

snoopy

Gerson Kurz

COLLABORATORS

	<i>TITLE :</i> snoopy		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Gerson Kurz	September 19, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	snoopy	1
1.1	Snoopy Version 2.0 - Help Guide	1
1.2	Introduction	2
1.3	USAGE	3
1.4	The SCRIPT startup argument	3
1.5	The TASKINFO/S startup argument	4
1.6	The OUTPUT/K startup argument	4
1.7	The PRI/N startup argument	4
1.8	The SHOW/K/M startup argument	5
1.9	The STICKY/S startup argument	5
1.10	The MATCH/S startup argument	6
1.11	The QUIT/K/N startup argument	6
1.12	The INCDIR/K startup argument	6
1.13	The NOSEGTRACKER/S startup argument	7
1.14	The scriptfile format (Syntax Background)	7
1.15	Using escape characters in scriptfiles	8
1.16	Using comments in scriptfiles	9
1.17	The register list format	9
1.18	STRUCTURE macros for includefiles	11
1.19	The BASE keyword	12
1.20	The WATCH keyword	13
1.21	The DEVICE keyword	14
1.22	The DEVCMD keyword (BEGINIO,ABORTIO)	16
1.23	The RESOURCE keyword	17
1.24	The ALIAS keyword	18
1.25	The DEFINE keyword	20
1.26	The BITDEF keyword	21
1.27	The INCLUDE keyword	22
1.28	The INCDIR keyword	23
1.29	The HIDE keyword	24

1.30	The SHOW keyword	25
1.31	The PRI keyword	26
1.32	The ECHO keyword	27
1.33	The OUTPUT keyword	28
1.34	The Sticky keyword	28
1.35	The MATCH keyword	29
1.36	The TASKINFO keyword	30
1.37	The SEGTRACKER keyword	30
1.38	The ASSUME keyword	31
1.39	The SKIPSIMILAR keyword	32
1.40	The DELAY keyword	33
1.41	Debugging your own application code - concept & ideas	34
1.42	SnoopyDebug() Templates	37
1.43	Boolean output	39
1.44	Segtracker support	39
1.45	Running multiple instances of Snoopy	40
1.46	Advanced concepts	42
1.47	AUTHOR	43
1.48	DISTRIBUTION	44
1.49	HISTORY	45
1.50	The offset/ directory	46
1.51	The scripts/ directory	46
1.52	The support/ directory	47
1.53	MAKEOFFSETS	47
1.54	BUILDWATCH	48
1.55	TASKLIST,DEVLIST,RESLIST	49
1.56	BREAK	50

Chapter 1

snoopy

1.1 Snoopy Version 2.0 - Help Guide

SNOOPY 2.0 --- Monitors any Amiga library, device or resource calls

This is I-WANT-TO-BE-FREE-WARE

(C) Copyright Gerson Kurz, Sometime early January 94

Freely Distributable!

Dedicated to the global house & techno scene

plus all wonderful human beings!

General information:

[Introduction Usage](#)

Startup arguments:

[SCRIPT TASKINFO/S](#)

[OUTPUT/K PRI/N](#)

[SHOW/K/M STICKY/S](#)

[MATCH/S QUIT/K/N](#)

[INCDIR/K NOSEGTRACKER/S](#)

Handling Scriptfiles:

[Script file format Escape characters](#)

[Comments Registers](#)

[STRUCTURE macros](#)

Scriptfile keywords:

[Base Watch](#)

[Device Devcmd](#)

[Resource Alias](#)

[Define Bitdef](#)

[Include Incdir](#)

[Hide Show](#)

[Pri Echo](#)

[Output Sticky](#)

[Match TaskInfo](#)

[SegTracker Assume](#)

[SkipSimilar Delay](#)

Debugging your own application code:

[Concept & Ideas Debug Template](#)

Additional information:

[Boolean output Segtracker Support](#)

[Multiple Instances Advanced Concepts](#)

[Author Distribution](#)

[History](#)

Support tools:

[Offsets/ directory Support/ directory](#)

[Scripts/ directory Makeoffsets](#)

[Buildwatch Tasklist](#)

[Break](#)

1.2 Introduction

This tool enables you to monitor library, device or resource function calls - of any [library](#) , [device](#) or [resource](#) you wish. The idea of course came from SnoopDos by Eddy Carroll, but Snoopy is different in approach and purpose. Snoopy has no specific patches for specific functions - it is an all-purpose tool to monitor **ANY** library, **ANY** device and **ANY** resource call in **ANY** system library, device or resource!

