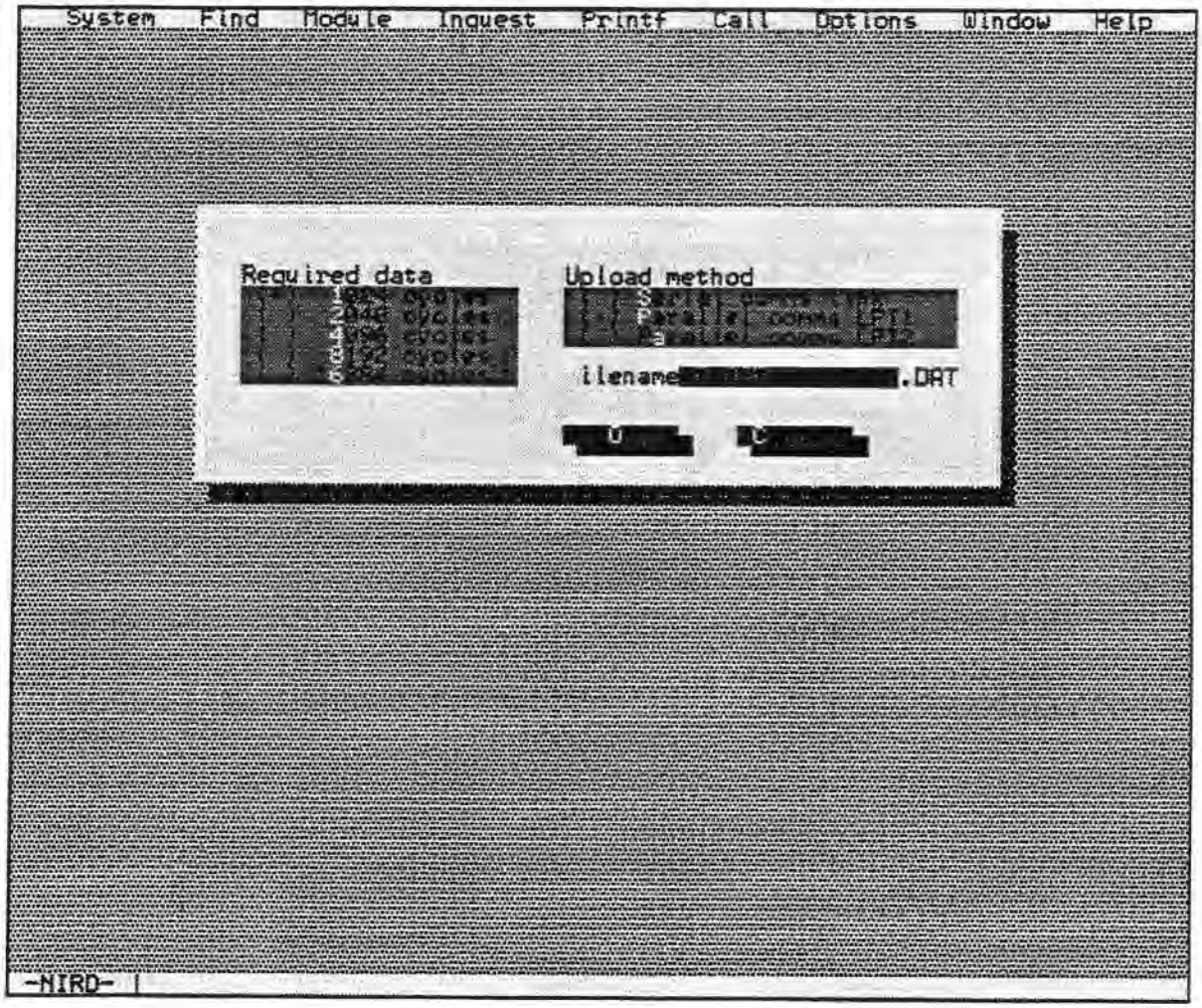

Figure 9: The upload control dialog box

The debugging cartridge maintains a circular buffer of the last 16384 bus cycles. This means that when the trigger is detected, the cartridge has data which describes the last 16384 bus cycles up to the trigger point. This corresponds to approximately 12ms or so of player execution time, depending on the type of code executing and the areas of memory that are accessed.

In order to analyse this data it must be uploaded to the host PC and disassembled. Since the uploading and disassembling tasks take time that is directly proportional to the amount of data, the upload control dialog box allows you to specify how much data you wish to upload. Many problems can be fixed by just uploading the last 1024 cycles leading to the trigger being detected. Others may require more. It is common to upload just the last 1024 cycles to begin with, analyse that data, and if you need more data then upload it using the **Inquest | Upload** menu-item. The data will remain in the debugging cartridge until either CD-i player power is turned off or a new trigger condition is set so that data storage starts again.

On the right hand side of the upload control dialog box is an area where you may choose which method is to be used to upload the data. The fastest way is to use parallel communications, in which case you should choose the parallel port with which the host PC is connected to the debugging cartridge. If bidirectional communication is not supported on your host PC then you must use the serial upload option which is slower.

A filename entry area allows you to specify the name of the file into which the data is to be saved. This has to be into a **.DAT** file which is created in the current working directory. It is helpful to use the name to describe the appearance of the bug (such as **CRASH1** or **FREEZE3**) and to make a note of this file name for later reference. When you are happy with your upload selection select **OK**.

During the upload of inquest data a progress dialog box is displayed. If an error occurs during the upload then an appropriate message will be displayed. The most likely upload error is the attempt to upload data over the parallel link with a unidirectional parallel port.

Once the data has been uploaded into the inquest data file you must disassemble the data before it can be analysed. You will automatically be led to the disassemble data option if you have just uploaded some data. If you go automatically from trigger to upload to disassemble then the disassembler will mark the cycle where the trigger occurred with a "**T**". If you upload more data, or re-disassemble the data, then the trigger mark will not be saved. However, it is usually simple to identify the trigger cycle's position. If you have selected no trigger hold-off in the trigger selection dialog then the trigger cycle will be the last but one cycle in the data, and if you did select trigger hold-off then the trigger cycle will be 129 cycles before then end of the data.

4.2.3 Disassembling inquest data

The data that the cartridge buffers is every read or write access that the CD-i system performs to external memory. These reads and writes are called bus cycles, and may be due to instruction and operand fetches, manifest cycles which are the cycles resulting from instructions which access memory (such as moves, etc), DMA cycles where data is being moved from disc into memory, exceptions and interrupt stack frames being saved or read resulting from things such as the timer, system calls, pointer button presses, and so on.

The data stored in the cartridge buffer has fields for the following information:

- 24-bit address bus value.
- 16-bit data bus value.
- Control signals (read/write, size of access, DMA identifier, etc).
- 16-bit time-stamp.

The format of this data (which is saved in an "event frame" in the cartridge's buffer and is uploaded in the same format) is given in Appendix C. It is the task of the disassembler within the host software to convert the inquest data into a meaningful format. This task is not an easy one. There are no signals available on the expansion connector which describe whether a read bus cycle is an instruction fetch, and operand fetch, manifest cycle, or read of an exception stack frame.

DMA cycles can be identified from a DMA control signal. Non-DMA write cycles could be exception stack frame pushes or manifest cycles. To complicate things further there are memory addresses within the 68070 processor (used for DMA and timer control registers, etc) the access cycles to which do not appear on the external bus. Furthermore, exceptions and DMA bursts can appear at any time, even in the middle of instructions, and the disassembler must not lose synchronisation with the data. This is quite a different situation from that facing a traditional disassembler.

The disassembly function is available using the **Inquest | Disassemble** menu-item. The first thing the disassembler must do is try to decide where to start disassembling. It does this by looking for the first change of program flow that it can verify. Disassembly then starts with each bus cycle being identified. A dialog box shows the progress of the disassembly. The disassembled data is written to a **.INQ** file which varies in size depending on the number of instructions disassembled and the type of bus cycles in the data. As a guide, a **.DAT** file containing data for 1024 cycles is disassembled into a **.INQ** file about 25K in size. By way of comparison, a **.DAT** file for all 16384 cycles in the buffer is disassembled into a **.INQ** file about 425K in size.

4.2.4 Viewing inquest data

One inquest file can be viewed at a time using the **Inquest | Load INQ file** menu-item. The inquest file is displayed in a window with subsequent lines showing subsequent bus cycles. The final lines of the data are the cycles that triggered the cartridge, and thus as you scroll down through the data you are looking forward in time towards the trigger cycle. This means that the more data you upload, disassembled and view, the further back in time from the trigger cycle you can analyse.

For each bus cycle you can select which data you wish to view. The data is shown in vertical columns down the window. The available data is as follows:

Timestamp This column shows a millisecond, microsecond, nanosecond timestamp for each bus cycle with 50ns granularity.

Module This column shows a truncated module name (limited to eight characters) for each bus cycle. This information can only be shown if the address accessed during the bus cycle corresponds with a known module as defined in a **.PMD** file or a **.AMD** file (see later). A module name truncated to 16 characters is shown at the top banner row of the inquest data window as you move the cursor through the data.

Label This column shows a truncated label name (limited to eight characters) for each bus cycle. This information can only be shown if the address being accessed during the bus cycle corresponds with an access to a module which has had a **.STB** file loaded for it. A label name truncated to 32 characters is shown at the top banner row of the inquest data window as you move the cursor through the window.

Address This column shows (in hexadecimal) the address being accessed for each bus cycle.

Data This column shows (in hexadecimal) the data being read or written to the given address during each bus cycle.

ASCII character This column shows the ASCII equivalent of the two data bytes making up the data bus for each bus cycle for printable ASCII characters. If the bytes are not valid printable ASCII then this column entry is left blank.

Access This column shows whether the access was a **R**ead or **W**rite cycle.

Size This column shows whether the access was a **B**yte or **W**ord cycle.

Interpretation This column shows the colour-coded disassembled instructions and interpretation for each bus cycle.

The timestamp, module, label and character columns are optionally visible using the selection dialog obtained using the **Options | Inquest** menu-item.

Figure 10 shows an inquest window at the start of some inquest data.

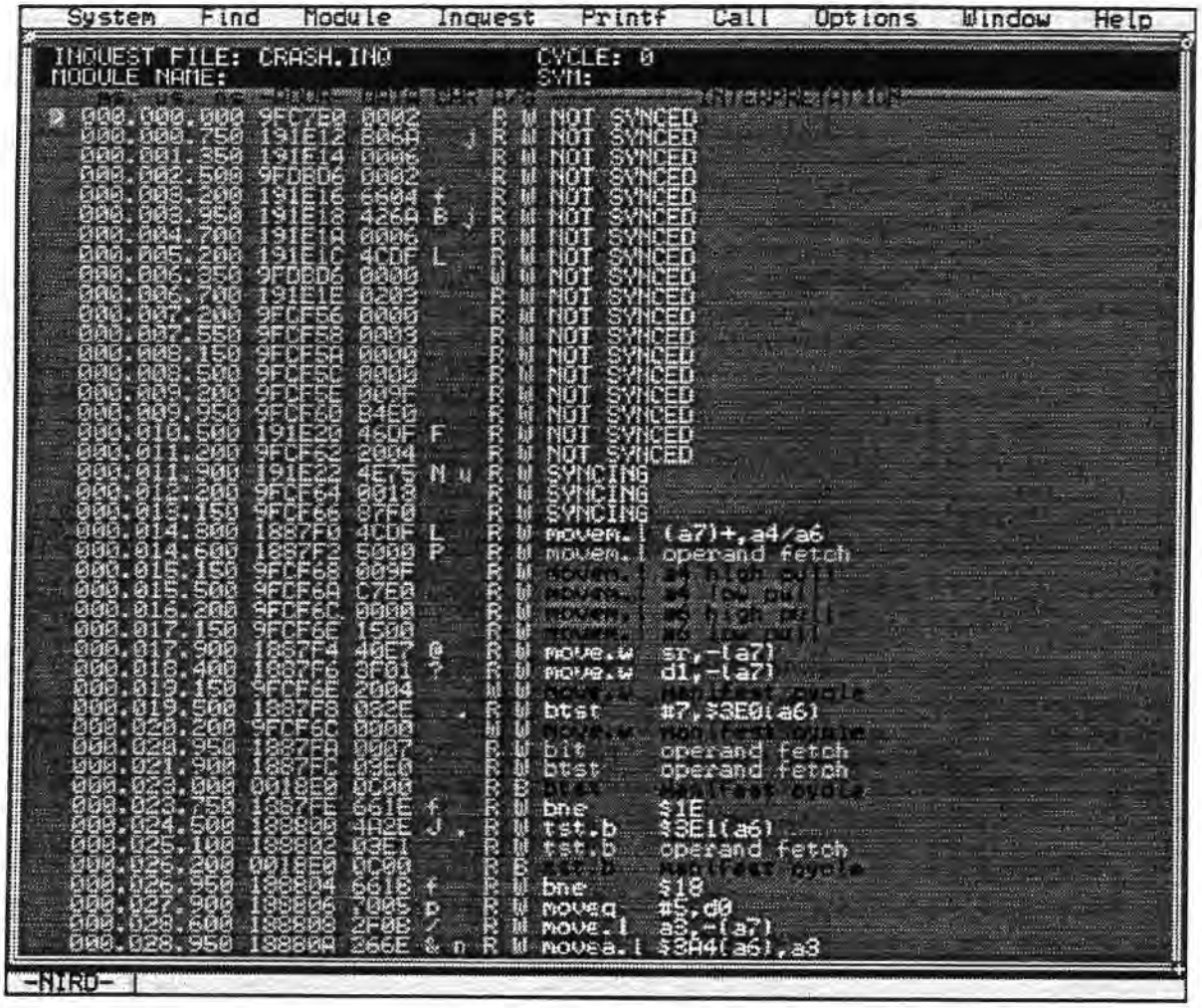

Figure 10: Viewing inquest data

At the top of the inquest window you can see two lines of header information. On the top line is the filename being shown, and the bus cycle number within the inquest data. The cursor (white > symbol next to the timestamp column for the first row of data) is at the top of the data and so the current cycle number is zero. The second header line shows the full module name and symbol name for the current cycle. This is provided as the columns of module and symbol information are truncated to eight characters. If this information is not available for a given bus cycle (as in the case shown in Figure 10) then these fields are left blank.

At the start of the inquest data the disassembler has not synchronised with the data. Synchronisation takes place at the first change of program flow that can be verified. In this case you can see that this occurs over three "SYNCING" cycles. These have data value **0x4E75**, **0x0018**, and **0x87F0**, which corresponds with the instruction **RTS** and a stack pop of address **0x001887F0**. Sure enough the cycle following these is a read from address **0x1887F0**. From this point on the interpretation column shows, with colour highlighting, what the bus cycle was for and the disassembled assembly code. Synchronisation typically occurs anywhere between the first cycle and cycle 50. However, it may take longer if there are no changes in program flow. Cycles before and during synchronisation are shown in light gray.

During normal program execution there are three types of bus cycle. Instruction fetches and fully decoded assembly instructions are shown in white, operand fetches (cycles that form part of an instruction such as displacements) are shown in dark gray (their information is used in the fully disassembled white cycles), and manifest cycles resulting from instructions which access memory are shown in black. Other types of bus cycles are DMA accesses as data is loaded from disc (shown in blue), exception bus cycles resulting from errors, hardware interrupts or exceptions (shown in yellow), and disassembler problem cycles (shown in red). If you see problem cycles anywhere other than at the end of the inquest data then it means that the disassembler has become confused, and you should email the **.DAT** file to the NIRD support service (see Section 6 of this document).

To scroll through the data you can use the up and down arrow keys to move the cursor a bus cycle at a time, page up and page down to move a number of bus cycles, and the home and end keys to skip to the start and the end of the data respectively.

Sometimes when debugging you may want to skip to a certain bus cycle, or find the next (or last) access to a certain address. These searching facilities are available from the Find menu. The **Find | Goto cycle** menu-item allows you to jump to a certain cycle number. The **Find | Find cycle** menu-item allows you to define a search pattern (address, data, etc) to look for in a forwards or backwards direction from the cursor. Figure 11 shows the dialog box that allows you to specify the search conditions.

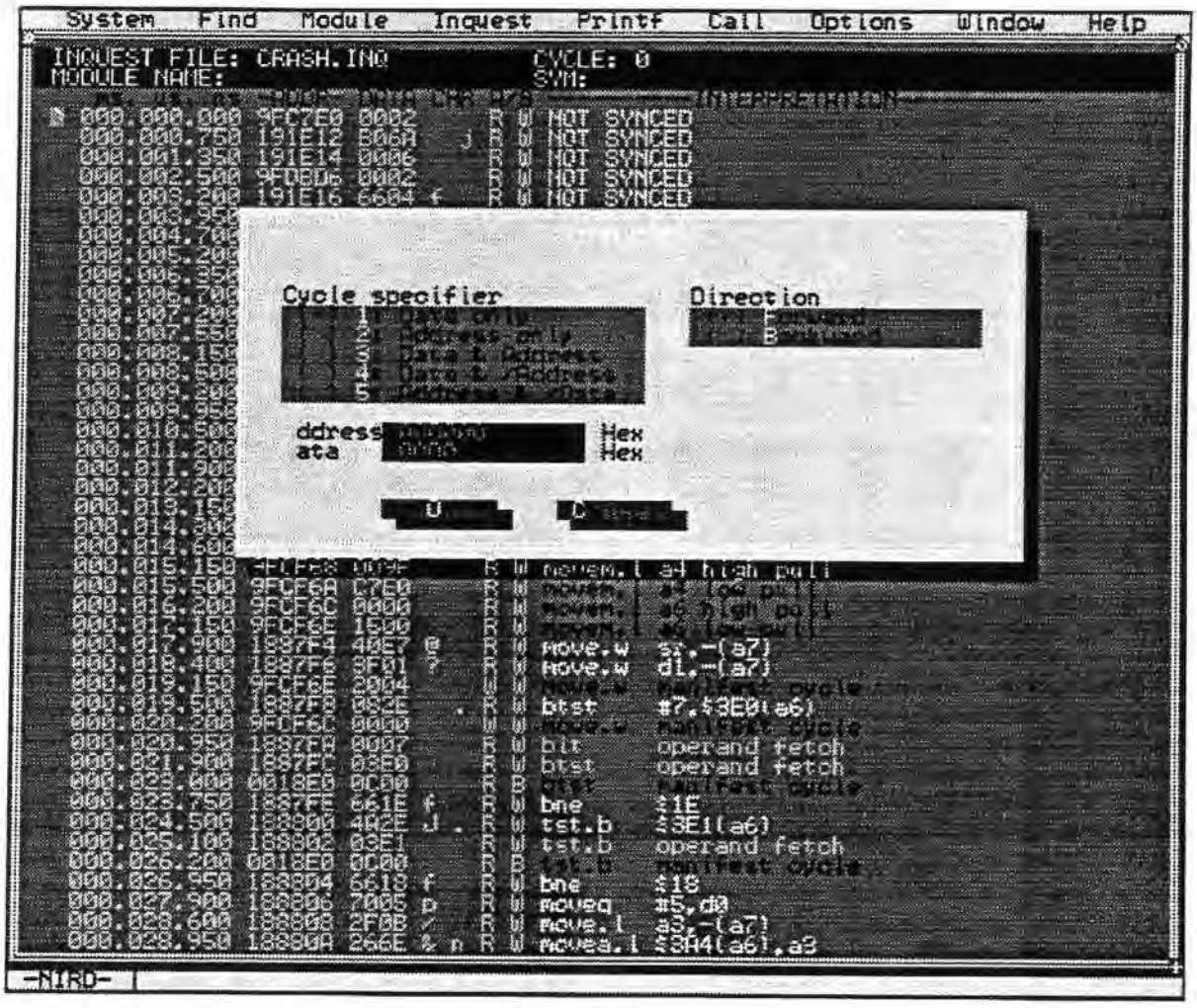

Figure 11: Defining the search specification

The **Find | Find next** menu-item allows you to search for the next match of the previously-defined search specification.

The data display in the inquest window can seem confusing at first, and some knowledge by the user of 68070 assembly code is a definite requirement. However, it is not just the sequence of instructions that can take some effort to understand, but also the order of the bus cycles and their correlation with the instruction sequence. This difficulty arises because modern processors such as the 68070 used in CD-i players use the bus as efficiently as possible. This means that memory manipulations (or manifest cycles) from a given instruction may be held off while the next instruction opcode is being interpreted.

Figure 12 shows a section of inquest data in the inquest window. This is the same data, but at a position a few cycles towards the trigger point at the end, as is shown in Figure 10 and Figure 11. After the synchronisation cycles at the top there is a move multiple (**MOVEM**) instruction followed by a bit-mask operand fetch to show which registers to move. This is followed by the required registers being moved (in this case read) from memory.

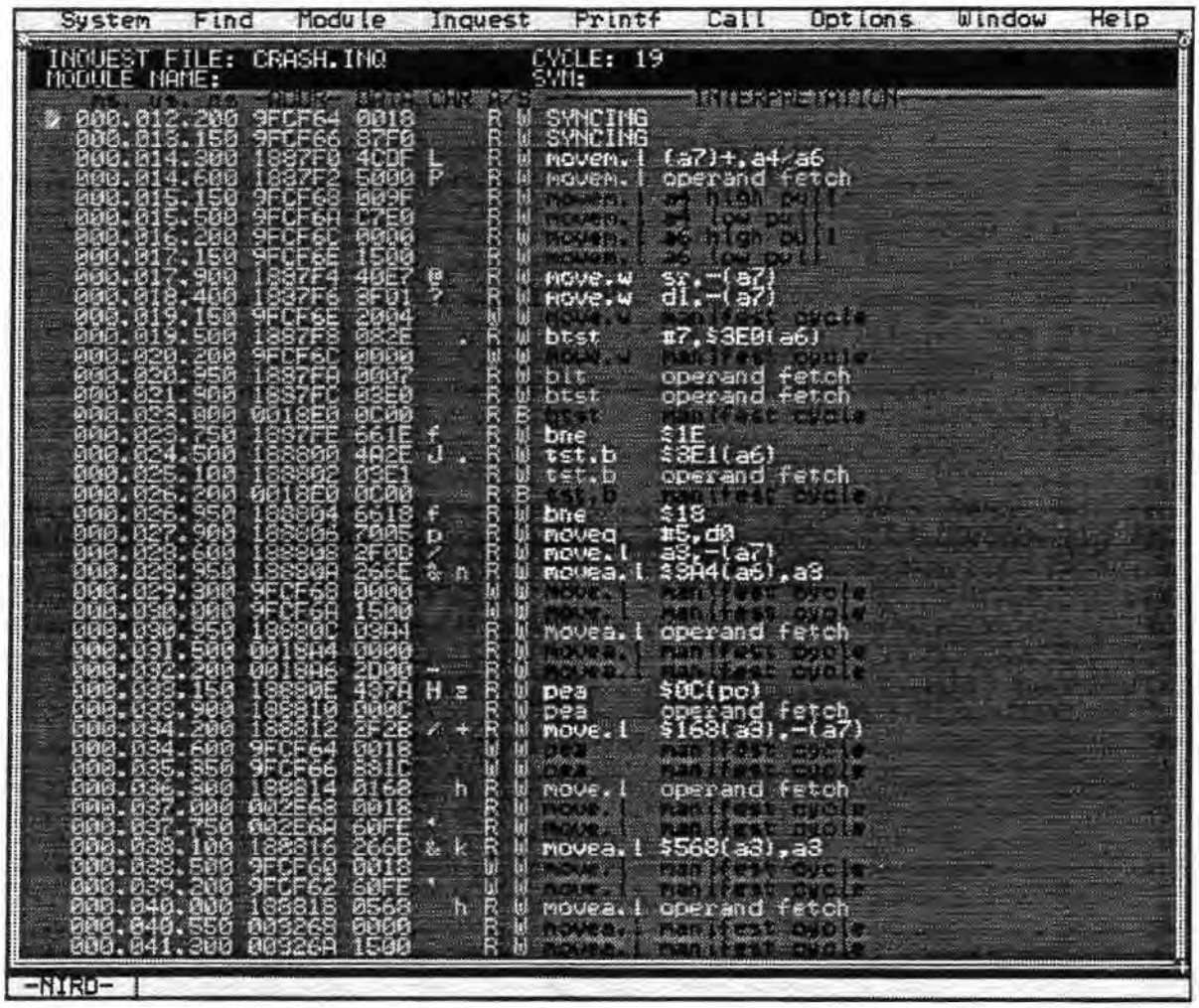

Figure 12: Example inquest data

After the move multiple instruction there is an instruction to save the status register on the stack (**move.w sr,-(a7)**). However, whilst you may expect to now see a write cycle as the status register is written into memory, instead there is another instruction read. This is an instruction to save the lower word of data register **d1** onto the stack (**move.w d1,-(a7)**). Here we have an example of the memory access being delayed until after the next instruction opcode has been read. Then, while the new instruction is analysed by the processor, the status register is written into memory. Now the write of the data register into memory is delayed until after the next instruction fetch, which is a bit test opcode. The bit test requires two operands (one showing the bit to test, and the other the address displacement) and these are read after the save of the data register.

This interleaving can seem confusing at first, but once you have seen some examples of it you will become used to identifying what the CDi player is actually doing. Manifest cycles can be very useful in debugging as they usually show what the contents of a register is. For example, the move multiple in the above example read values for address registers **a4** and **a6** off the stack, so you also know the value for the stack pointer register **a7**. The save of the status register allows you to identify what condition codes were set at that point, whether the system was in user mode or supervisor mode, and so on. Tracing back register contents from triggers on error exceptions fixes many of the common bugs causing address and bus errors. Seeing where an address register was loaded with the value zero can fix most null pointer references.

Previously mentioned in this section on non-intrusive debugging were processor-internal manifest cycles. These result from instructions accessing addresses within the actual 68070 processor. Registers within the processor control modules such as the serial ports, DMA controllers, timer, and so on. When an access is made to these addresses (residing in the memory space **0x80000000** to **0xBFFFFFFF**) the manifest cycles are not visible on the external bus.

An example of this is shown in the timer interrupt code shown in the inquest data window of Figure 13.

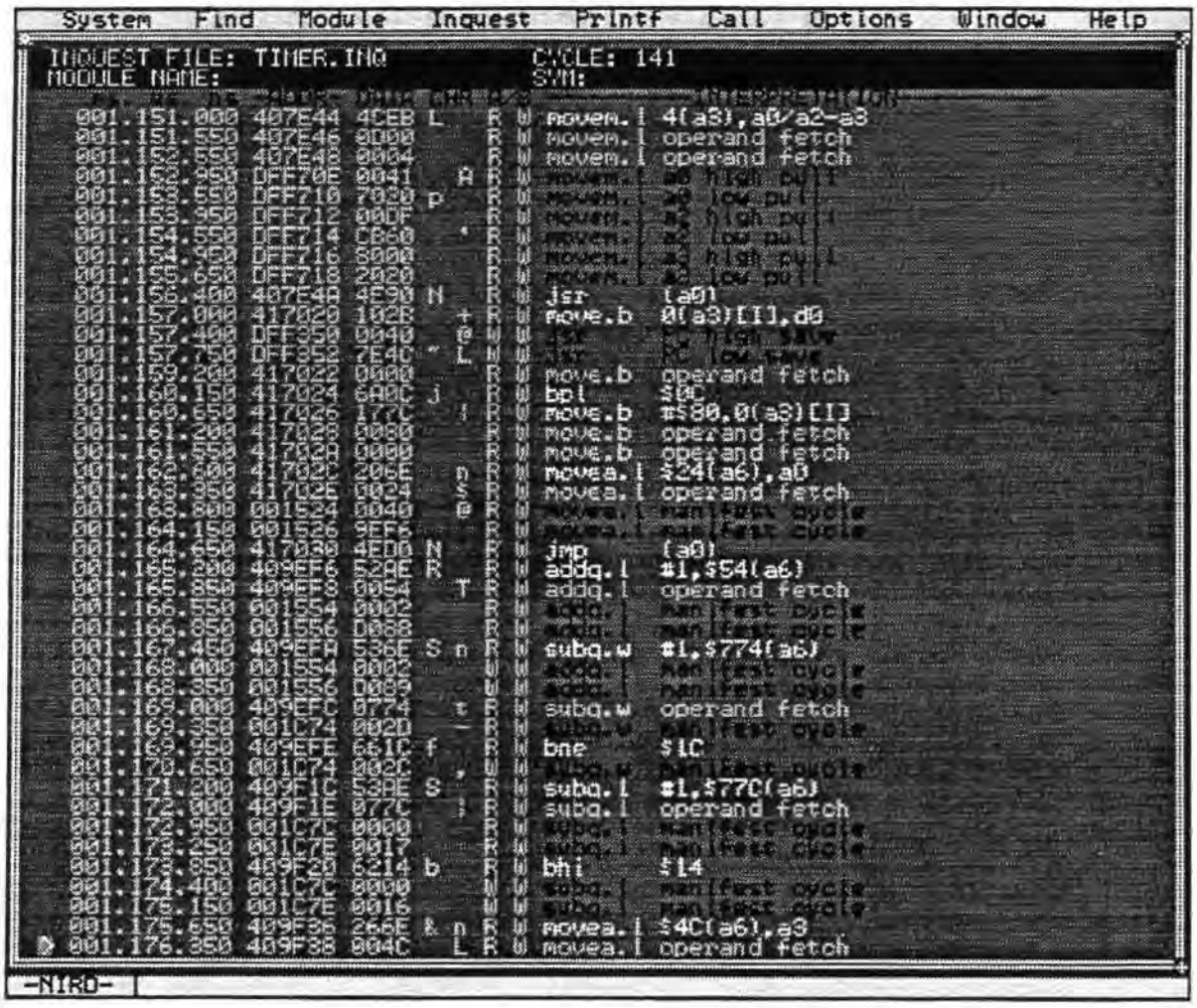

Figure 13: Example of processor-internal manifest cycles

At the top of the window is a move multiple instruction which causes address registers **a0**, **a2** and **a3** to be loaded from the stack. Note that **a3** is loaded with the value **0x80002020**, which is the address of the timer status register. There then follows a jump subroutine (**jsr**) to the address in **a0**.

◆ Note that the following instruction (a byte read into a data register) is read and decoded while the return address is stored on the stack. This move byte instruction (**move.b 0(a3),d0**) is reading the timer status register and so there is no visible manifest cycle for this instruction. This means, unfortunately, that we do not get to see what the value of the status register is. Note also the waste of a bus cycle as the null displacement is read as an operand fetch. Instructions which result in processor-internal manifest cycles have a [I] marker included in the interpretation for each manifest bus cycle that is internal. Here it is a byte read and so there is a single [I], long reads will show [I][I] for clarity.

4.3 Module, symbols and C-source

The inquest data viewing window allows you to see exactly what the CD-i player is doing at any point in time. However, a disassembly output showing just the addresses being accessed is usually not enough to fix a bug. To be really useful you need to know which addresses correspond with which CD-RTOS modules that are in the system. Furthermore, if the software knows where your application is loaded in memory, then it can use **.STB** and **.DBG** files to pinpoint exactly where in your code corresponds with a given bus cycle.

4.3.1 Detecting player modules

Every CD-i player comes with a number of CD-RTOS modules in ROM. These are all the operating system modules, device driver modules, and so on. Even if you are not debugging system module code it is useful to know the player module directory of the player on which you are debugging. For example, if in the middle of your code there is a timer interrupt then you will be able to see the processor switch to execute from the timer module code. Information about a given CD-i player's module directory is stored in a textual **.PMD** file. The format for this is given in Appendix E. Whilst it could be generated by hand if you knew the required information, it is much easier to let the host software build the file for you. Player module directory detection is performed by executing the application on the NIRD utility disc in conjunction with an option in the host software.

To detect a player's module directory select the **Module | Detect mdir** menu-item. This presents you with the dialog box shown in Figure 14.

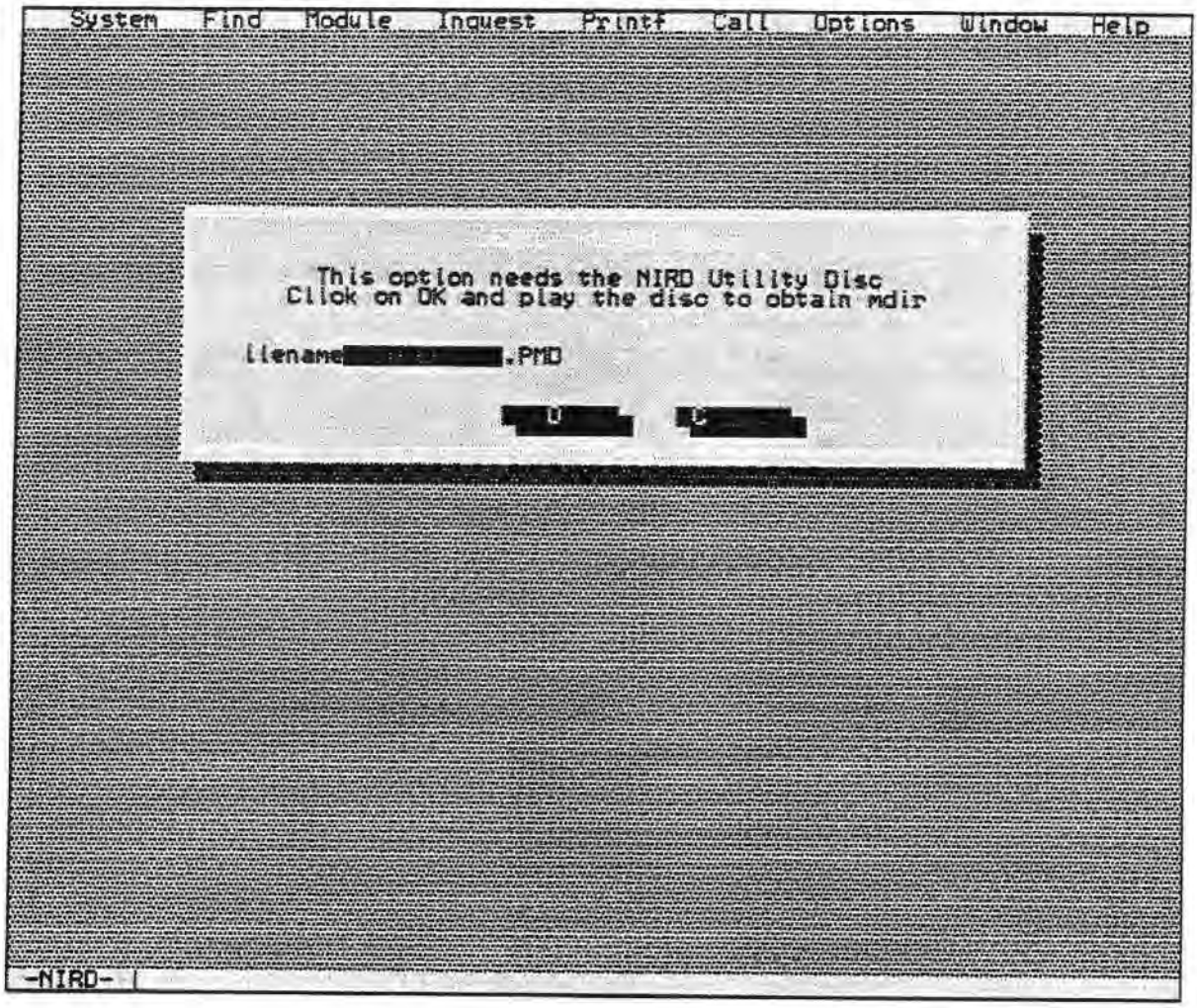

Figure 14: Detect player module directory dialog box

The filename entry area allows you to specify the **.PMD** filename for the player. It is usual to make this a name that usefully describes the given player, such as **220NDV40**. This example describes the player as a 220/40 model without a Digital Video cartridge connected.

When you are happy with the filename make sure that the player is at the startup shell and select **OK** to continue. After selecting **OK** a small dialog box will show how many modules have been detected. If you press a key whilst modules are being detected then detection is aborted.

In order for the cartridge to detect player modules you need to play the NIRD utility disc that was supplied with the unit. This is a small application (the source code can be found in Appendix K of this document) which gets the module directory into a buffer using a system call, and then communicates the data to the cartridge by writing it to a global variable that the cartridge is monitoring. The cartridge then send the data down to the host PC where you will see an incrementing count in the small dialog box. After communicating the module directory to the cartridge the application deliberately crashes. This is used as a debugging example in Section 3 of this document.

After all of the module directory has been received by the host PC, the **.PMD** file is built and then loaded. The module directory window is shown in Figure 15, and may show modules defined by a single **.PMD** file for the player's modules, and a single **.AMD** file for an applications modules.

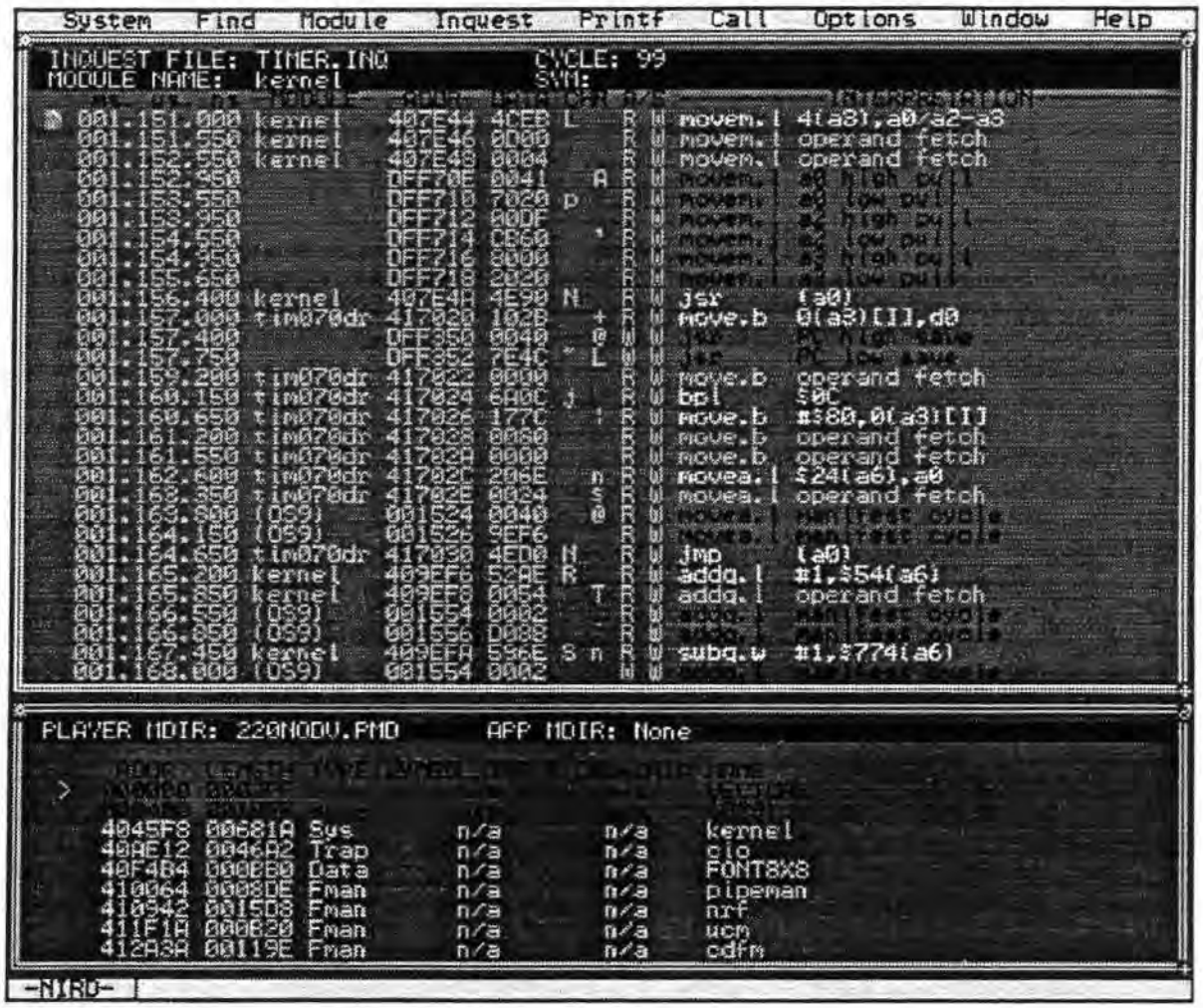

Figure 15: Inquest and module display windows

At the top of the window is a title bar which displays the name of the **.PMD** and **.AMD** files that have been loaded. Below this is the module data itself. This consists of a banner describing the displayed columns, and a scrollable module list. For each module there is a column showing start address in hex, length in hex, the type of the module, the name of a loaded symbol table file that has been loaded for the module, the name of the source debug data file that has been loaded for the module, and the modules name.

The displayed modules are colour coded to show their origin. Player modules detected using the utility disc are shown in gray, application modules (described in the next section of this document) are shown in yellow, and the host software already knows about the exception vectors and the CD-RTOS global variable data which are shown in black.

The module window list of modules is organised in ascending memory space order. Module types are in numerical form in the **.PMD** file, but these CD-RTOS defined numbers are displayed as text names in the type column of the module window. Possible types are:

Prog for program modules,
Subr for subroutine modules,
Mult for multi-modules,
Data for data modules,
Resv for reserved types,
Trap for user trap library modules,
Sys for system modules,
Fman for file manager modules,
Driv for physical device drivers,
Desc for device descriptor modules, and
User for user defined modules.

Looking closely at Figure 15 you will notice that the inquest window at the top of the screen shot contains the same data as was used in Figure 13 to show the meaning of processor-internal manifest cycles. This code is executed whenever the timer interrupt occurs. Notice now the extra column in the inquest display window for the module name. This shows which module each player bus cycle is executing from. At the top of the window the cycles are executing from the **kernel** module. A jump subroutine is made into the **tim070dr** module, and this accesses some CD-RTOS global variables before jumping back into the **kernel** module. Cycles where no module name is given show accesses into free memory - in this case the stack.