Snoopy can be very easily [started](#) from the CLI by just typing its name. It is driven by an input file (called a [scriptfile](#)) which describes in detail which functions you want to monitor, and how this monitoring is to be done.

This input file is a plain ASCII text that contains a simple but very readable syntax, and you can edit it with your favourite text editor.

Snoopy comes with some [default scripts](#) . One emulates SnoopDos behaviour, an other watches interesting intuition calls and a third one is used to track down some very important exec.library functions. These scripts will do the job for the causal users that need some kind of "advanced concepts" SnoopDos thing. Programmers however should read the [script syntax](#) , Snoopy will give you much debugging power (oh well ;-).

1.3 USAGE

Snoopy can be started from the CLI and from the Workbench. If you start it from the CLI (or a shell), the command line usage is :

```
Snoopy SCRIPT,TASKINFO/S,OUTPUT/K,PRI/N,SHOW/K/M,STICKY/S,MATCH/S,  
QUIT/K/N,INCDIR/K,NOSEGTRACKER/S
```

Note that you can start Snoopy without any arguments, see the defaults in the option descriptions for what will happen in this case. Also note that starting with Version 1.4 Snoopy detaches itself from the CLI, so you don't need to type "RUN Snoopy". Possible arguments are :

SCRIPT TASKINFO/S

OUTPUT/K PRI/N

SHOW/K/M STICKY/S

MATCH/S QUIT/K/N

INCDIR/K NOSEGTRACKER/S

Once Snoopy has started you will see the calls in either the output window or whatever output handle you have specified. You quit Snoopy by sending CTRL-C to it (if you have redirected your data with the **OUTPUT** option, use **support/BREAK** to do this, or use the **quit** option). Also, you can press CTRL-D to disable and CTRL-E to enable output, which means that you can pause Snoopy for some time. You can use CTRL-F to toggle the **TaskInfo** feature on or off.

If you start Snoopy from the Workbench you can add the following tooltypes

SCRIPT TASKINFO/S

OUTPUT/K PRI/N

STICKY/S MATCH/S

NOSEGTRACKER/S

Note that Snoopy does NOT take icons into consideration if started from the CLI.

1.4 The SCRIPT startup argument

DESCRIPTION

You can specify a Snoopy **script file** by giving its filename here. If this argument is not given, Snoopy will try to open "s:snoopy.script" - which hopefully exists. Snoopy cannot work if no scriptfile is given, so you either have to keep "s:snoopy.script" or provide your own scriptfile each time you start Snoopy. If Snoopy doesn't find your scriptfile it will refuse to work because in this case there is nothing to work.

EXAMPLE

```
l> snoopy test.script
```

will try to load "test.script" in the current directory

SEE ALSO

[script file syntax](#)

1.5 The TASKINFO/S startup argument

DESCRIPTION

If you activate this switch on the command line, Snoopy will print out the address and the name of each task that called a patched function before the patch message itself. This helps you to determine who called which function in which particular order - a very usefull feature. TASKINFO is disabled by default.

EXAMPLE

```
l> snoopy taskinfo=on
```

will enable taskinfo from startup

SEE ALSO

[The TASKINFO keyword](#)

1.6 The OUTPUT/K startup argument

DESCRIPTION

You can give a filename to redirect Snoopys output to. This might come in handy if you know that there'll be lots of calls or you want to examine the output later (or whatever). Just give the name of the file you want Snoopy to print its messages to. If this argument is not given, Snoopy will open its standard output window, which in turn will be "con:0/0/640/150/Snoopy".

EXAMPLE

```
l> snoopy output=prt:
```

will send all Snoopy messages directly to your printer

SEE ALSO

[The OUTPUT keyword](#)

1.7 The PRI/N startup argument

DESCRIPTION

You may need to change the priority Snoopy is running at (some reasons for this are described in the [PRI KEYWORD](#) section). You can give the pri you

want to use at this option, but it should probably be in the range -3 to +3 (I cannot guarantee for any Snoopy function if running outside this priority range).

EXAMPLE

```
1> snoopy pri=3
```

will start snoopy with a priority of +3

SEE ALSO

[The PRI keyword](#)

1.8 The SHOW/K/M startup argument

DESCRIPTION

You can tell Snoopy to watch only a series of tasks (found by their name). This comes in handy if e.g. you want to track down calls made by a specific task (most likely the one you're debugging). Note that this option automatically disables snooping of any other task (i.e. any not-to-be-SHOWn-task or [hidden](#)).