If you select the module display window (either by clicking in it with the mouse or by using one of the menu-items in the **Window** menu) then you can use the up and down arrow keys, the page up and page down keys, and the home and end keys to scroll through the module directory.

If you debug using exactly the same player (same model such as 220, and same generation such as /25) again then you do not have to re-detect the player's module directory. Simply use the **Module | Load PMD file** menu-item to load the existing file. Note that you need separate module directory files for a given player if it is used with and without a DV cartridge. The DV cartridge has ROMs that contain the Digital Video drivers.

4.3.2 Detecting application modules

If the host software knows where your application modules have been loaded then it can use symbol tables and source debug files to show you exactly which parts of your code correspond with given bus cycles.

Information about an application's module directory is stored in a textual **.AMD** file. The format for this is given in Appendix F. Whilst it could be generated by hand if you knew the required information, it is much easier to let the host software watch modules loading off disc or being created by an application and build the file for you. The NIRD detects modules being linked into the player's module directory by finding the location of the module header parity evaluation code for your player. This code evaluates the module header parity for a given module each time a link is performed. By watching this code execute the cartridge is able to see application modules being linked. As parity evaluation does not include the name storage area of a module it is not possible for the cartridge to detect module names.

To detect a player's module directory select the **Module | Watch loads** menu-item. This presents you with the dialog box shown in Figure 16.

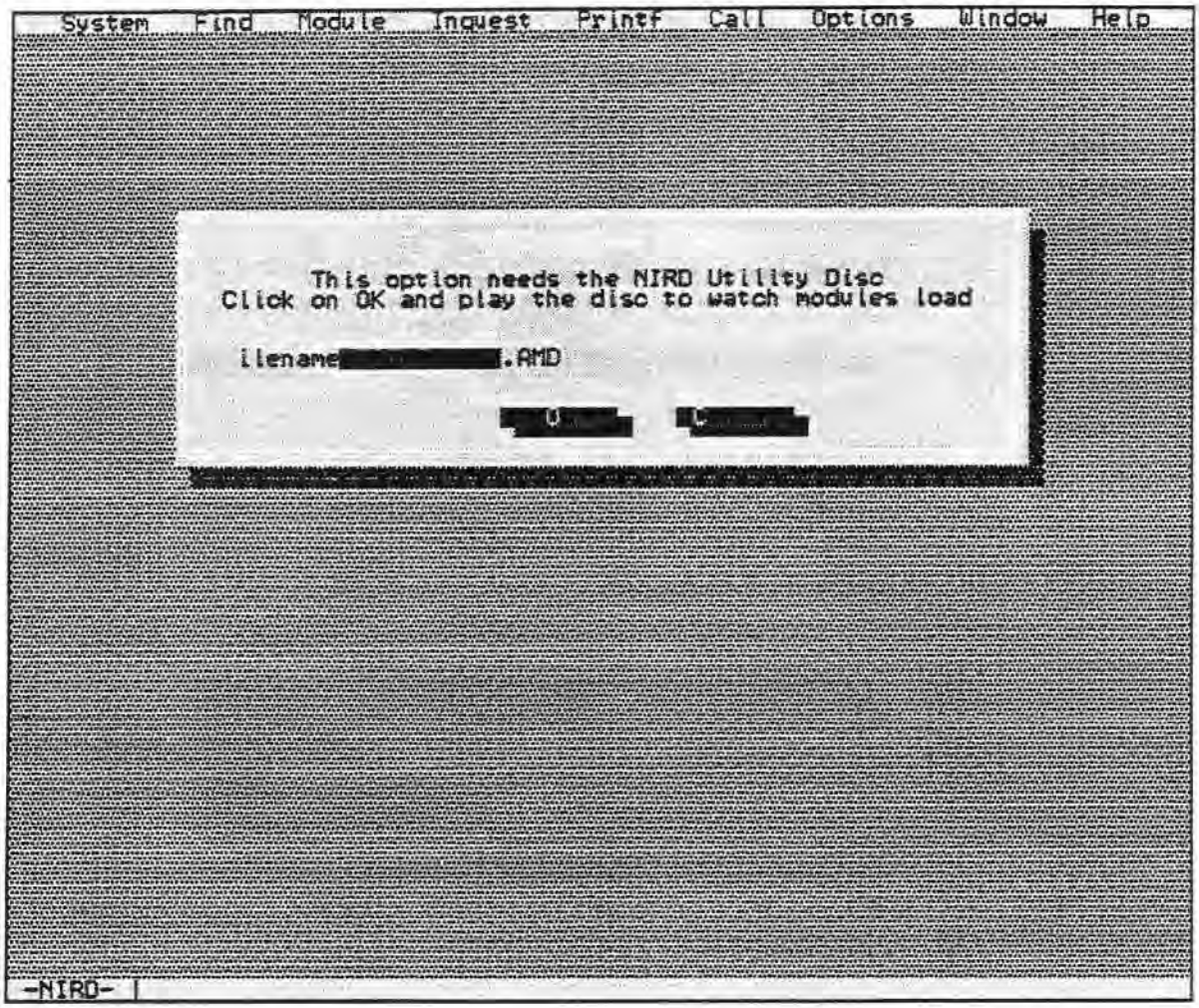

Figure 16: Detect application modules dialog box

The filename entry area allows you to specify the **.AMD** filename for the title. When you are happy with the filename make sure that the player is at the startup shell and select **OK** to continue. After selecting **OK** you will twice be asked to play the NIRD utility disc.

The first time this disc plays and resets the player the cartridge identifies the module header parity evaluation loop for the player. The second time the disc plays the cartridge detects the first module to be linked when the player resets. This is so that a reset can be detected.

After the two plays of the NIRD utility disc you will be asked to play your application disc. As modules are loaded off disc or created by the application their information is shown in the scrolling module list of a dialog box. This is shown in Figure 17.

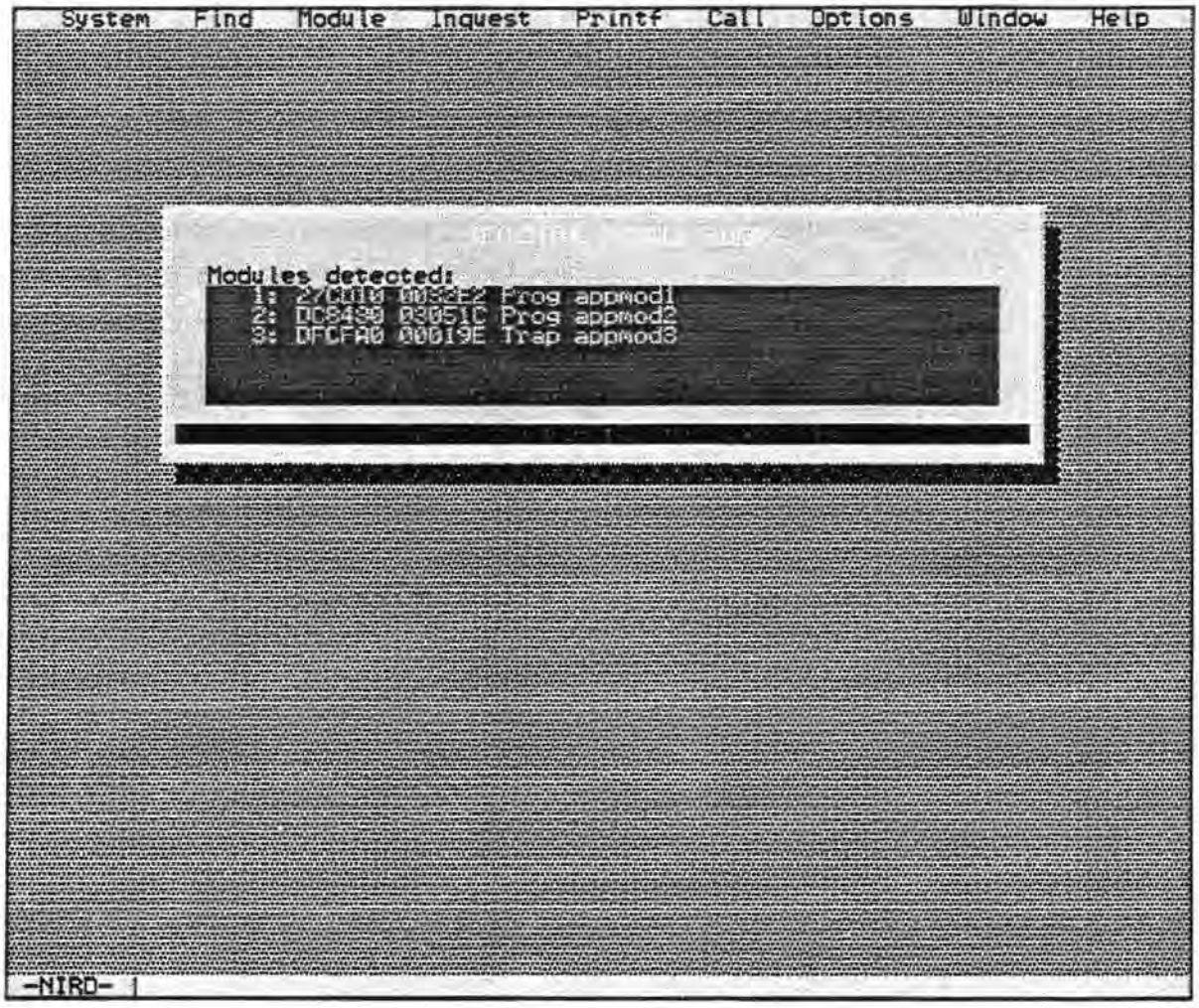

Figure 17: Scrolling module display in dialog box

- Note that for each module the start address, length and type are shown. The name of each module is allocated by the host software in the form **app**lication **mod**ule #. This name is allocated as the cartridge cannot detect the real module name.

  When the cartridge has detected all the required modules press a key for the **.AMD** file to be built. Module detection will stop automatically if the player resets at any point.

  After the **.AMD** file has been generated it is loaded into the module display window provided that another **.AMD** file has not been previously loaded.

  The final task to be performed is to rename the application modules to their real names.

  This is can be achieved be manually editing the **.AMD** file but the host software provides an easier way to do it. Use the cursor keys in the module display window to align the cursor with an application module and press return. This will produce a dialog box where you can change the name of the module. Do this for each module in your application.

This change name dialog box is shown in Figure 18.

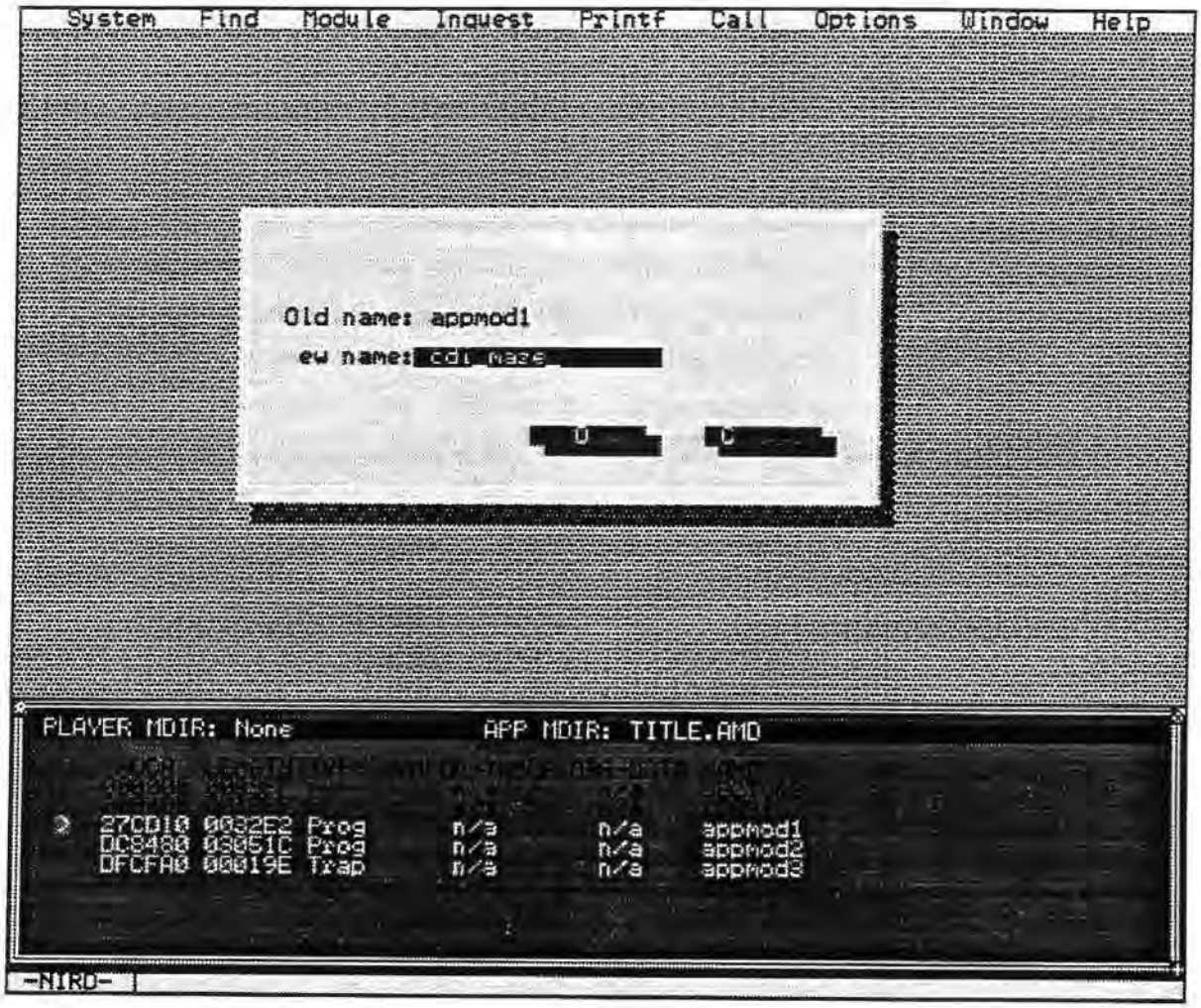

Figure 18: Modifying an application module's name

The application module names should be known to you! However, if you have lots of modules and are unsure about the order of loading/size of them then use the **ident** utility to make sure that you can identify them correctly. This is particularly true for modules which have been concatenated together.

Once all of the application modules have been detected and named correctly you should be able to see their names references in the module column of the inquest window for accesses within their respective memory spaces. Be aware that just as the player module directory file is only valid for a particular player (same model such as 220, and same generation such as /25), so the application module directory is only valid for a single player also. Whilst a given player will always load application modules in the same position, there is no guarantee that other players will load into the same place. The consequences of an incorrect application module directory are that symbol and source debug references may be invalid.

4.3.3 Loading symbol tables

Using a module memory map of the player the inquest data display window can identify which module each bus cycle is accessing. If you provide a symbol table for a module then another column in the inquest data display window can show the label in your code corresponding to the address within each module.

The host software can only use symbol table's adhering to MicroWare's symbol table format. This is described in the MicroWare OS-9 Language manuals. The symbol table is produced automatically if you link with **-g** included in your options.

To load a symbol table file for a module select the **Module | Load symbols** menu-item. This will produce a file opening dialog box where you may select the symbol table file to be read. The software determines which module the symbol table is for from a field within the symbol table file itself. Provided a module with this name is displayed in the module directory window the symbol table will be loaded.

The name of each symbol is displayed on the screen as loading proceeds. Once loading has been completed the symbol table file name is shown in the **SYMBOL-TABLE** column of the module directory window against the appropriate module entry. This can be seen against the **cdi_demo** module in the module directory window of Figure 19.

Figure 19: Host software workspace after loading a symbol table

Whilst the symbol table is being loaded, the data is stored in a temporary file in the current directory with the name **SYMBOL.TXT**. This file is automatically loaded when the symbol table file has been processed, and can be seen occupying the top two thirds of the host software workspace in Figure 19.

This text file is very useful as it shows you the type and values of all the symbols in the symbol table file. The host software uses the program text symbols to identify the labels corresponding with each access within a module in the inquest data viewing window. However, you can use the uninitialised and initialised data symbols (ie: your globals) to identify which variables are being accessed in the inquest data viewing windows. Global variables are usually referenced from address register **a6**, so if you see an instruction such as **move.l $8126(a6),d0** in the disassembled data then you would know that the code was trying to access global variable **FctID** in this case. You may scroll through this text file to look at its contents, but there are currently no searching facilities available within it.

If you now look within an inquest data window at accesses within a module which has a symbol table loaded then you will see the appropriate program label in the label column. This is shown in Figure 20. For direct matches with a program symbol (eg: **NIRD_Str**) just the truncated symbol name is shown. For accesses within the scope of a symbol (eg: **main**) the label is followed by a '+' character.

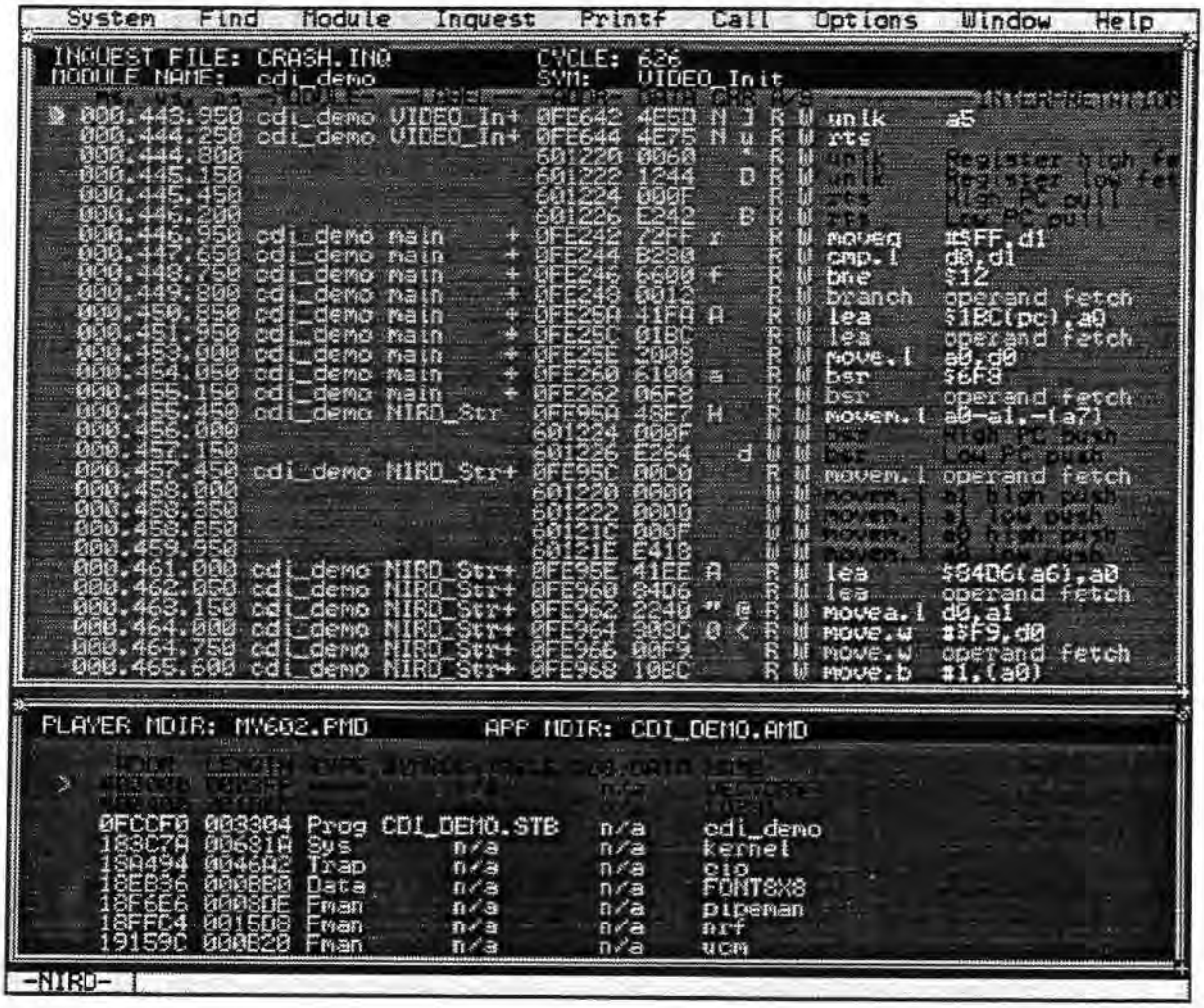

Figure 20: Program symbols shown in the inquest data window

4.3.4 Viewing C-source

Many users will be able to debug simply by using inquest data together with the information that the symbol table provides. The label column shows the code function being accessed, and the globals being used are shown by comparing the offsets to the global pointer register (such as the access to **$84D6(a6)** in Figure 20) with the data in the **SYMBOL.TXT** window. Similarly, uninitialised variable locations may be found by adding the associated offset to the value of the stack pointer. However, there may be occasions when it is useful for the host software to actually show you the line of source code that corresponds to a given application bus cycle. The relationship between an application module and the corresponding source code is given in the **.DBG** file. This is produced by the linker if the compilation and linking processes are both performed with the **-g** option.

To load a debug data file for a module select the **Module | Load dbg data** menu-item. This will produce a file opening dialog box where you may select the debug data file to be read. The debug data file always has the same name as the CD-RTOS module it corresponds to, but it has the **.DBG** filetype. Provided a module with the corresponding name is displayed in the module directory window the debug data file will be loaded. The name of each associated source file is displayed on the screen as loading proceeds.

Once loading has been completed the word "**LOADED**" is shown in the **DBG-DATA** column of the module directory window against the appropriate module entry. This can be seen against the **cdi_demo** module in the module directory window of Figure 21.

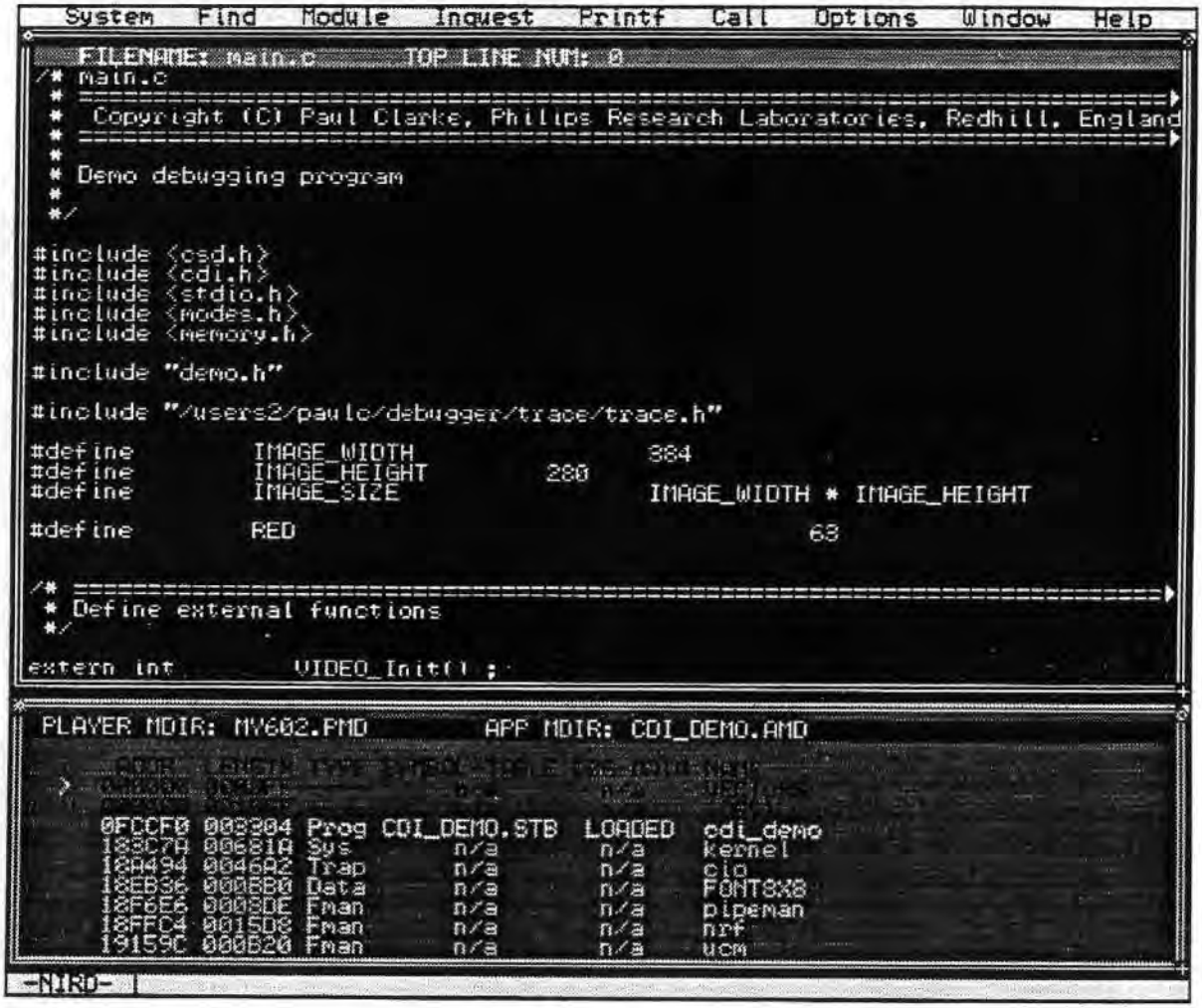

Figure 21: Workspace having loaded debug data file

The top two-thirds of the screen is occupied by a source viewing window. The title line at the top of this window shows the name of the source file that has been loaded. Below the title line is the text of the source file. If an inquest data file has been loaded then the host software will show you the line of source code that corresponds with a given application bus cycle. This can be seen in Figure 22.

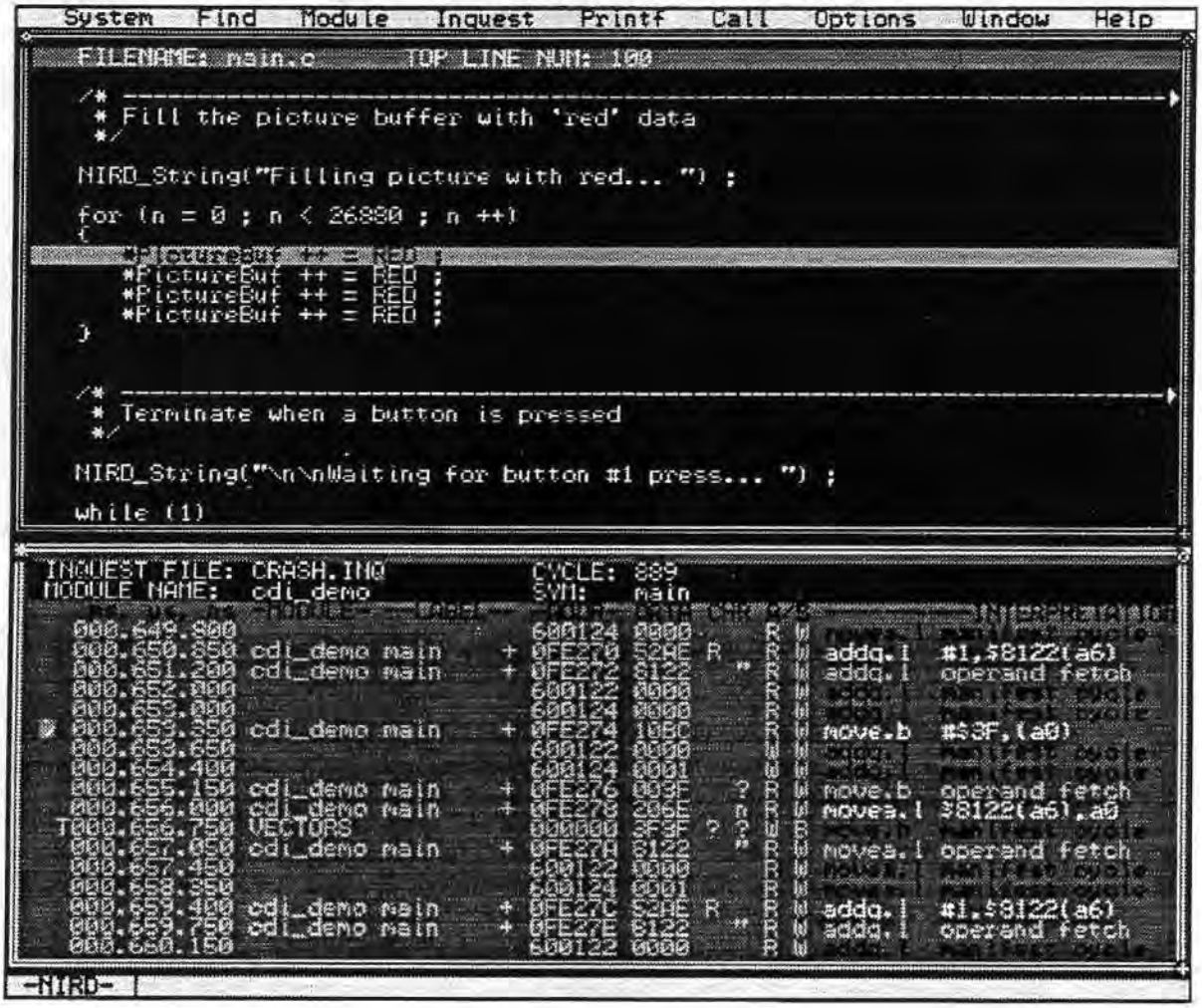

Figure 22: Inquest data indexing into source file

As you move the cursor through the inquest data the source code window changes to show the current source line highlighted by a flashing bar (as with the ***PictureBuf ++ = RED;** line in Figure 22). Here you can see that the user has moved the cursor in the inquest data window to cycle **889**. This is the disassembled instruction fetch for a byte-write of **0x3F** to the address contained in address register **a0**. This instruction results in a write to address zero (a bug!) during the trigger cycle (marked with a '**T**'). The source window shows that this instruction corresponds to the first dereference of the **PictureBuf** variable within a **for** loop.

For every line of C-source there are usually a number of assembler instructions. The **.DBG** file data only provides information about 'real' lines of source, so the host software will not highlight comments, lines with just curly brackets, etc. Furthermore, if the host software does not have the source corresponding to an instruction then no data is highlighted. As manifest cycles are outside of the application module they do not cause a source line to be highlighted.

4.4 Minimally intrusive printfs

The most commonly used debugging aid that CD-i developers use is the **printf()** call. This is used to send messages out of the serial port of a CD-i player to a terminal. There are three main problems with this.

The first problem with the use of printfs is their slow speed of execution. This is partly due to the fact that they involve a system call with its associated overhead, and also because the player stops executing the application whilst the message is being sent synchronously out of the slow serial port.

The second problem with printfs is that they cannot be used in time-critical pieces of code or from device drivers as they are so slow. For example, if a game were to use printfs to show the frame rate every second then the value shown would be slower than a version of the code without any printfs.

The third (and greatest) problem with printfs is that they must be removed from an application before it is released. This is because printfs will cause the player to freeze if used on a consumer player with no terminal attached. Unfortunately, bugs can sometimes be introduced during the 'printf removal' stage, and putting the messages back in can remove the bugs again. Furthermore, if a title appears to have a bug in a new player in the future then the testers will have no debugging output to make use of.

Minimally intrusive printfs provide a fast messaging facility with the following features:

- Calling special functions in a supplied library outputs debugging messages to the debugging cartridge, and from there to a terminal-emulation screen on the host PC.
- Messages are saved into a PC file for later analysis.
- Linking to another library maps the same function calls onto traditional printfs for multi-person projects with only one debugging cartridge.
- Very fast, so they hardly perturb the timing characteristics of code and can be used in games, device drivers, etc.
- Very small library, so they do not take up much room in your application.
- Asynchronous, which means that they operate as fast as possible even if the debugging cartridge is not connected. They can also be left in your title removing the risk of encountering new bugs at the printf-removal stage, and providing a means to debug the title whenever a cartridge is attached to a player in the future.

The debugging messages are written to a global variable within a shared data module, and this global variable is monitored by the cartridge. Any writes to this global are detected in real time and are sent to the host PC for displaying and saving in a text file. As the global is in a shared data module, different processes can write to it as it is not within a particular module.

Each process needs to initialise the NIRD printf mechanism. If the shared data module does not exist then it is automatically created (it is very small with only one byte of data), and if it already exists then it is linked. When the shared data module is first created it is declared to the cartridge. This code within the **NIRD_Init()** function is shown in Figure 23.

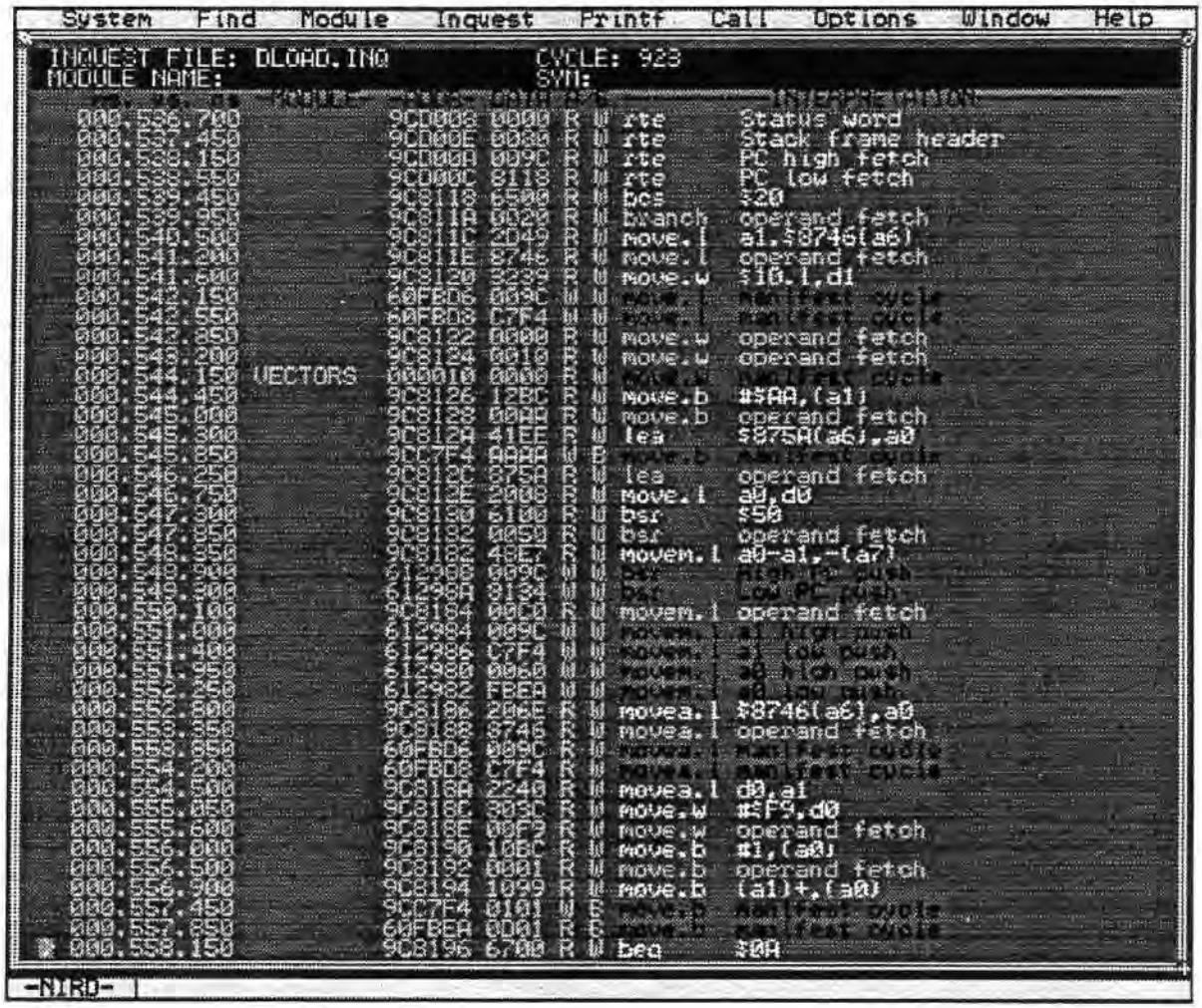

Figure 23: Inquest data showing write port being created

The **NIRD_Init()** function needs to show the cartridge that the write port is about to be declared, and it does this by reading a low memory address. To stop this triggering the cartridge in inquest debugging do not have Illegal instructions set as a trigger as it is this vectors address that is used to arm the NIRD circuitry for printf initialisation.

The inquest data in Figure 23 shows how the initialisation code declares the global variable in the shared data module to the cartridge. At the top of the screen the **F$DatMod** system call returns having created the data module. The global variable pointer **(a1)** is saved in a global variable and then declared to the cartridge. The address of the write port is defined as the first address to have **0xAA** written to it after address **0x000010** is read. This is performed by two assembly-code instructions. The **move.w $10.l,d0** reads address **0x000010** and arms the port declaration circuitry on the cartridge, and the **move.b #$AA,(a1)** instruction writes **0xAA** to the write port. At this point the cartridge hardware latches the global variable's address so that further writes to this address can be detected and communicated to the host PC. The initialisation code then displays a welcome banner using the write-port. Although messages are sent to the cartridge through the global variable, the variable should always be accessed using the provided functions and ***not*** by your application directly.

The minimally intrusive printf library makes the following functions directly available to programmers:

NIRD_Init() This declares the write-port to the cartridge and displays a welcome banner to the user using it. If the shared data module does not exist then it is created (80 bytes in size, called **nird_WritePort**) and the write port declaration process described on the previous page is performed. If the shared data module already exists then the address of the global variable stored within it is stored in a global within the current process and used for NIRD printf messages. An initialisation banner is always displayed when this function is called.

NIRD_String() This displays a string given a string pointer.
eg: **NIRD_String("Hello world\n") ;**

NIRD_Char() This displays a signed character given a value.
eg: **NIRD_Char(CurrentCharacter) ;**

NIRD_uChar() This displays an unsigned character given a value.
eg: **NIRD_uChar(Byte) ;**

NIRD_Short() This displays a signed short (16-bit) given a value.
eg: **NIRD_Short(LoopCounter) ;**

NIRD_uShort() This displays an unsigned short (16-bit) given a value.
eg: **NIRD_uShort(Word) ;**

NIRD_Int() This displays a signed integer (32-bit) given a value.
eg: **NIRD_Int(BigLoopCounter) ;**

NIRD_uInt() These display an unsigned integer (32-bit) given a value.
NIRD_Unsigned() eg: **NIRD_uInt(32BitValue) ;**
eg: **NIRD_Unsigned(32BitValue) ;**

NIRD_Long() This displays a signed long (32-bit) given a value.
eg: **NIRD_Long(MyLongVariable) ;**

NIRD_Pointer() This displays an unsigned long pointer (32-bit) given a value.
eg: **NIRD_Pointer(HotspotListRoot) ;**

Assembly coders should load the parameter into data register **d0** before the call. C-programmers should pass these functions the required parameter. These functions are declared in the header file provided, **trace.h**. Some macros are also defined there to allow the more usual mixed messages (such as "Size=15") to be generated easily.

The string followed by value macros are as follows:

```
NIRD_String_String() ;
NIRD_String_Char() ;
NIRD_String_uChar() ;
NIRD_String_Short() ;
NIRD_String_uShort() ;
NIRD_String_Int() ;
NIRD_String_uInt() ;
NIRD_String_Unsigned() ;
NIRD_String_Long() ;
NIRD_String_Pointer() ;
```

Some simple messages just need a value and a newline. These types of message are provided for by the following macros:

```
NIRD_NL() ;
NIRD_String_NL() ;
NIRD_Char_NL() ;
NIRD_uChar_NL() ;
NIRD_Short_NL() ;
NIRD_uShort_NL() ;
NIRD_Int_NL() ;
NIRD_uInt_NL() ;
NIRD_Unsigned_NL() ;
NIRD_Long_NL() ;
NIRD_Pointer_NL() ;
```

Putting these together for simple messages with a new line, there are the following macros:

NIRD_String_String_NL() ;
NIRD_String_Char_NL() ;
NIRD_String_uChar_NL() ;
NIRD_String_Short_NL() ;
NIRD_String_uShort_NL() ;
NIRD_String_Int_NL() ;
NIRD_String_uInt_NL() ;
NIRD_String_Unsigned_NL() ;
NIRD_String_Long_NL() ;
NIRD_String_Pointer_NL() ;

Also declared in this library is an easy way to trigger the cartridge in the inquest debugging mode. There is a function called **NIRD_Trigger()** which reads address **0x000008** (the bus error exception). This means that you can make a call to this function, trigger the inquest filter on a read from address **0x000008**, upload and disassemble data, and use the resulting information to show you exactly how your code is executing for verification or profiling.

◆ Note that if you have "Illegal instructions" as one of your inquest trigger conditions then the cartridge will count the write-port declaration read of address **0x000010** as a trigger condition being met. If you are debugging an application which uses minimally intrusive printfs then you can avoid this problem by setting the trigger count selector to one to ignore the write-port generation, or do not trigger on illegal instructions. Remember - always keep debugging messages short to minimise title disruption. To use minimally intrusive printfs in your code you must use the files in the supplied **printf** subdirectory. In this subdirectory you will find the following files:

trace.h This file provides function prototypes and macros for using the minimally intrusive printf library from C-code.

trace.r This is the library file which you should link with to provide the minimally intrusive printf capability.

trace_p.r This is the library file which you should link with to map your minimally intrusive printf calls onto normal printfs for serial-port debugging messages.