EXAMPLE

```
1> snoopy show=myapp
```

will show messages only for calls made by "myapp"

SEE ALSO

[The SHOW keyword](#) , [The HIDE keyword](#)

1.9 The STICKY/S startup argument

DESCRIPTION

If you don't like Snoopy to detach itself, use this option. This was especially designed to enable arguments like [OUTPUT=*](#) (which redirects Snoopy output to the current CLI). You can also use it to be able to break redirected Snoopys with CTRL+C such as in "snoopy sticky output=ram:snoopy.output". Workbench ignores this command as it doesn't make any sense; if you start any program from Workbench, it will always be detached.

EXAMPLE

```
1> snoopy sticky
```

will start Snoopy but without detaching itself [so you can press CTRL+C both in the output window and in the CLI window where you started Snoopy]

SEE ALSO

[The OUTPUT argument](#) , [The STICKY keyword](#)

1.10 The MATCH/S startup argument

DESCRIPTION

If this option is given, Snoopy will use case sensitive name matching instead of the default, case insensitive algorithm. This means, if MATCH is OFF, "disko" equals both "DISKO" and "disko LOVER", if MATCH is ON, it equals NEITHER OF BOTH.

EXAMPLE

```
1> snoopy match=on
```

will enable the strict string comparison mechanism

SEE ALSO

[The MATCH keyword](#)

1.11 The QUIT/K/N startup argument

DESCRIPTION

You can remove multiple instances one-by-one of Snoopy : you just tell Snoopy the instance number you want to remove. If you give a number of 0 all active Snoopys will be removed (or at least, Snoopy tries it). This is a bit tricky, so you probably should see more on this in the [multiple instances](#) section.

EXAMPLE

```
1> snoopy quit=3
```

will quit the Snoopy instance #3 if it exists

```
1> snoopy quit=0
```

quit all active Snoopys

SEE ALSO

[Multiple Instances](#)

1.12 The INCDIR/K startup argument

DESCRIPTION

You can set the directory for your includefiles in the startup arguments using this option. All you have to do is to give the directory path here; the rest works the same as [the INCDIR keyword](#)

EXAMPLE

```
1> snoopy incdir=dh2:snoopy/offsets
```

will force Snoopy to look for all includes in "dh2:snoopy/offsets"

SEE ALSO

[The INCDIR keyword](#) , [The INCLUDE keyword](#)

1.13 The NOSEGTRACKER/S startup argument

DESCRIPTION

If you want to disable SegTracker output globally, use this keyword on the command line. Note that if you use a SEGTRACKER keyword in the scriptfile, it takes priority over the command line argument; which means that if you turn SegTracker OFF with this option and then ON with a statement in the scriptfile, SegTracker is ON. This is not my fault, the problem is that I cannot find out if a ReadArgs() boolean switch has been given on the command line or not.

EXAMPLE

```
l> snoopy nosestracker
```

will disable segtracker output for the default scriptfile

SEE ALSO

[The SEGTRACKER keyword](#) , [SegTracker support](#)

1.14 The scriptfile format (Syntax Background)

DESCRIPTION

The script file is the most important thing about Snoopy - it gives you all the nice flexibility you need. A scriptfile is a plain ASCII text which basically contains any of the following possible elements or "keywords"

[Base Watch](#)

[Device Devcmd](#)

[Resource Alias](#)

[Define Bitdef](#)

[Include Incdir](#)

[Hide Show](#)

[Pri Echo](#)

[Output Sticky](#)

[Match TaskInfo](#)

[SegTracker Assume](#)

[SkipSimilar Delay](#)

Other Scriptfile syntax elements are

[Comments](#)

[Registers](#)

[Escape characters](#)

[STRUCTURE macros](#)

In general, each syntax element has the form "<keyword>=<arguments>".

Keywords have to start in the first column of a line and therefor multiple keywords in one single line are not supported. The following pages of this document describe what the statements above can be used for

EXAMPLE

A SnoopDos emulation script could look something like this

```
include=dos
...
base=dos,dos.library
...
watch=dos,Open,D1L/D2L/RBD0L,Open( "%s", $%lx ) = %s
watch=dos,Close,D1L,Close( $%lx )
...
```

1.15 Using escape characters in scriptfiles

DESCRIPTION

Snoopy understands several escape characters which can be used to get characters that Snoopy would normally use for its structures. All escape characters start with a backslash '\'. They are :

\; - results in a single ';' which is probably the only way of including this character in your output

\n - add a NEWLINE. You must use this if you want to have output thats more than one line long, because Snoopy is line oriented and doesn't support keywords that use more than one line.