A minimally intrusive printf version of the famous "Hello World" program would thus be:

```
/*
============================================================
 * Example "Hello World" program using minimally intrusive printfs
 *
 * Link againsteithertrace.r to get minimally intrusive printfs
 *  or  trace_p.rto get normal serial-port messages
 */

#include <stdio.h>
#include "~/nird/printf/trace.h"

    /*=======================================================
    *This simple program prints "Hello World"
    */

main()
{
    /*-------------------------------------------------------
    *First we initialise the minimally intrusive system
    *- This declares the write port and prints the welcome banner
    */

  NIRD_Init() ;

    /*-------------------------------------------------------
    *Now we can print messages!
    */

  NIRD_String("\n\nHello world!\n\n") ;
}
```

To detect minimally intrusive printfs using the debugging cartridge select the **Printf | Start tracing** menu-item. This presents you with a dialog box where you can enter the name of the file into which you want the messages to be saved. This file always has the **.LOG** filetype. When you are happy with the log file name select **OK**. After selecting **OK** a virtual-terminal window will be displayed on the screen of the host PC. Any detected printf messages will be shown here. Figure 24 shows message detection and display in operation.

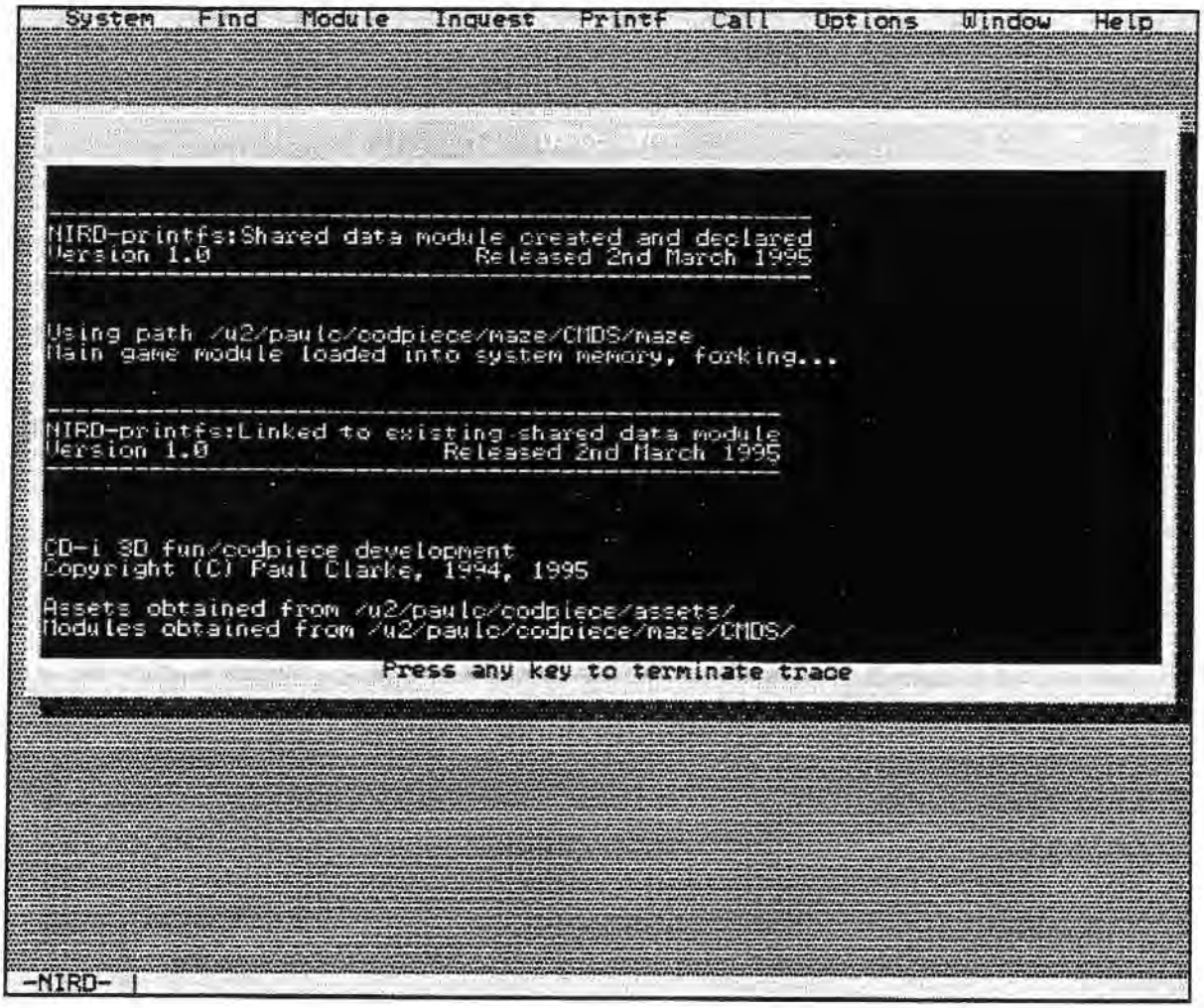

Figure 24: Detecting and displaying minimally intrusive printfs

- Note that here the first process created the shared data module and forked a second process which links to this module.

Text is displayed in the scrolling window with line wrap where appropriate. Numerical values are shown in both decimal and hexadecimal for clarity, and characters are also shown in ASCII form if possible.

To reduce the amount of data sent to the host software, and to increase the performance of the minimally intrusive printf library, the data is formatted into ASCII at the host PC and not at the player. The normal printf scheme would have to convert a number into ASCII digits and display each one in turn - the minimally intrusive system informs the PC that a number of a given format is to be displayed, and then sends the PC the actual number. This format identifier preceeds each burst of data, and thus if you write to the monitored global yourself you will confuse the system. When you press a key to terminate the detection of minimally intrusive printf messages the host software automatically loads the file for analysis. You may load an existing log file by using the **Printf | View log file** menu-item which presents you with the standard file chooser.

4.5 Tracing system calls

System calls are the interface between an application and the operating system. The use of system calls means that an application programmer does not need to concern themself with the physical layout of data on a disc or electronic components within the CD-i system, but can just make requests of the operating system such as "open file".

The CD-RTOS operating system provides a large number of system calls to the user and these are summarised in Appendix I of this manual. The debugging cartridge is able to detect system calls and returns with their parameters as they occur in real time. The host software can then show a scrolling list of system calls and returns in a window on the screen, and logs this data to a file for later analysis.

Figure 25 shows inquest data of the start of a system call.

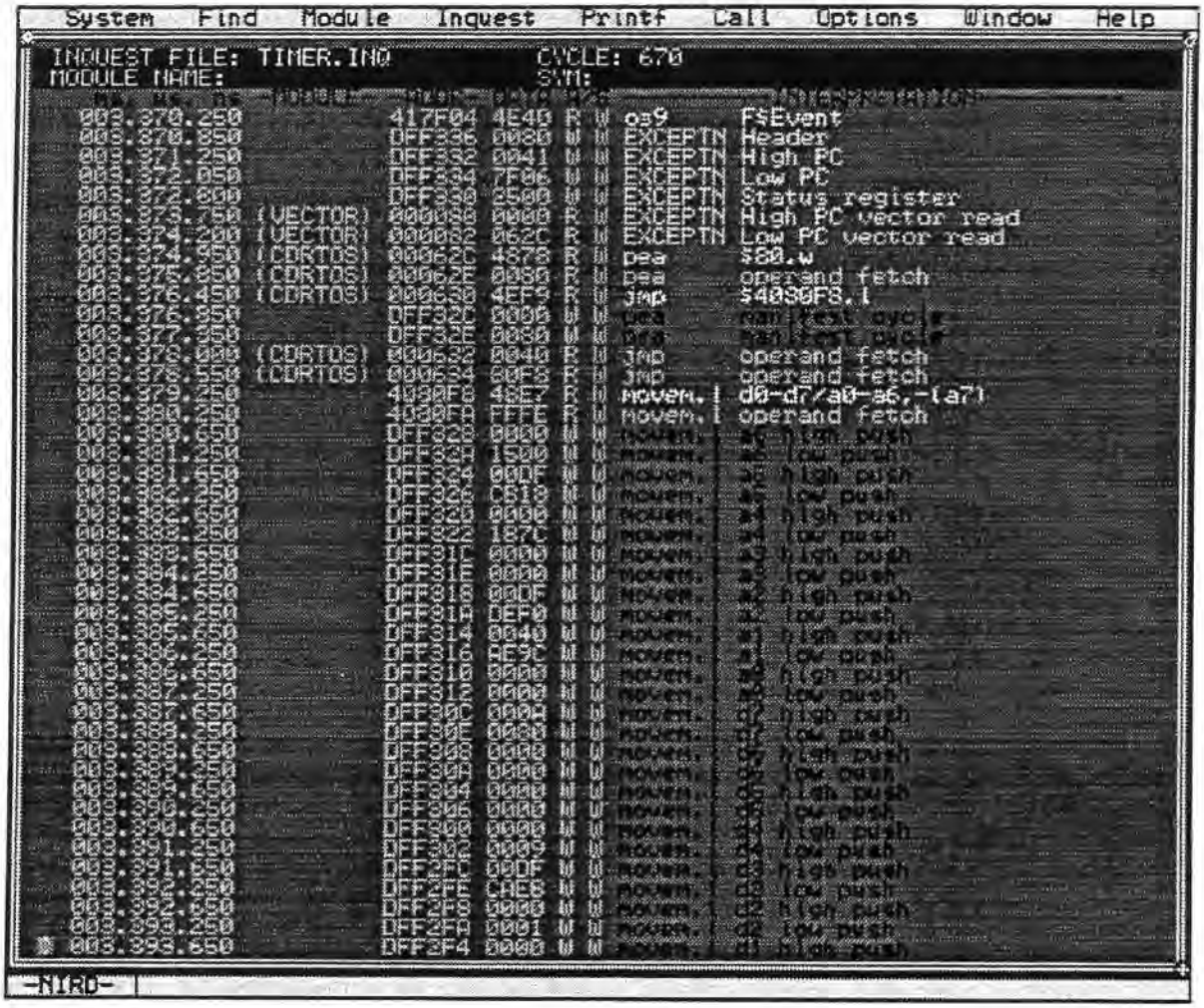

Figure 25: Inquest data showing what happens on a system call

At the top of this data a system call is made (**0x4E40** is a **TRAP #0** instruction). This causes an exception frame to be saved on the stack after which the **TRAP #0** vector is read (long word at address **0x000080**). This causes the processor to execute from the **CDRTOS** area and to vector into the **kernel** module. Here a move multiple instruction causes all the registers to be saved on the stack.

This save of registers is useful to us as the registers are used to pass parameters to the system call. The cartridge can monitor this register save burst and can thus detect the value of parameters for each system call.

After this point the kernel module jumps to the required module to manage the system call. For example, if a file is being opened then the CD File Manager module (**cdfm**) may perform the required function. The handling module then returns to the kernel module which pulls the registers off the stack and returns to the application. This is shown in Figure 26.

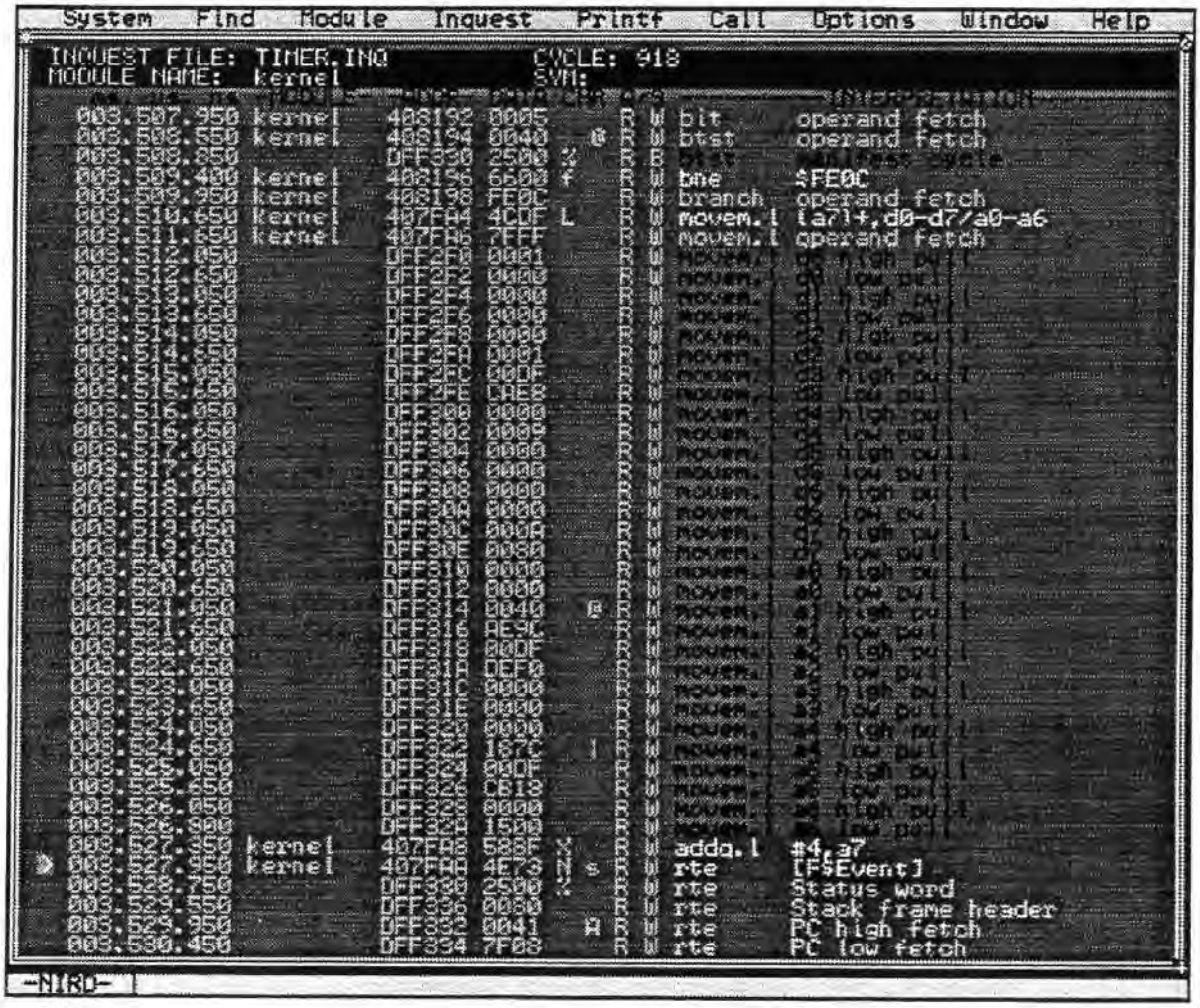

Figure 26: Inquest data showing what happens at the end of a system call

It can be seen that at the end of a system call the return parameters are read from the stack, and a return from exception (**RTE**) instruction causes the processor to read the exception frame from the stack and return to the application. By monitoring this register read burst the debugging cartridge can determine the parameters returned from a system call.

There is one problem with this approach to monitoring system calls - interrupts are not disabled during a system call, and so an interrupt can occur between the vector read and register save burst, or the register read burst and the return from exception. If this happens then that system call cannot be analysed.

Tracing of system calls can be done by selecting the **Call | Trace syscalls** menu-item. This presents you with a system call selection area. This dialog box is shown in Figure 27.

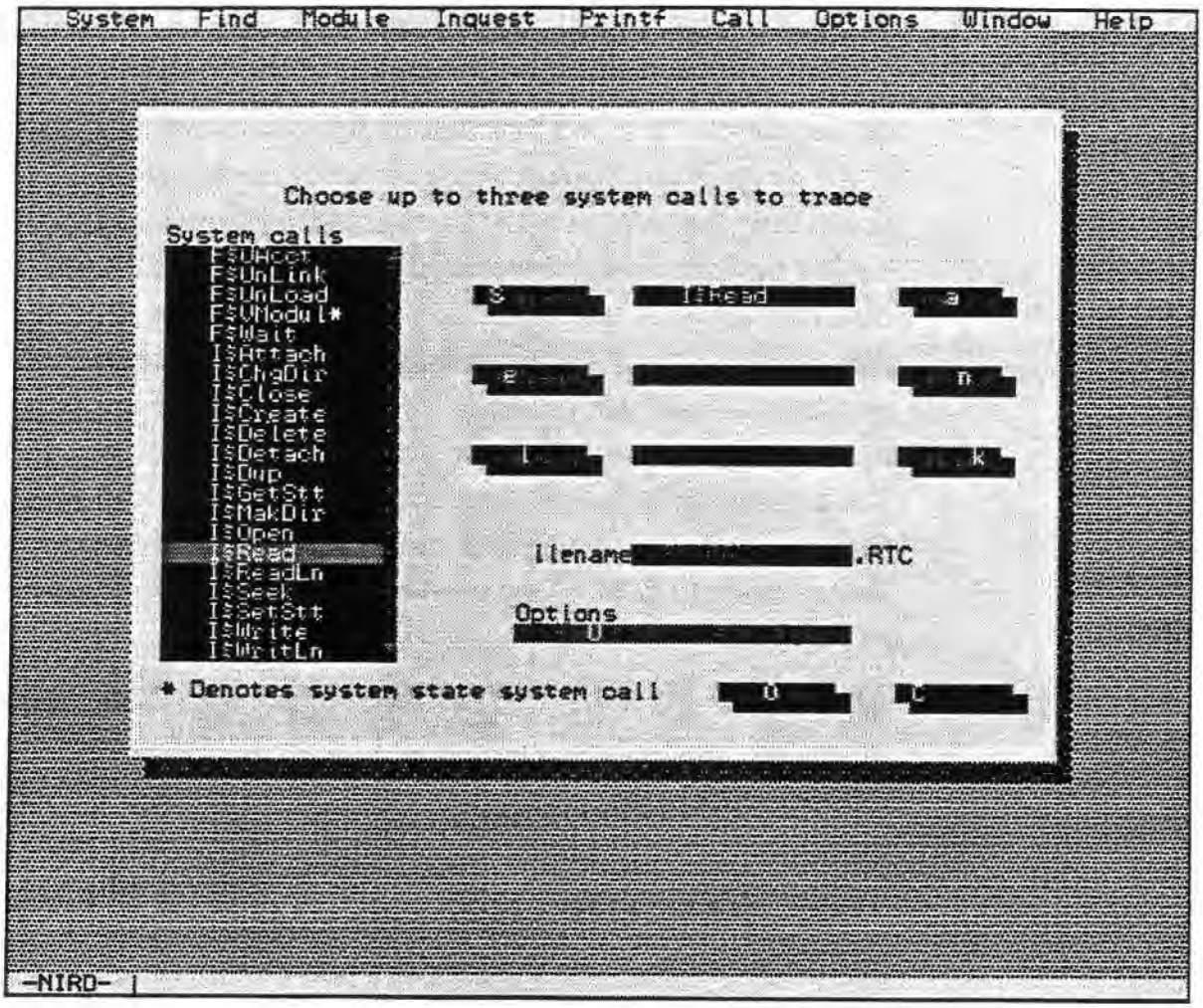

Figure 27: Choosing which system calls to trace

On the left of this dialog box is the scrolling system call selection area. The currently highlighted system call is shown with a pale background. You may use the cursor keys, home and end keys, and page up and page down keys to choose system calls. You can also select a given system call by clicking on it with the mouse cursor, or click in the scroll bar to scroll the list. When you have the required system call highlighted you must copy it to any one of the three selection areas in the centre of the screen. This can be done using the **Select** button for that bar, by pressing **RETURN** key, or by clicking in the required bar with the mouse cursor. The cartridge can only trace up to three system calls at a time. The **Blank** buttons clear a given selection area. Below the selection bars is an area where you may enter the filename of the file that the data is to be logged into, and a toggle selector to indicate if you are only interested in user-state (meaning application rather than operating system) system calls. System calls are always logged in a text **.RTC** file. Once you are happy with your system call selection select **OK**.

After selecting **OK** you will be asked to play the NIRD Utility CD-i disc. This enables the cartridge to find where the system call stub code (which causes the register save) is for your player. You will then be prompted to play your application and press a key when ready to trace the required system calls. As system calls are usually happening most of the time in background, and because the reporting of system calls usually lags behind their execution due to the delay of displaying the information, it is important to think carefully about when to start tracing system calls and how long to trace them for.

During the detection of system calls their information is displayed in a scrolling window on the host computer's screen. When you wish to stop tracing system calls press any key and the log file will be automatically loaded for you to analyse. This is shown in Figure 28.

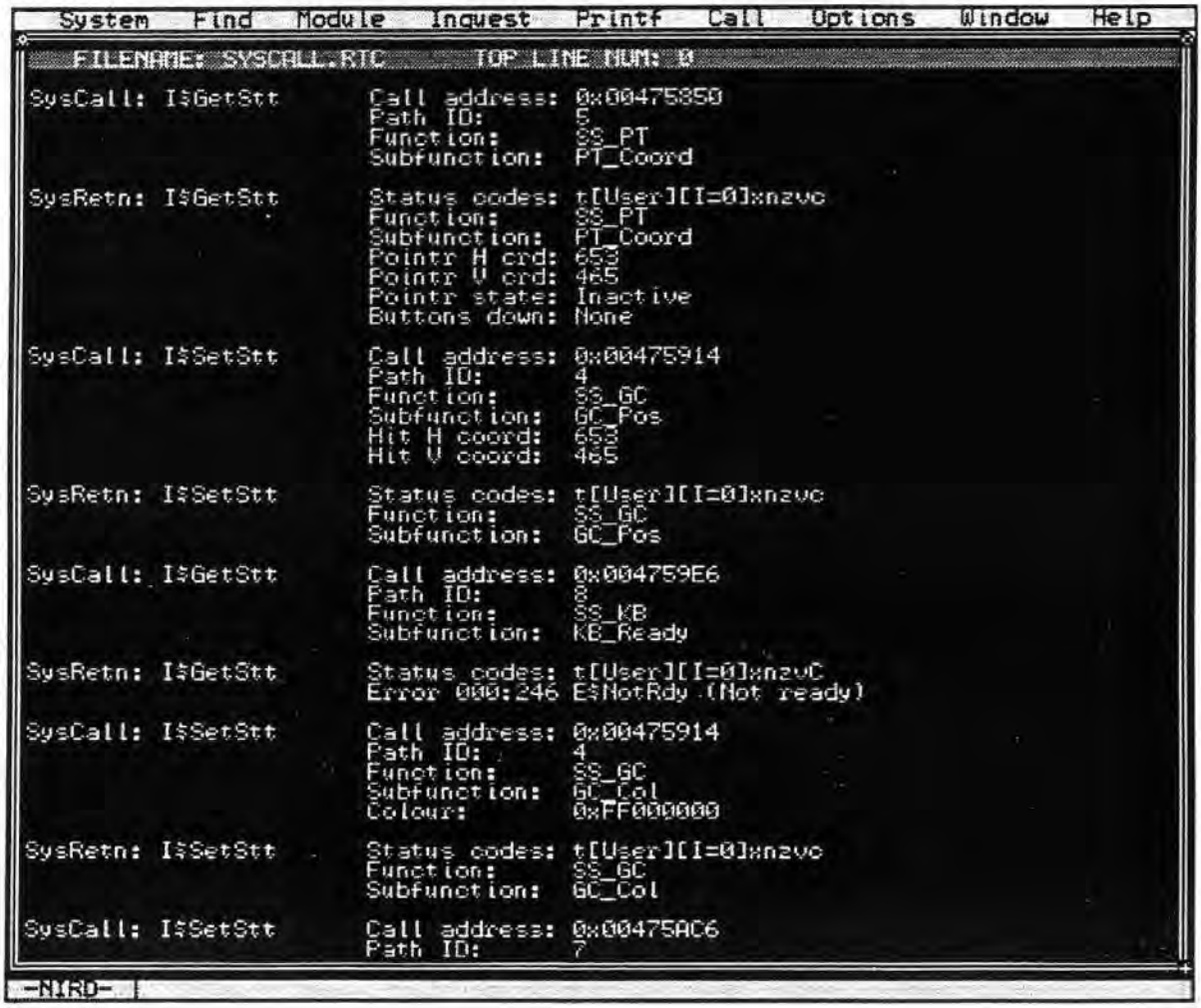

Figure 28: Viewing a log file of traced system calls

- Note that there are two situations where the system will not detect system calls reliably. The first of these is if an interrupt occurs during the system call stub code that the cartridge is monitoring. If an interrupt occurs then analysis of that system call is aborted. The second situation where system calls are not reported is if the cartridges internal buffer of system call occurrences overflows. This occurs where the rate of system calls over a period of time exceeds the rate at which the cartridge can analyse and report them to the host PC. If the buffer overflows then a large number of system calls occurrences will be lost before they can be reported.

Viewing of a system call log file may also be done manually by selecting the **Call | View log file** menu-item. The data shown in Figure 28 is typical of a system call trace. Here, the **I$GetStt** and **I$SetStt** system calls have been traced and these show the application polling the pointing device and the keyboard, and manipulating the graphics contest. Notice how the call and return parameters have been interpreted by the host software including the error returned by the **KB_Ready I$GetStt call**. Due to the speed of displaying system calls during detection the trace system calls option is mostly useful for verifying that a given system call occurs, or finding out the address where a given system call occurs so that the address can be used as a trigger in the inquest debugging mode.

4.6 Hints and tips

This section contains some hints and tips for using the NIRD, for tracking down bugs, designing for testability, and for generally writing more bug free code.

4.6.1 Avoiding null pointers

Null pointers arise for two main reasons. Firstly, the CD-i loader initialises all global variables to zero when a title loads. If the title makes use of a global pointer variable which has not been set to a valid value then it will inadvertently access the bottom area of memory which will either cause obscure crashes or can cause strange errors later in the execution of a title. Secondly, null values are frequently used to terminate lists. If the end of list is not detected then the resulting pointer can again access the low memory area. The main problem with null pointer references is that they may not crash a system, but they may crash on a different system. People then assume that it is the new system that is at fault when this is not the case.

One way to avoid most null pointer references to initialise all global pointers with an odd value, and to use an odd value as an end of list pointer. Whilst this will not prevent code from containing null pointer references, it will increase the likelihood of an erroneous pointer reference crashing the player with an address error. This will draw your attention to the bug and reduce problems with titles when new players are released.

If writing C-code, it is a good idea to define a "nothing" value that will be used throughout the code in place of the traditional null value:

```
#define NOTHING 1
```

When you declare global pointers in your code you should then initialise them to this value like this:

```
struct Palette *PaletteBuffer        = NOTHING ;
struct HotSpotNode *HotSpotList      = NOTHING ;
```

Then, when manipulating lists, use the **NOTHING** value where previously you have used the **NULL** value.

Many libraries use **NULL** as an end of list value (such as Balboa) so you have to be careful when interfacing your code to library functions that the expected end of list value is used, however, for private data structures that do not interface with library functions, try to use an odd end of list value.

Remember that as byte access to odd addresses are permitted, a **char *** pointer with an odd value is permitted and will not cause a crash. even if it has value of one.

4.6.2 Fixing cyclic lists

List data structures can become cyclic due to corruption and this can lead to player freezes. Pressing a key on the host PC keyboard to trigger the cartridge will then just show a buffer full of list traversals to the same node. It is difficult getting the cartridge to trigger at the point where list corruption takes place.

For debugging purposes the best way to detect the point of corruption is to test for cyclic loops in the list traversal code. If a loop is detected then use the **NIRD_Trigger()** function call or purposely read an address in low memory to trigger the cartridge.

The extra code to test for loop corruption can always be removed once the corrupting bug has been identified and fixed. In other words, change the list traversal code where the loop occurs from something like this:

```
struct ListNode *FindEndOfList(struct ListNode *StartNode)
{
    struct ListNode *CurrentNode = StartNode ;

    while (CurrentNode->Next != NULL)
    {
        CurrentNode = CurrentNode->Next ;
    }

    return(CurrentNode) ;
```

Change it to something which checks for cyclic pointers like this:

```
struct ListNode *FindEndOfList(struct ListNode *StartNode)
{
    struct ListNode *CurrentNode = StartNode ;

    while (CurrentNode->Next != NULL)
    {
        if (CurrentNode->Next == CurrentNode)
        {
            NIRD_Trigger() ;
        }
        CurrentNode = CurrentNode->Next ;
    }

    return(CurrentNode) ;
}
```

4.6.3 Avoiding software rot

Every time you link software with the **-g** option a new symbol table (**.STB**) file is created. These frequently become lost or out of date, and by the time a bug is found in a WORM no valid symbol file can be found. This is one example of "software rot". As symbol tables are small in size compared with all the other things that go on a WORM it is trivial to include the symbol table file as a yellow file as part of your disc image. This extra couple of lines in your master script can allow bugs to be narrowed down to specific functions even if the current version of the source code has changed.

Chapter 5. SOFTWARE GUIDE

This section describes the function of each item within each menu of the host software, and is intended to form a reference to the host software.

5.1 The 'System' menu

The **System** menu provides facilities concerned with the debugging host or host software. Each menu item is described in turn.

5.1.1 The 'About' menu item

The **System | About** menu item displays a dialog box which shows the following information:

- The name of the product.
- The version number and release date of the host software.
- The copyright message.

Select **OK** to close the dialog box.

5.1.2 The 'Print' menu item

The **System** | **Print** menu item allows you to print the contents of the inquest data window to a text file. When you select this menu item you will be presented with a dialog box where you can control the printing to disc. There are fields in the dialog box to enter:

- The name of the text (**.TXT**) file that is to be printed into.
- The cycle number where printing should start.
- The cycle number where printing should end.
- The number of lines to be printed before a new page (**Ctl+L** plus header information).

In addition to these data entry areas, there are checkboxes which control the following:

- Whether to print all data or just the decoded instruction fetches.
- Whether to end each line with **CR/LF** or just **LF** characters.

Select **OK** to proceed with printing or **Cancel** to close the dialog box.

5.1.3 The 'Statistics' menu item

The **System** | **Statistics** menu item displays a dialog box showing a number of debugging statistics. Different types of memory usage in the system are shown in tabular form, and the total amounts of memory used and free memory are shown.

The dialog box also shows the version of the cartridge operating system in use. When the dialog box is displayed the version is shown as **n/a** for "not available". The host software then asks the cartridge for its OS level and this is displayed when the data is received. Usually this happens so quickly that you never see the letters **n/a**. However, if the communications link is broken or if the CD-i player with the cartridge connected is not switched on then the OS level will be reported as **n/a**. This menu item is therefore useful to make sure that the complete debugging system is ready for work.

Select **OK** to close the dialog box.

5.1.4 The 'DOS shell' menu item

The **System** | **DOS shell** menu item returns you temporarily to DOS. The host software and all your work are still resident in memory and you must type **EXIT** to return to it. If you are using the host software and need to do something in DOS and then use the host software again, then this method is a lot quicker than exiting, restarting the software, and setting up all your windows again.

5.1.5 The 'Quit' menu item

The **System** | **Quit** menu item closes all the open files and windows and exits the host debugging software.

5.2 The 'Find' menu

The **Find** menu provides searching and jumping facilities within the inquest data window. Each menu item is described in turn.

5.2.1 The 'Goto cycle' menu item

The **Find** | **Goto cycle** menu item displays a small dialog box where you may enter the cycle number you wish to jump to. When you press return the inquest data window will show the desired cycle.

5.2.2 The 'Find cycle' menu item

The **Find** | **Find cycle** menu item displays a dialog box where you may specify a search pattern in terms of address value, data value and search direction. The software will then search the inquest data from the current cursor cycle for this search specification. This is especially useful if you find that an address has become corrupted causing a crash. Searching back for previous accesses in the inquest data to this address may reveal how the address was corrupted.

You may choose to search on one of the following search criteria:

- Data match only.
- Address match only.
- Data and addresses both match.
- Data match but address must not match.
- Address match but data must not match.

Select **OK** to perform the search or **Cancel** to simply close the dialog box.

5.2.3 The 'Find next' menu item

The **Find | Find next** menu item applies the previously set search criteria to another search from the current cursor cycle, and is useful for tracing through inquest data to see all access to a given address for example. This menu item does not produce a dialog box, it simply starts a new search.

If no previous search criteria have been set then the default values are used which is a forward search for data value **0x0000**.

5.3 The 'Module' menu

The **Module** menu provides facilities for defining what the host software knows about CD-RTOS modules during a debugging session. Facilities to determine the player's module directory and watch application modules load are provided, and it is possible to then associate symbol tables and compiler debug data files with modules to aid debugging.

5.3.1 The 'Detect mdir' menu item

The **Module | Detect mdir** menu item provides a facility to determine the CD-i player's ROMed CD-RTOS modules. These include the operating system itself, device descriptors and drivers, font modules, and so on.

Selecting this menu item produces a dialog box. This reminds you that this facility needs to use the NIRD Utility Disc, and provides a data entry area where you can supply a name for the **.PMD** file in which the player module directory data is to be stored.

If you select **OK** then the cartridge will wait for the Utility Disc to be run. Alternatively selecting **Cancel** aborts this debugging facility. The application on the NIRD Utility Disc communicates the player's module directory to the waiting cartridge. A small dialog box on the screen shows the number of modules that have been reported as the this proceeds. Once data about all the modules has been received the required PMD file is built and loaded so that you can review the information.

5.3.2 The 'Load PMD file' menu item

The **Module | Load PMD file** menu item allows you to load a previously detected PMD file for use during debugging. The player module directory of a given player remains constant unless a Digital Video cartridge is connected. Player module directories for different players should be assumed to be different.

Selecting this option produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.PMD** filetype. Highlighting a particular file and selecting **OK** will open this file. Selecting **Cancel** at any time will abort this debugging function.

5.3.3 The 'Watch loads' menu item

The **Module | Watch loads** menu item provides a facility to monitor the linking of application modules into the players memory map in real-time. To do this the cartridge determines where the module header parity evaluation loop is in the kernel module of the operating system and then monitors this code executing in real time.

Selecting this menu item produces a dialog box. This reminds you that this facility needs to use the NIRD Utility Disc, and provides a data entry area where you can supply a name for the **.AMD** file in which the application module directory data is to be stored.

If you select **OK** , the cartridge will wait for the Utility Disc to be run. Alternatively selecting **Cancel** aborts this debugging facility. The first time the Utility Disc is run, the cartridge determines the position of this "link loop" in the operating system. You will then be asked to play the Utility Disc again during which the cartridge determines which is the first module to be linked following a player reset. You are then instructed to play your application. Information about application modules is displayed in a scrolling window.

The monitoring of modules linking stops when either a key is pressed, or when the cartridge detects a player reset. The required AMD file is built and loaded so that you can review the information.

◆ Note that the cartridge cannot determine the name of modules as they are not evaluated as part of the module header parity checking. Modules are name **appmod1**, **appmod2**, and so on. After loading an AMD file you may change the names of modules by moving the cursor to the appropriate row in the window and pressing **RETURN**. This then produces a dialog box which contains a data entry area in which you may enter the module's correct name.

5.3.4 The 'Load AMD file' menu item

The **Module | Load AMD file** menu item allows you to load a previously detected AMD file for use during debugging. The application module directory can be assumed top remain the same over multiple runs of a title on the same CD-i player. However, remember that different CD-i players will load application modules in different places.

Selecting this option produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.AMD** filetype. Highlighting a particular file and selecting **OK** will open this file. Selecting **Cancel** at any time will abort this debugging function.

5.3.5 The 'Load symbols' menu item

The **Module** | **Load Symbols** menu item allows you to load a symbol table for a module shown in the module data window. Once a symbol table has been defined, access to that module in the inquest data window have label data showing where in the module was accessed.

Symbol tables are generated by the MicroWare linker by using the **-g** option. The resulting symbol table is described in the MicroWare OS-9 technical manual and has a specific format including a field giving the module name to which it relates, and a table of name/value associations for labels within the module.

Selecting this option produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.STB** filetype. Highlighting a particular file and selecting **OK** will open this file. Selecting **Cancel** at any time will abort this debugging function.

As labels are read from the file their name is shown in a progress dialog box and their data is saved in a temporary text file called **symbol.txt** in the current directory. This file is loaded into a text window for browsing upon completion of symbol table reading.

5.3.6 The 'Load dbg data' menu item

The **Module | Load dbg data** menu item allows you to load a debug data file for a module shown in the module data window. Once a debug data file has been defined the host software will show you the line of C-source associated with access to that module, provided the debug data covers that area of the module.

Debug data files are generated by the MicroWare compiler linker by compiling and linking with the **-g** option. The resulting file contains information relating the line numbers of C-source files with offsets within the CD-RTOS module.

Selecting this option produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.DBG** filetype. Highlighting a particular file and selecting **OK** will open this file. Selecting **Cancel** at any time will abort this debugging function.

While the debug data file is being loaded the host software shows the current source file being analysed in a progress dialog box. When loading of the debug data file is complete, the host software creates a source window in which it shows the source file and highlighted line of source code relating to the bus cycle in the inquest data window where the cursor is currently positioned.

Source files must reside in the current directory. If no debug data is available for the cursor bus cycle then no highlighted line of code is shown. This usually occurs in library functions for example.

5.4 The 'Inquest' menu

The **Inquest** menu provides the inquest debugging facilities. These allow you to specify a trigger condition which will be detected in real-time on the expansion bus of the player. While waiting to detect the trigger the cartridge stores information about every bus cycle in a circular buffer large enough to hold data about 16384 cycles. After the trigger has been detected the user may upload the bus cycle data to the host PC for disassembly and then analysis.

5.4.1 The 'Trigger' menu item

The **Inquest | Trigger** menu item produces a dialog box where you may specify the trigger conditions that will stop the saving of player bus cycles. This may be an access to an error exception vector to trigger on crashes and null pointer references, or it may be a specific cycle specified in terms of address, data and control bus values. You may choose to hold the trigger off for 128 cycles, and may specify a trigger count that needs to be reached before data saving stops. The screen can be saved and an alarm raised on trigger detection if wished.

If you select **OK** then the cartridge will start looking for the trigger condition whilst saving data about player bus cycles. Alternatively selecting **Cancel** aborts this debugging facility. Whilst the cartridge is waiting to detect the trigger condition a small dialog box shows the current status. When the trigger is detected a green flashing message will indicate this in the dialog box, and if previously enabled, an audible alarm will sound.

Pressing a key after the trigger has been detected the software automatically takes you to the 'Upload' menu item.

5.4.2 The 'Upload' menu item

The **Inquest | Upload** menu item allows you to control the uploading of buffered bus cycle data from the debugging cartridge. When you select this menu item a dialog box is displayed. Here you may choose how many bus cycles up to the trigger cycle you wish to upload, the method of upload (slow serial, or fast parallel if your PC has a bidirectional parallel port), and the **.DAT** file in which the raw data is to be stored.

Selecting **OK** performs the upload and automatically takes you to the 'Disassemble' menu item. Alternatively, selecting **Cancel** removes the dialog box without performing the upload.

5.4.3 The 'Disassemble' menu item

The **Inquest | Disassemble** menu item produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.DAT** filetype. Highlighting a particular file and selecting **OK** will disassemble this file. Selecting **Cancel** at any time will abort this debugging function.

The disassembly uses the raw bus cycle data in the DAT file to produce a disassembled inquest file of filetype **.INQ**. After performing the disassembly the software automatically loads the file for viewing.

5.4.4 The 'Load INQ file' menu item

The **Inquest | Load INQ file** menu item produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.INQ** filetype. Highlighting a particular file and selecting **OK** will load this file for viewing. Selecting **Cancel** at any time will abort this debugging function.

5.5 The 'Printf' menu

The Printf menu provides the minimally intrusive printf debugging functions. These are intended to replace traditional printfs with faster/smaller code that writes the debugging messages to a global variable which is monitored by the cartridge. Any data that is written to that variable is also communicated to the host PC for display in a window and saving to a log file.

5.5.1 The 'Start tracing' menu item

The **Printf** | **Start tracing** menu item displays a dialog box where you can enter the name of the log file (of type **.LOG**) into which the data is to be saved. Selecting **OK** displays a large dialog box with a printf display area, and selecting **Cancel** aborts this debugging function. When you wish to stop monitoring for minimally intrusive printf messages simply press a key. The log file is then automatically loaded for you to browse using the normal cursor movement keys.

5.5.2 The 'View log file' menu item

The **Printf** | **View log file** menu item produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.LOG** filetype. Highlighting a particular file and selecting **OK** will load this log file for viewing. Selecting **Cancel** at any time will abort this debugging function.

5.6 The 'Call' menu

The Call menu provides the system call tracing debugging functions. System calls are the interface mechanism between an application and the operating system. Tracing system calls with parameters can allow you to analyse which system calls are made, when, with what parameters, and their return results.

5.6.1 The 'Trace syscalls' menu item

The **Call | Trace syscalls** menu item displays a dialog box where you can choose up to three system calls to trace. The left had side of the dialog box contains a scrolling list of system calls from which you may select. You may use the usual cursor keys or mouse to highlight the desired system call. Clicking with the mouse in one of the horizontal selection bars or using the appropriate on-screen **Select** button selects the system call for tracing. Using an on-screen **Blank** button frees the appropriate selection bar. There is also an area in the dialog box where you may enter the name of the file into which the data is to be logged for later analysis, and there is a checkbox where you may choose to just view system calls from user state to help reduce the number of system calls reported.

Selecting **OK** proceeds with system call tracing and **Cancel** aborts the debugging function. Before the cartridge can trace system calls it needs to find the stub code that is called on every system call for your CD-i player. To do this you are asked to play the NIRD Utility Disc. You will then be presented with a dialog box that prompts you to press a key when you wish to start tracing system calls.

System calls and returns are displayed in a scrolling area of a dialog box. To stop tracing system calls press a key on the keyboard. The system call log file is then loaded for you to browse using the normal cursor movement keys.

5.6.2 The 'View log file' menu item

The **Call** | **View log file** menu item produces the standard file chooser which allows you to browse the directory structure looking for a file of the required **.RTC** filetype. Highlighting a particular file and selecting **OK** will load this log file for viewing. Selecting **Cancel** at any time will abort this debugging function.

5.7 The 'Options' menu

The **Options** menu provides the facility to change some aspects of the host system. These and other system parameters are stored in a configuration file called **NIRD.CFG** in the **BIN** directory of the NIRD software. This file is read every time the host software starts, and is written every time the host software is terminated.