\e - results in ESC (\$27). You can use this to easily add ANSI sequences (for colors, font style and so on) to your scriptfile output. Refer to your AmigaDos manuals to find out about what options are possible

\ - will give you a single backslash

\\$ - place a single \$ character. Because \$ characters are used in alias references this escape character is probably used only if you want to have a "\$(" or "\$<" sequence in your output

\t - add a tab; used if you need to keep output arguments in a row

EXAMPLE

```
echo=\; \ $ \nNEWLINE
```

will display a requester containing

```
;\ $
```

NEWLINE

Note that "\t" and "\e" are not understood by the output requester, but by Printf(), so you could use something like the following to get underlined success messages and colored strings:

```
watch=dos,-30,I7L/D1L/D2L/RBD0L,a0=%lx : Open( \e[34m"%s"\e[0m , %lx ) = \e[4m%s\e[0m
```

1.16 Using comments in scriptfiles

DESCRIPTION

Comments can be used to make scriptfiles more readable. If you want to include a whole line as a comment, you should place an asterix '*' in the first column. This line then is ignored by the snoopy input parser. If you want to comment a line that also contains other data you can use the semicolon ';'. Everything following (and including) this char will then be ignored.

EXAMPLES

* This is a full line comment

base=dos,dos.library ;this is the more general comment type
;which should be preferred to *

1.17 The register list format

SYNOPSIS

```
reglist=<reg 0>/<reg 1>/.../<reg n>  
<reg ?>=<[RIB]((D|A)<0..7>)[L|W|b]>
```

FUNCTION

The reglist describes how the registers are to be examined before and after the call to a Snoopy **patch**. As you know, the concept is this : the <template> is a C-style format string like those you would use to call `exec/RawDoFmt()` or `dos/VPrintf()`. The order of the arguments to this <template> is specified by the <regs>-list, so that for each <template> format code the corresponding <register> format data is used. The reglist consists of up to `MAX_ARGUMENTS=64` <reg ?> specifications which can have the elements mentioned below. I would very much like to introduce a new register scriptfile syntax because the current syntax is not that readable. Perhaps one of the next Snoopy updates will have something more funky...

INPUTS

[R] - this register is a return value rather than data before the function call

[B] - indicates that you want to use boolean style output

[L] - says that the register contains 32-bit data (LONG)

[W] - says that the register contains 16-bit data (WORD) Note: if neither L nor W is given, Snoopy uses the size given with the assume keyword; or, if no assumption has been defined yet, WORD size

[b] - says that the register contains 8-bit data (BYTE). Note: Snoopy

extends byte data to WORD data (ext.w) because RawDoFmt() doesn't support byte-sized data; so use "%x" for WORD and BYTE, "%lx" for LONG data. Sorry that I didn't use 'B' for this keyword but it was already used up for BOOLEAN STYLE output and I didn't want you to have to rewrite your scripts.

[Dx] - a data register

[Ax] - an address register

[Ix] - an indirect address register (plus optional offset)

EXAMPLES

D0 - data register d0, default-size data,

pre-call (=function argument)

RD0 - data register d0, default-size data,

post-call (=function returncode)

LD0 - data register d0, long-size data,

pre-call

RLD0 - data register d0, long-size data,

post-call

RBLD0 - data register d0, long-size data,

post-call, boolean output

A0 - address register a0, default-size data,

pre-call (=function argument)

RA0 - address register a0, default-size data,

post-call (=function returncode)

LA0 - address register a0, long-size data,

pre-call

RLA0 - address register a0, long-size data,

post-call

I0 - address register (a0), default-size data,

pre-call (=function argument)

RWI0 - address register (a0), word-size data,

post-call (=function returncode)

LI0 - address register (a0), long-size data,

pre-call

RLI0 - address register (a0), long-size data,

post-call

I7 - pc address that called this function =(sp)

NOTE

You ***MUST*** use the proper format specifications; which means use "%ld" for LONG data, "%d" for WORD data. Use the following supported % options:

%[flags][width.limit][length]type

flags - only one allowed. '-' specifies left justification.

width - field width. If the first character is a '0', the field will be padded with leading 0's.

. - must follow the field width, if specified

limit - maximum number of characters to output from a string.
(only valid for %s).

length - size of input data defaults to WORD for types d, x, and c, 'l' changes this to long (32-bit).

type - supported types are:

b - BSTR, data is 32-bit BPTR to byte count followed by a byte string, or NULL terminated byte string.
A NULL BPTR is treated as an empty string.