5.7.1 The 'Inquest' menu item

The **Options** | **Inquest** menu item produces a small dialog box with check boxes to define which columns are visible in the inquest data viewing window. The available options are:

- Timestamps.
- Module name.
- Label name.
- ASCII codes.

When happy with the choices select **OK**.

5.7.2 The 'Misc' menu item

The **Options** | **Misc** menu item produces a small dialog box with a data entry area where you may define the number of spaces to be used in place of **TAB** characters encountered when loading files. This is useful when performing C-source debugging to ensure that the loaded code looks correct with regard to indentations, etc.

5.8 The 'Window' menu

The **Window** menu provides a number of controls for manipulating the currently selected window on the screen.

5.8.1 The 'Size' menu item

The **Window | Size** menu item allows you to control the size of the window. Once you have selected this menu item you may control the bottom right corner of the window using the cursor keys. When you are happy with the size of the window press the **RETURN** key.

5.8.2 The 'Move' menu item

The **Window | Move** menu item allows you to control the position of the window. Once you have selected this menu item you may control the position of the window using the cursor keys. When you are happy with the position of the window press the **RETURN** key.

5.8.3 The 'Zoom' menu item

The **Window | Zoom** menu item quickly changes the size of the window between the current size and the largest size available for the window. If the current size is already the largest size available for the window then the window is made to be the last non-largest size that the window has ever been.

5.8.4 The 'Next' menu item

The **Window | Next** menu item rearranges the depth order of windows on the screen. The window at the back of all others is brought to the front of the screen and the order of the others remains the same. Repeatedly using this option thus cycles the windows on the screen.

5.8.5 The 'Close' menu item

The **Window | Close** menu item closes the currently selected window.

5.9 The 'Help' menu

The **Help** menu provides on-screen help to the host debugging software. In the **BIN** subdirectory of the NIRD software are a number of **.HLP** text files. These contain similar information to that contained in this section of the user guide. The menu items within the **Help** menu load these files into a text window for browsing using the usual cursor control keys. Also, the text positioning on each line dictates its colour to make things clearer.

Chapter 6. FAULTFINDING

This section describes some of the common problems that you may experience and suggested remedies.

Serial communication problems

- The host debugging PC communicates with the debugging cartridge using a serial communications link. A cable is provided to connect the PC(serial) port of the cartridge with the 9-pin COM1 port of the PC. If the PC has a serial mouse then it must be connected to the 25-pin COM2 port.
- The NIRD communications device driver must be installed. This is done by a line such as **DEVICEHIGH=C:\NIRD\BIN\COMMS.SYS** in the **C:\CONFIG.SYS** file. Remember to replace the **C:\NIRD\BIN** element of this path with the correct text if you did not install the software in the default directory structure.
- Make sure that the NIRD communications device driver displays a start up message when you reboot the computer. With multiple menu-item configurations in the **CONFIG.SYS** and **AUTOEXEC.BAT** files it is sometimes tricky to see where to place the **DEVICE** and **PATH** lines correctly.
- The mouse driver must be instructed to use COM2. If no port is specified at the mouse driver definition line in the **C:\CONFIG.SYS** or **C:\AUTOEXEC.BAT** files then it may scan the ports, hooking the interrupts in the process, and preventing the NIRD communications device driver from operating.

Parallel communication problems

- The parallel communications link is used for the fast uploading of data from the cartridge when using the inquest debugging facility. The main problem that will be encountered here is that not all parallel ports are bidirectional. The host software will inform you if your parallel port is not bidirectional when you try to upload data using it. You may use the serial communications link instead but parallel uploading of data is much quicker. Replacement I/O cards which have IDE hard drive, floppy drive, serial ports and a bidirectional parallel port cost under $20.

Digital Video problems

- To use the NIRD with Digital Video you must connect the Digital Video cartridge and the NIRD to the test CD-i player. If you are using the older, larger, 100-pin connector Digital Video cartridges then the NIRD cartridge forms a long bus extender with a support caddy at the rear. Some player/NIRD/DV combinations cannot function with the extended bus. If DV does not work if a NIRD is attached to a player then you must try another DV cartridge. This happens relatively rarely, but unfortunately does happen.
- The adapter board which allows connection of a debugging cartridge to a player with a 120-pin DV connector does not allow you to then use an old-style 100-pin DV cartridge. The adapter board has a through 120-pin connector for connection to a new-style 120-pin DV cartridge.

Disassembly problems

- The disassembler is a complex part of the host software which converts the captured bus cycle data obtained using the inquest facility into useful disassembled data that you can analyse. It is possible, though unlikely, that a particular data set will confuse the disassembler. If this ever happens and the resulting data is corrupt, please report the incident and email the source DAT file to your NIRD support service.

Reporting bugs and getting technical support

- Support for the NIRD is provided by Philips Interactive Media Centre (PIMC) at Hasselt in Belgium.
  Either:

  - Email **support@pimc.be**
    with the word NIRD in the subject line.
  - Write to Philips Interactive Media Centre
    Maastrichterstraat 63,
    3500 Hasselt,
    Belgium.
  - Telephone +32 11 242 546
  - Fax +32 11 242 168

Chapter 7. Appendices

Appendix A. Cartridge connectors

Table I shows the pin assignment for both serial ports available on the cartridge.:

Pin	*Signal descriptiont*
1	No connection
2	Receive data (RXD)
3	Transmit data (TXD)
4	Linked with pin 6
5	Signal ground (GND)
6	Linked with pin 4
7	Ready to Send (RTS)
8	Clear to Send (CTS)
9	No connection

Table I: Pin allocations for serial ports on cartridges

Table II shows the pin assignment for the parallel port available on the cartridge.

Pin	*Signal descriptiont*
1	**STROBE**
2-9	**Databits (D0-D7)**
10	**ACK**
18-25	**Signal Ground (GND)**
Others	**Not Connected (NC)**

Table II: Pin allocations for parallel port on cartridge

Appendix B. Upgrading firmware

Upgrades to the cartridge require an EPROM device to be changed on the cartridge's printed circuit board. This is usually done by Philips Interactive Media Centre (PIMC) at Hasselt in Belgium. The rest of this section describes how to perform an upgrade if you are asked to do it yourself.

Access to the cartridge circuit board is achieved by unscrewing the six securing screws on the underside of the cartridge. The lid then slides out leaving the base behind with the Printed Circuit Board (PCB) attached.

The cartridge firmware resides on a 128 KByte EPROM (271024) with access time equal to or faster than 120ns. The NEC D271024-12 chip is the preferred device. The EPROM is situated in a socket at position U2 at the same edge of the board as the parallel port.

The two jumpers L1 and L2 near the player Digital Video connector are not used in this release and should be left in the open position unless otherwise instructed.

The two sockets U35ODD and U36EVEN may in the future accept EPROMS holding OS-9 modules to be mapped into the player's memory. This facility is not supported in the current implementation and the sockets should be left empty.

Appendix C. The DAT file structure

Inquest data is uploaded to the host PC and is stored in a .DAT data file. Table III shows the 'event frame' format of each bus cycle stored in the file for format revision 1 (the only currently available DAT file format).

Word	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
0	**Address 23..16**								**B**	**U**	**L**	**R**	**Event Code**			
1	**Address 15..1**															**D**
2	**Data 15..0**															
3	**TS 15..0**															

Table III: Event frame format

The data frame consists of the following fields:

Event Code 4-bit event code describing event frame

R Player read or write cycle identifier

L Lower Data Strobe signal from processor

U Upper Data Strobe signal from processor

D DMA cycle flag

B Bus Error signal

TS<15:0> 16-bit time-stamp

Address<23:1> Address bus contents during cycle

Data<15:0> Data bus contents during cycle

Appendix D. The INQ file structure

The DAT data file is disassembled into a .INQ inquest file ready for analysis by the programmer. Table IV shows the format of this file for format revision 1 (the only currently available INQ file format).

Field name	*Length in bytes*	*Contents*
File ID	**3**	**"INQ"**
Rev Number	**1**	**File format revision**
Cycle Count	**2**	**Number of states in file (n)**
Cycle Index	**n x 4**	**Offset for each state into file**
Cycle Entry	**16 + Comment**	**Inquest Cycle Frame (Cycle 0)**
Cycle Entry	**16 + Comment**	**Inquest Cycle Frame (Cycle 1)**
...	**16 + Comment**	**...**
Cycle Entry	**16 + Comment**	**Inquest Cycle Frame (Cycle n)**

Table IV: Inquest file format

Table V shows the format of each bus cycle event frame referenced in Table IV.

Field name	*Length (bytes)*	*Contents*
State Number	**2**	**Number of this state (0..n)**
Decoded	**6**	**Cycle 0 = 'time zero'**
Address	**2**	**Address of access**
Data	**2**	**Data for access**
Flag Byte	**1**	**Bits for read/write, word/byte**
Instruction ID	**1**	**Disassembled instruction**
Comment Length	**1**	**Length of comment field (m)**
Comment Field	**m**	**ASCII comment field**

Table V: Inquest Cycle Frame format

Appendix E. The PMD file structure

The player module directory is stored in an ASCII .PMD file. The file format is given in Table VI for format revision 1 (the only currently available PMD file format).

Line number	*Field name*	*Contents*
1	**File ID**	**"PMD"**
2	**Rev Number**	**File format revision (1)**
3	**Module Count**	**Number of modules in file (n)**
4	**Module Entry**	**Module 1 entry line**
5	**Module Entry**	**Module 2 entry line**
...	**Module Entry**	**...**
3 + n	**Module Entry**	**Module n entry line**

Table VI: Player Module Directory file format

Each Module Entry line has the following ASCII format (space delimited):

<Hex start address> <Hex length> <Decimal type> <Name>

The decimal type is defined in the Green Book as one of the following values:

0	Not used
1	Program module
2	Subroutine module
3	Multi-module
4	Data module
5	Configuration status descriptor
6-10	Reserved
11	User trap library
12	System module (CD-RTOS component)
13	File manager module
14	Physical device driver
15	Device descriptor module
16-255	User definable

Appendix F. The AMD file structure

The application module directory is stored in an ASCII .AMD file. The file format is given in Table VII for file format revision 1 (the only currently available AMD file format).

Line number	*Field name*	*Contents*
1	File ID	"AMD"
2	Rev Number	File format revision (1)
3	Module Count	Number of modules in file (n)
4	Module Entry	Module 1 entry line
5	Module Entry	Module 2 entry line
...	Module Entry	...
3 + n	Module Entry	Module n entry line

Table VII: Application Module Directory file format

Each Module Entry line has the following ASCII format (space delimited):

<Hex start address> <Hex length> <Decimal type> <Name>

The decimal type is defined in the Green Book as one of the following values:

0	Not used
1	Program module
2	Subroutine module
3	Multi-module
4	Data module
5	Configuration status descriptor
6-10	Reserved
11	User trap library
12	System module (CD-RTOS component)
13	File manager module
14	Physical device driver
15	Device descriptor module
16-255	User definable

Appendix G. Configuration file format

The following entries may appear in the configuration file called **NIRD.CFG** in the **BIN** subdirectory of the host software. If a given entry is not found then the default value is used. Each text label is followed by a value, with one label on each line of the file.

TextTabWidth Number of spaces to be used for a tab character when loading a text file (numerical value). Example value is **3**.

PrintPageLength Number of lines that should be printed to disc before a form feed and a new banner is displayed (numerical value). Example value is **60**.

JustPrintAsm Flag to show if software should just print to disc the fully decoded (white) assembly code bus cycles. Example value is **ON**.

NoPrintCR Flag to show if software should just print to disc a LF character at the end of a line rather than a LF/CR combination. Example value is **OFF**.

InqViewTimestamp Flag to show if user wishes to display the timestamp column in the inquest data window. Example value is **ON**.

InqViewASCII Flag to show if user wishes to display the ASCII column in the inquest data window. Example value is **OFF**.

Appendix H. Other file structures

Symbol table files The format of the **.STB** files produced by the linker with the **-g** option is given in the Assembler, Linker, Debugger section of the MiroWare OS-9 Compiler Technical Manual.

Debug data files A description of the format of the **.DBG** files produced by the compiler and linker if both are used with the **-g** option is beyond the scope of this document.

Other files All other files used or produced by the host software (namely **.HLP**, **.LOG**, **.TRC**, and **.C**) are pure text files.

Appendix I. Instruction set summary

This appendix summarises the instruction set of the 68070 microprocessor to help you understand the information in the inquest data window.

Status register

The status bits are allocated as shown below:.

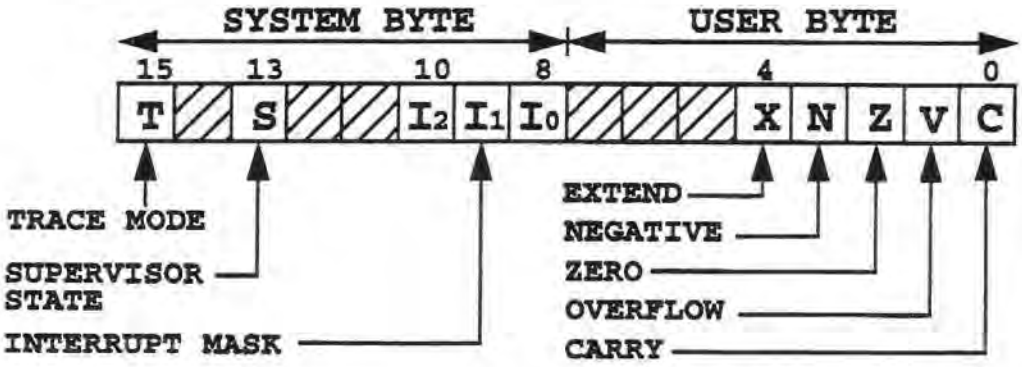

Table VIII: Status register

ABCD Add decimal with extend. Add the source operand to the destination operand along with the extend bit, and store the result in the destination location. The addition is performed using binary coded decimal arithmetic.

ADD Add binary. Add the source operand to the destination operand, and store the result in the destination location.

ADDA Add address. Add the source operand to the destination address register, and store the result in the address register.

ADDI Add immediate. Add the immediate data to the destination operand, and store the result in the destination operand.

ADDQ Add quick. Add the immediate data to the operand at the destination location.

ADDX Add extended. Add the source operand to the destination operand along with the extend bit and store the result in the destination location.

AND Logical AND. AND the source operand to the destination operand and store the result in the destination location.

ANDI AND immediate. AND the immediate data to the destination operand and store the result in the destination location.

ASL, ASR Arithmetic shift. Arithmetically shift the bits of the operand in the direction specified.

Bcc Branch conditional. If the specified condition is met then program execution continues at location (PC) + displacement. Displacement is a twos complement integer which counts the relative distance in bytes. 'cc' may be one of the following conditions:

CC	carry clear	**CS**	carry set	**EQ**	equal
GE	greater/equal	**GT**	greater	**HI**	high
LE	less/equal	**LS**	low/same	**LT**	less than
MI	minus	**NE**	not equal	**PL**	plus
VC	overflow clear	**VS**	overflow set		

BCHG Test a bit and change. A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code. After the test, the state of the bit is changed in the destination.

BCLR Test a bit and clear. A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code. After the test, the specified bit is cleared in the destination.

BRA Branch always. Program execution continues at location (PC) + displacement. Displacement is a twos complement integer which counts the relative distance in bytes.

BSET Test a bit and set. A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code. After the test, the specified bit is set in the destination.

BSR Branch to subroutine. The long word address of the instruction immediately following the BSR instruction is pushed onto the system stack. Program execution continues at location (PC) + displacement. Displacement is a twos complement integer which counts the relative distance in bytes.

BTST Test a bit. A bit in the destination operand is tested and the state of the specified bit is reflected in the Z condition code.

CHK Check register against bounds. The content of the lower order word in the data register specified in the instruction is examined and compared to the upper bound. The upper bound is a twos complement integer. If the register value is less than zero or greater than the upper bound contained in the operand word then the processor initiates exception processing. The vector number is generated to reference the CHK instruction exception vector.

CLR Clear an operand. The destination is cleared to all zeros.

CMP Compare. Subtract the source operand from the specified data register and set the condition codes according to the result; the specified data register is not changed.

CMPA Compare address. Subtract the source operand from the destination register and set the condition codes according to the result; the address register is not changed.

CMPI Compare immediate. Subtract the immediate data from the destination operand and set the condition codes according to the result; the destination location is not changed.

CMPM Compare memory. Subtract the source operand from the destination operand and set the condition codes according to the result; the destination location is not changed.

DBcc Test condition, decrement and Branch. This instruction is a looping primitive with three parameters: a condition, a counter (data register), and a displacement. The instruction first tests the condition to determine if the termination condition for the loop has been met, and if so, no operation is performed. If the termination condition is not true then the low order 16 bits of the counter data register are decremented by one. If the result is -1 then the counter is exhausted and execution continues with the next instruction. If the result is not equal to -1 then execution continues at the location indicated by the location (PC) + displacement. Displacement is a twos complement integer which counts the relative distance in bytes. 'cc' may be one of the following conditions:

CC	carry clear	**CS**	carry set	**EQ**	equal
F	never true	**GE**	greater/equal	**GT**	greater
HI	high	**LE**	less/equal	**LS**	low/same
LT	less than	**MI**	minus	**NE**	not equal
PL	plus	**T**	always true	**VC**	overflow clear
VS	overflow set				

DIVS Signed divide. Divide the destination operand by the source and store the result in the destination. The operation is performed using signed arithmetic.

DIVU Unsigned divide. Divide the destination operand by the source and store the result in the destination. The operation is performed using unsigned arithmetic.

EOR Logical exclusive OR. Exclusive OR the source operand to the destination operand and store the result in the destination location.

EORI Exclusive OR immediate. Exclusive OR the immediate data to the destination operand and store the result in the destination location.

EXG Exchange registers. Exchange the contents of the two registers.

EXT Sign extend. Extend the sign bit of a data register from a byte to a word, or from a word to a long operand, depending on the size selected.

ILLEGAL Illegal instruction. This bit pattern causes an illegal instruction exception.

JMP Jump. Program execution continues at the effective address specified by the instruction.

JSR Jump to subroutine. The long word address of the instruction following the JSR instruction is pushed onto the system stack. Program execution then continues at the address specified in the instruction.

LEA Load effective address. The effective address is loaded into the specified address register.

LINK Link and allocate. The current content of the specified address register is pushed onto the stack. After the push, the address register is loaded from the updated stack pointer. Finally, the 16-bit sign-extended displacement operand is added to the stack pointer.

LSL, LSR Logical shift. Shift the bits of the operand in the direction specified.

MOVE Move data from source to destination. Move the content of the source to the destination location. The data is examined as it is moved and the condition codes are set accordingly.

MOVEA Move address. Move the content of the source into the destination address register. Condition codes are not affected.

MOVEM Move multiple. Selected registers are transferred to or from consecutive memory locations starting at the location specified by the effective address. A register is transferred if the bit corresponding to that register is set in the mask field.

MOVEP Move peripheral data. data is transferred between a data register and alternate bytes of memory, starting at the location specified and incrementing by two.

MOVEQ Move quick. Move immediate data to a data register. The data is contained in an 8-bit field within the instruction word. The data is sign-extended to a long operand and all 32 bits are transferred to the data register.

MULS Signed multiply. Multiply two signed operands yielding a signed result.

MULU Unsigned multiply. Multiply two unsigned operands yielding an unsigned result.

NBCD Negate decimal with extend. The operand addressed as the destination and the extend bit are subtracted from zero. The result is saved in the destination location. The operation is performed using binary coded decimal arithmetic.

NEG Negate. The operand addressed as the location is subtracted from zero. The result is saved in the destination location.

NEGX Negate with extend. The operand addressed as the destination and the extend bit are subtracted from zero. The result is saved in the destination location.

NOP No operation. No operation occurs.

NOT Logical complement. The ones complement of the destination operand is taken and the result is stored in the destination location.

OR Inclusive logical OR. Inclusive OR the source operand to the destination operand and store the result in the destination location.

ORI Inclusive OR immediate. Inclusive OR the immediate data to the destination operand and store the result in the destination location.

PEA Push effective address. The effective address is computed and pushed onto the stack.

RESET Reset external devices. The reset line is asserted for 124 clock periods causing all external devices to be rest. The processor state, other than the program counter, is unaffected and execution continues with the next instruction.

ROL, ROR Rotate without extend. Rotate the bits of the operand in the direction specified.

ROXL, ROXR Rotate with extend. Rotate the bits of the destination operand in the direction specified. The extend bit is included in the rotation.

RTE Return from exception. The status register and program counter are pulled from the system stack. The vector number is also pulled from the stack and the format is examined to determine the amount of information to be restored.

RTR Return and restore condition codes. The condition codes and program counter are pulled from the stack. The supervisor portion of the status register is unaffected.

RTS Return from subroutine. The program counter is pulled from the stack.

SBCD Subtract decimal with extend. Subtract the source operand from the destination operand along with the extend bit and store the result in the destination location. The subtraction is performed using binary coded decimal arithmetic.

Scc Set according to condition. The specified condition code is tested; if it is true then the byte specified by the effective address is set to TRUE (all ones), otherwise that byte is set to FALSE (all zeros). 'cc' may be one of the following conditions:

CC	carry clear	**CS**	carry set	**EQ**	equal
F	never true	**GE**	greater/equal	**GT**	greater
HI	high	**LE**	less/equal	**LS**	low/same
LT	less than	**MI**	minus	**NE**	not equal
PL	plus	**T**	always true	**VC**	overflow clear
VS	overflow set				

STOP Load status register and stop. The immediate operand is moved into the entire status register; the program is advanced to point to the next instruction and the processor stops fetching and executing instructions. If an interrupt request is asserted with a higher priority than the level set by the status register then an interrupt exception occurs otherwise the interrupt request has no effect.

SUB Subtract binary. Subtract the source operand from the destination operand and store the result in the destination.

SUBA Subtract address. Subtract the source operand from the destination address register and store the result in the address register.

SUBI Subtract immediate. Subtract the immediate data from the destination operand and store the result in the destination location.

SUBQ Subtract quick. Subtract the immediate data to the operand at the destination location.

SUBX Subtract with extend. Subtract the source operand from the destination operand along with the extend bit and store the result in the destination location.

SWAP Swap register halves. Exchange the 16-bit halves of a data register.

TAS Test and set. Test and set the byte operand addressed by the effective address field. The current value of the operand is tested and N and Z are set accordingly. The high order bit of the operand is set.

TRAP Trap. The processor initiates exception processing. The vector number is generated to reference the TRAP instruction exception vector specified by the low order four bits of the instruction.

TRAPV Trap on overflow. If the overflow condition is set then the processor initiates exception processing. The vector number is generated to the TRAPV exception vector. If the overflow condition is reset, then no operation is performed and execution continues with the next instruction in the sequence.

TST Test an operand. Compare an operand with zero. No results are saved but the condition codes are set according to the results of the test.

UNLK Unlink. The stack pointer is loaded from the specified address register. The address register is then loaded with the long word pulled from the top of the stack.

Appendix J. System call summary

This appendix summarises the system calls used in the CD-i player. A full description of their function and parameters passed can be found in the Green Book.

F$Alarm Manage alarms. This system call manages alarms which are asynchronous software alarm timers. The function of the timer is to send a signal to the calling process when the specified time period has elapsed. A process may have multiple alarm requests pending.

F$AllBit Set bits in an allocation bit map. This system call sets bits in the allocation bit map that was found by an F$SchBit system call.

F$AllPD Allocate process/path descriptor. This system state system call is used to dynamically allocate fixed length blocks of system memory. It allocates and initialises (to zeros) a block of storage and returns its address.

F$AllPrc Allocate process descriptor. this system state system call allocates and initialises a process descriptor. The address of the descriptor is kept in the process descriptor table.

F$AProc Insert process in active process queue. This system state system call inserts a process into the active process queue so that it may be scheduled for execution.

F$CCtl Cache control. This system call is used to perform operations on the system instruction and/or data caches. Standard Philips CD-i players do not have caches.

F$Chain Load and execute a new primary module. This system call is used when it is necessary to execute an entirely new program. It is functionally similar to a Fork command followed by an Exit.

F$CmpNam Compare two names. This system call compares a target pattern to a source pattern to determine if they are equal. Case insensitive.

F$CpyMem Copy external memory. This system call copies external memory into the user's buffer for inspection.

F$CRC Generate CRC. This system call generates or checks the CRC (cyclic redundancy check) values of sections of memory.

F$DatMod Create data module. This system call creates a data module and clears its data portion. Several processes can communicate with each other using such a module.

F$DelBit Deallocate in a bit map. This system call clears bits in an allocation map that were previously allocated and are now free for general use.

F$DelPrc Deallocate process descriptor service request. This system state system call deallocates a process descriptor previously allocated by F$AllPD.

F$DExec Execute debugged program. This system call controls the execution of a suspended child process that has been created by the F$DFork system call.

F$DExit Exit debugged program. This system call terminates a suspended child process that was created with the F$DFork system call.

F$DFork Fork process under control of debugger. This system call works similar to F$Fork except that it is provided for a debugger utility to create a process whose execution can be closely controlled.

F$Exit Terminate the calling process. This system calls provides a means by which a process can terminate itself. Its data memory is deallocated and its primary module is unlinked. All open paths are automatically closed.

F$FindPD Find process/path descriptor. This system state system call converts a process or path number into the absolute address of its descriptor data structure.

F$Fork Create a new process. This system call creates a new process which becomes the child of the caller. It sets up the process' memory and standard I/O paths.

F$GBlkMp Get free memory block map. This system call copies the address and size of the system's free RAM blocks into the user's buffer for inspection.

F$GModDr Get module directory. This system call copies the system's module directory into the user's buffer for inspection.

F$GPrDBT Get process descriptor block table copy. This system call copies the process descriptor block table into the caller's buffer for inspection.

F$GPrDsc Get process descriptor copy. This system call copies a process descriptor into the caller's buffer for inspection.

F$Gregor Get gregorian date. This system call converts julian dates to Gregorian dates.

F$ID Get process/user ID. This system call returns the caller's process ID number, group and user ID, and current process priority.

F$IOQu Enter I/O queue. This system state system call links the calling process into the I/O queue of the specified process and performs an untimed sleep.

F$Icpt Set up a signal intercept trap. This system call tells OS-9 to install a signal intercept routine.

F$IRQ Add or remove device from IRQ table. This system state system call installs or removes an IRQ service routine in the system polling table.

F$Julian Get Julian date. This system call converts gregorian dates to julian dates.

F$Link Link to a memory module. This system call causes OS-9 to search the module directory for a module having the name, language, and type as given in the parameters. The module's link count is incremented.

F$Load Load module(s) from a file. This system call opens the specified file and reads one or more modules from the file into memory until an error or end of file is reached.

F$Mem Resize data memory area. This system call is used to contract or expand the process' data memory area.

F$Move Move data. This system state system call is a fast block move routine capable of copying data bytes from one address space to another.

F$NProc Start next process. This system state system call takes the next process out of the Active Process Queue and initiates its execution.

F$Panic System catastrophic occurrence. This system state system call is made by the kernel when it detects a disastrous system condition.

F$PErr Print error message. This is the system's error printing facility which writes an error message to the standard error path.

F$PrsNam Parse a path name. This system call parses a string for a valid pathlist element, returning its size.

F$RetPD Return process/path descriptor. This system state system call deallocates a process or path descriptor.

F$RTE Return from interrupt exception. This system call may be sued to exit from a signal processing routine.

F$SchBit Search bit map for a free area. This system call searches the specified allocation bit map for a free block of the desired length.

F$Send Send a signal to another process. this system call sends a signal to a specified process.

F$SetCRC Generate a valid CRC in module. This system call updates the header parity of a module in memory.

F$SetSys Set/examine CD-RTOS system global variable. This system call is used to change or examine a system global variable.

F$SigMask Masks/unmasks signals during critical code. This system call is used to enable or disable signals from reaching the calling process.

F$Sleep Put calling process to sleep. This system call deactivates the calling process until the number of request ticks have elapsed.

F$SPrior Set process priority. This system call changes the process priority to the new given value.

F$SRqCMem System request for coloured memory. This system call allocates a block of specific type of memory.

F$SRqMem System memory request. This system call allocates memory from the top of available RAM.

F$SRtMem Return system memory. this system call deallocates memory after it is no longer required.

F$SSpd Suspend process. This system call suspends the process with the given process ID.

F$SSvc Service request table initialisation. This system state system call is used to add or replace function requests in OS=0's user and privileged system service request tables.

F$STime Set system date and time. This system call is used to set the current date/time and start the system real-time clock to produce time slice interrupts.

F$STrap Set error trap handler. This system call enters "process local" Error trap routines into the process descriptor dispatch table.

F$SUser Set user ID number. This system call alters the current user ID to the specified ID.

F$SysDbg Call system debugger. this system call invokes the system level debugger, if one exists, to allow system state routines such as device drivers to be debugged.

F$Time Get system date and time. This system call returns the current date and time in the required format.

F$TLink Install user trap handling module. This system call attempts to link or load the named module and installs a pointer to it in the user's process descriptor for subsequent use in trap calls.

F$Trans Translate memory address. This system call is used when the external bus address must be passed to hardware devices.

F$UAcct User accounting. This is a user-defined system call which may be installed by an OS9P2 module. It is called in at the beginning and end of a process.

F$UnLink Unlink a module by address. This system call tells OS-9 that the module is no longer needed by the calling process. The module's link count is decremented. When the link count equals zero the module is removed from the module directory and its memory is deallocated.

F$UnLoad Unlink module by name. This system call locates the module in the module directory, decrements its link count, and removes it from the directory if the count reaches zero.

F$VModul Verify module. This system state system call checks the module header parity and CRC bytes of an OS-9 module.

F$Wait Wait for child process to terminate. This system call causes the calling process to deactivate until a child process terminates by executing an F$Exit system call, or otherwise is terminated.

I$Attach Attach a new device to the system call. This system call causes an I/O device to become "known" to the system. It is used to attach a new device to the system, or to verify that it is already attached.

I$ChgDir Change working directory. This system call changes a process' working directory to another directory file specified by the pathlist.

I$Close Close a path to a file/directory. This system call terminates the I/O path specified by the path.

I$Create Create a path to a new file. This system call is used to create a new file.

I$Delete Delete a file. This system call deletes the file specified by the pathlist.

I$Detach Remove a device from the system. This system call removes a device from the system device table if it is not in use by another process.

I$Dup Duplicate path. This system call returns a synonymous path number for the given file or device.

I$GetStt Get file/device status. This system call is used for obtaining status information from devices within the CD-i system. For example, it is used to get the pointer position.

I$MakDir Make a new directory. This system is the only way that a new directory file can be created.

I$Open Open a path to a file or device. This system call opens a path to an existing file or device as specified by the pathlist.

I$Read Read data from a file or device. This system call reads a specified number of bytes from the path number given.

I$ReadLn Read a text line with editing. This system call is similar to I$Read except that it reads data from the input file or device until an end-of-line character is encountered.

I$Seek Reposition the logical file pointer. This system call repositions the path's file pointer.

I$SetStt Set file/device status. This system call is used for setting the status information of devices within the CD-i position. For example, it is used to set the pointer position.

I$Write Write data to a file or device. This system call outputs bytes to a file or device associated with the path number specified.

I$WriteLn Write a line of text with editing. This system call is similar to I$Write except it writes data until a carriage return character is encountered, or the maximum number of bytes specified is reached.

Appendix K. Source for utility disc

```
*  cdi_mdir.a68
*  ============================================================
*               Copyright (C) Philips Electronics, 1994, 1995
*  ============================================================
*        Project:          Hardware Debugger Utility Disc
*
*        Module:           Player Module Directory Grabber
*
*        Author:           Paul Clarke
*
*        Created:          24/03/94
*        Last Modified:    07/04/94
*
*        Description:      This code communicates the player's module
*                          directory to the cartridge and then crashes
*                          with an address error to provide a
*                          debugging example.
*  ============================================================
*  Define assembler options for assembly operation

                NAM       Hardware Debugger Utility Disc
                TTL       Player Module Directory Grabber
*  ============================================================
*  Use OS-9 definitions file

                USE       /dd/defs/oskdefs.d
*  ============================================================
*  Declare equates

Type_Lang       EQU       (Prgrm<<8)+Objct
Attr_Revs       EQU       (ReEnt<<8)+0
Edition         EQU       1
*  ============================================================
*  * Define program section

                PSECT     MDIR_CODE,Type_Lang,Attr_Revs,Edition,0,Start
*  ============================================================
```

```
*  Program entry point
*
*               - Claim memory
*               - Define writeport for communication with cartridge
*               - Get module dir and communicate it to cartridge
*               - Crash the player for the example debugging exercise
*               ---------------------------------------------------
*               Request memory block for use as an mdir buffer

Start:          move.l      #$4000,d0           d0 = 16KB required
                os9         F$SRqMem            Allocate memory system call
                bcs         End                 End if carry bit set
                move.l      d0,Size(a6)         Save buffer size
                move.l      a2,Buffer(a6)       Save pointer to block
*               ---------------------------------------------------
*               Declare communications port (write-only)

MakePort:       lea         WritePort(a6),a1    a1 ->WritePort
                movea.l     #0,a2               a2 ->Illegal instruction
                move.w      $10(a2),d0          Arm port id circuitry
                move.b      #$AA,(a1)           Declare write port
*               ---------------------------------------------------
*               Get the module directory into the buffer

GetMdir:        move.l      #$4000,d1           Buffer size is 16KB
                movea.l     Buffer(a6),a0       a0 ->Start of memory buffer
                os9         F$GModDr            Mdir system call
                bcs         End                 Check for error
                lsr.l       #4,d1               d1 = Entries in mdir table
                movea.l     Buffer(a6),a0       a0 ->Start of mdir buffer
*               ===================================================
```

```
*               Communicate current entry to cartridge

DoEntry:        move.l      (a0),a3         a3->Module header
                cmpa.l      #0,a3           Check for end of mdir
                beq         End             Quit if end reached
                cmpi.w      #$4AFC,(a3)     Check sync word in header
                bne         End             Quit if error in mdir
                move.b      #0,(a1)         Send data sync byte
*               ----------------------------------------------
*               Send module start address to cartridge

SendAddr:       move.b      (a0),(a1)       High address byte
                move.b      1(a0),(a1)      Middle address byte
                move.b      2(a0),(a1)      Lowest address byte
*               ----------------------------------------------
*               Send module length

SendLen:        move.b      4(a3),(a1)      Highest length byte
                move.b      5(a3),(a1)      High length byte
                move.b      6(a3),(a1)      Low length byte
                move.b      7(a3),(a1)      Lowest length byte
*               ----------------------------------------------
*               Send module type

SendType:       move.b      18(a3),(a1)     Type byte
*               ----------------------------------------------
*               Send module name

SendName:       move.l      12(a3),d0       d0 = Name string offset
                adda.l      d0,a3           a3 ->Start of name offset

SendName2:      move.b      (a3)+,d0        d0 = Name data byte
                beq         NextEntry
                move.b      d0,(a1)         Save name byte
                bra         SendName2
*               ----------------------------------------------
```

```
*               Consider next entry

NextEntry:      adda.w      #16,a0          a1 ->Next header pointer
                dbf         d1,DoEntry
*               ----------------------------------------------
*               Crash the player using an address error

End:            move.b      #$FF,(a1)       Signal end of data
                move.b      #$FF,(a1)       Make sure signal occurs
                move.w      1(a1),d0
                bra         End
*  ============================================================
*  Define variable section

                VSECT

WritePort:      dc.b        00
                ALIGN

Buffer:         dc.l        00
Size:           dc.l        00

                ENDS
*  ============================================================
*  End of cdi_mdir code module

*               ENDSECT  MDIR_CODE

*               END
```

Chapter 8. ALPHABETICAL INDEX

A

Address errors 1-3, 3-2, 4-8
Application file names 3-22
Application module directory 3-18
  Detecting 3-18, 4-33
  Naming modules 3-18, 4-36, 5-8
Autoexec.bat file 2-3, 6-1

B

Bugs
  Causing freeze 1-4
  Causing reset 1-3, 3-2, 4-5
  Obscure bugs 1-4, 4-65, 4-66
Bus cycles
  Decoded instructions 3-15, 4-20
  Disassembler error cycles 3-16, 4-20, 6-3
  DMA cycles 3-7, 3-15, 4-9, 4-20
  Exception cycles 3-7, 3-16, 4-20
  Instruction fetches 3-7, 4-20
  Manifest cycles 3-7, 3-15, 4-20, 4-24
  Operand fetches 3-7, 3-15, 4-20
  Processor-internal manifests 4-25
  Synchronisation cycles 3-15, 4-19
Bus errors 1-3, 3-2, 4-8

C

Cartridge connectors 7-1
Compiler 3-22, 4-42, 5-10, 7-12
Config.sys file 2-2, 6-1
Configuration file 5-16, 7-11

InqViewASCII keyword.....7-11
InqViewTimestamp keyword.....7-11
JustPrintAsm keyword.....7-11
NoPrintCR keyword.....7-11
PrintPageLength keyword.....7-11
TextTabWidth keyword.....7-11
C-Source viewing.....3-22, 4-42

D

Design for testability.....1-5

E

Error exceptions.....3-6, 4-8, 4-9

F

Faultfinding.....6-1
Digital Video problems.....6-2
Disassembly problems.....6-3
Parallel comms problems.....6-2
Reporting bugs.....6-3
Technical support.....6-3
Serial comms problems.....6-1
File types
AMD files.....3-18, 4-17, 4-33, 5-7, 5-8, 7-9
CFG files.....5-16, 7-11
DAT files.....3-11, 4-13, 5-12, 7-4
DBG files.....3-13, 3-22, 4-42, 5-10, 7-12
HLP files.....5-19, 7-12
INQ files.....3-11, 3-13, 3-14, 4-16, 5-12, 7-5
LOG files.....4-55, 5-13, 7-12
PMD files.....3-17, 4-17, 4-27, 5-6, 5-7, 7-7
RTC files.....4-62, 5-15
STB files.....3-13, 3-19, 3-21, 4-17, 5-9, 7-12

TRC files 7-12

G

Getting started 3-1

H

Hardware requirements 1-7
120-pin adapter 1-7, 1-9, 2-4
60X players 1-7
CD-i players 1-7
Host PC 1-8
Hints and tips 4-65
Avoiding null pointers 4-65
Avoiding software rot 4-68
Fixing cyclic lists 4-66
Host software
Distribution disc 1-9
Installation 2-1
Overview 4-1

I

Ident OS-9 utility 4-37
Illegal instructions 1-3, 3-2, 3-6, 4-9, 4-52
Inquest facility 1-3, 1-4, 3-4, 4-5
Disassembling data 3-11, 4-14
NIRD_Trigger() 4-52, 4-66, 4-67
Triggering 3-4, 4-7
Uploading data 3-9, 4-12
Viewing data 3-13, 4-16
Installation 2-1
120-pin adapter 2-4
Comms problems 2-3, 2-6, 6-1
Device driver 2-2

Hardware 2-4
Mouse problems 2-3, 6-1
PC to cartridge connection 2-5
Software 2-1
Testing 2-6
Instruction set summary 7-13

L

Linker 3-19, 3-22, 4-38, 4-42, 5-9, 5-10, 7-12

M

Menus
Call 4-3, 5-14
Call | Trace syscalls 4-61, 5-14
Call | View log file 4-64, 5-15
Find 4-3, 5-4
Find | Find cycle 4-21, 5-4
Find | Find next 4-22, 5-5
Find | Goto cycle 4-21, 5-4
Help 5-19
Inquest 4-3, 5-11
Inquest | Disassemble 3-11, 4-16, 5-12
Inquest | Load INQ file 3-14, 4-16, 5-12
Inquest | Trigger 3-5, 4-7, 5-11
Inquest | Upload 3-10, 4-12, 5-12
Module 4-3, 5-6
Module | Detect mdir 3-17, 4-28, 5-6
Module | Load AMD file 5-8
Module | Load dbg data 3-22, 4-42, 5-10
Module | Load PMD file 4-32, 5-7
Module | Load symbols 3-19, 4-38, 5-9
Module | Watch loads 3-18, 4-34, 5-7
Options 4-3, 5-16
Options | Inquest 3-15, 3-19, 4-17, 5-16

Options | Misc 5-16
Printf 4-3, 5-13
Printf | Start tracing 4-55, 5-13
Printf | View log file 4-56, 5-13
System 2-6, 4-3, 5-1
System | About 5-1
System | DOS shell 5-3
System | Print 5-2
System | Quit 5-3
System | Statistics 2-6, 3-4, 5-3
Window 4-3, 4-4, 4-32, 5-17
Window | Close 5-18
Window | Move 5-17
Window | Next 5-17
Window | Size 5-17
Window | Zoom 5-17
Minimally-intrusive printfs 1-5, 4-46
Benefits 4-47
Declaration mechanism 4-49
Demo program 4-54
Detecting messages 4-55
Header file trace.h 1-10, 4-51, 4-53
Initialisation 4-48
Library file trace.r 1-10, 4-53
Macros 4-51
NIRD_Char() 4-50
NIRD_Init() 4-48, 4-49, 4-50, 4-54
NIRD_Int() 4-50
NIRD_Long() 4-50
NIRD_Pointer() 4-50
NIRD_Short() 4-50
NIRD_String() 4-50, 4-54
NIRD_uChar() 4-50

NIRD_uInt() 4-50
NIRD_Unsigned() 4-50
NIRD_uShort() 4-50
Normal library trace_p.r 1-10, 4-53
Shared data module 4-48
Viewing log files 4-56
Module types 4-31

N

Null pointer references 1-4, 4-24, 4-65

O

On-line help 5-19

P

Player module directory 3-17
Detecting 3-17, 4-27

S

Software requirements 1-8
Symbol tables 3-19, 4-38
System call summary 7-22
System call tracing 4-57
Problems 4-60, 4-64
Viewing log files 4-64

T

Technical support 6-3
Timestamps 4-16

U

Upgrading firmware 7-3
Utility disc 3-3, 3-17, 3-18, 4-29, 4-34, 4-62, 7-30