(Added in V36 exec)

d - decimal

u - unsigned decimal (Added in V37 exec)

x - hexadecimal

s - string, a 32-bit pointer to a NULL terminated byte string. In V36, a NULL pointer is treated as an empty string

c - character

%% will give you a single % character

SEE ALSO

The **WATCH** keyword , The **DEVCMD** keyword , exec/RawDoFmt

1.18 STRUCTURE macros for includefiles

DESCRIPTION

Because Snoopy now has many uses for defines, I thought it would be a good idea to add the STRUCTURE macros known to assembler programmers from exec/types.i to the scriptfile language. This way you can much easier **define** structure elements and don't have to manually calculate the offsets.

The following types are supported:

description: STRUCTURE

32-bit data: APTR,CPTR,FPTR,LONG,ULONG,FLOAT

16-bit data: BOOL,RPTR,WORD,UWORD,SHORT,USHORT

8-bit data: BYTE,UBYTE

other elements: LABEL,STRUCT,ALIGNWORD,ALIGNLONG

This way it is now directly possible to convert assembler includes to

Snoopy includes with a minimum set of changes. Remember that these macros

to the same as the DEFINE Link DefineKeyword} keyword, only you don't have to give the numbers yourself

EXAMPLES

;----- Required portion of IO request:

STRUCTURE=IO,\$<MN_SIZE>

APTR=IO_DEVICE ; device node pointer

APTR=IO_UNIT ; unit (driver private)

UWORD=IO_COMMAND ; device command

UBYTE=IO_FLAGS ; special flags

BYTE=IO_ERROR ; error or warning code

LABEL=IO_SIZE

this example declares a structure originally defined in exec/io.i

SEE ALSO

[The DEFINE keyword](#) , [The BITDEF keyword](#)

1.19 The BASE keyword

NAME

base - open a library for snooping

SYNOPSIS

base=<alias>,<library>[,<version>]

FUNCTION

You have to give Snoopy a list of all libraries you want to monitor so that Snoopy can manage the function monitoring effectively. This is done by using the BASE keyword with the above syntax.

INPUTS

<alias> - an abbreviation for the library you want to monitor. Typically this will be the part before the '.library'.

<library> - the complete name of the library you want to watch (which is directly transferred to OpenLibray()). You can even give a path (such as "dh1:foo/bar.library").

<version> - is an optional parameter which specifies the version number required. If <version> is 0 or not defined no version checking is done.

This parameter should be used if you want to monitor functions specific to a certain library version

EXAMPLES

base=dos,dos.library

will open any version of the dos.library to be accessed via the name "dos"

Base=_DOSBase,dos.library,37

will open any version of the dos.library", named "_DOSBase"

BASE=mylib,dh2:source/libtest/libs/test.library

will open "dh2:source/libtest/libs/test.library" as "mylib"

LIMITATIONS

you cannot open the same library twice with different version requests

(this is not my fault, the OS just doesn't seem to support it)

SEE ALSO

exec/OpenLibrary, [The WATCH keyword](#)

1.20 The WATCH keyword

NAME

watch - snoop into a library or resource function

SYNOPSIS

watch=<base>,<offset>,<regs>,<template>

FUNCTION

This is the most important keyword, since it specifies which function you want to watch, and how this should be done.

INPUTS

<base> - a base alias defined by a base keyword. Note that forward referencing is not possible, so you need to specify a base before you can use it in a watch

<offset> - the (signed) offset of the library function [decimal by default, use prefix \$ for hex numbers], which should probably always be a negative number. However, starting with Snoopy 1.6 you can **define** function offsets once and for all in **include** files which makes for much more readable specifications. If you include "dos", you can use something like "dos,Open,..." instead of "dos,-30,..." and so on.

<regs>,<template> - the <regs> part describes how the **registers** are to be examined before and after the call, whereas the <template> defines the output string. The concept is this : the <template> is a C-style format string like those you would use to call exec/RawDoFmt() or dos/VPrintf(). The order of the arguments to this <template> is specified by the <regs> list, so that for each <template> format code the corresponding **<register> format data** is used.

EXAMPLES

base=dos,dos.library

include=snoopy:offsets/dos

watch=dos,-36,D1L,Close(\$%lx)

```
watch=dos,Open,D1L/D2L/RLD0,Open( "%s", $%lx ) = $%lx
```

Now, the base-statement should be quite clear. The include statement loads all defines required for the dos.library function offsets. You don't NEED to add this line, but if you don't you'll have to type "-30" instead of "Open" for the second "Watch"-statement. As you see, you can mix "old-style" number offsets and "new-style" function names defined in includes. What does this first watch-statement mean ?

"watch=" - the keyword tells Snoopy that this line is a watch description

"dos," - tells Snoopy that this watch applies to the dos.library

"-36," - tells Snoopy that you want to watch the function with the offset `_LVOClose`, which is -36

"D1L," - tells Snoopy that you want to see the Register D1 ("D1") and you want to interpret it as a Longword ("L")

"Close(\$%lx)" - tells Snoopy that you want to see a line "Close(`<hexadecimal representation of register D1>`)"

Understood ? Now take a look at the second statement and try to understand it. Its quite simple, really..... If you have problems making a script, try **BUILDWATCH** from the [support/](#) directory. More advanced "concepts" (ha!) can be seen in the example scriptfiles provided together with this distribution.

SEE ALSO

[The BASE keyword](#) , [The RESOURCE keyword](#) , [Register Format](#)

1.21 The DEVICE keyword

NAME

device - open a device for snooping

SYNOPSIS

```
device=<alias>,<device>,<IO size>[,<unit bits>]
```

FUNCTION

You have to give Snoopy a list of all devices you want to monitor. Because of the nature of the device-snooping, this syntax is a little bit more complex than the one for the BASE keyword (and it certainly has more to it as regards the internal implementation).

INPUTS

`<alias>` - an abbreviation for the device you want to monitor. If you don't restrict output to specific units you can use the part before the ".device" for this, if you want to see certain unit numbers, use the DOS device names such as "df0:", "df1:", "ram:" and so on.

<device> - the name of the device you want to watch. You ***CANNOT*** give a path here (unlike the BASE keyword), it must be the (case sensitive!) name of a device that is part of the SysBase->DeviceList, if you know what I mean.

<IO size> - it is VERY important that you give the LARGEST possible size of IO blocks this device will receive. The following list gives you an idea of sizes you should use

audio.device ioa_SIZEOF

clipboard.device iocr_SIZE

narrator.device NDI_SIZE

parallel.device IOEXTPar_SIZE

printer.device iodprp_SIZE (not iopcr_SIZEOF!)

sana2.device S2IO_SIZE

serial.device IOEXTSER_SIZE

timer.device IOTV_SIZE

trackdisk.device IOTD_SIZE

If your device is not in this list and it is a "trackdisk.device lookalike" (like mfm.device, static.device, fms.device or so) use IOTD_SIZE, else use IOSTD_SIZE. Note that this field understands only numbers, you must reference defines with \$<.> (see example below)

<unit bits> - a bitfield that specifies which units you want to see. If this field is -1, no unit checking is done; else, only units that match bits in this field are displayed; e.g. %110 will enable units 1 and 2 but disable unit 0. Note: UNITS ARE NOT SUPPORTED YET and will not be to the next updates (because unit handling is a little bit more complex)

EXAMPLES

device=yes,breakbeat.device

will open the (imaginary) breakbeat.device. Since its imaginary, I don't know about the size of its IO blocks, so I just assume its IOSTD_SIZE

device=serial,serial.device,\$<IOEXTSER_SIZE>

will open the serial.device (the size was taken from devices/serial.i)

device=static,static.device,\$<IOTD_SIZE>

this will open the static.device (a RAD: style ramdisk and thus a TD lookalike) for Snoopy

NOTE

many device-related constants are defined in offsets/devicesupport

LIMITATIONS

The device must have been already opened before you can snoop into it.

Sorry, but its the only way I could do it....(yet)

SEE ALSO

The **DEVCMDB** keyword

1.22 The DEVCMD keyword (BEGINIO,ABORTIO)

NAME

devcmd - snoop into a device function

beginio - snoop into a device function, BeginIO() only

abortio - snoop into a device function, AbortIO() only

SYNOPSIS

devcmd=<device>,<command>,<IO data>,<template>

beginio=<device>,<command>,<IO data>,<template>

abortio=<device>,<command>,<IO data>,<template>

FUNCTION

This is the most important keyword, since it specifies which command of a device you want to watch, and how this should be done. You really should understand the **WATCH** keyword before trying this function, since it works similar, but just adds more complexity (yow! ;-). This command comes in three versions, the DEVCMD is the recommended version whereas BEGINIO has exactly the same syntax but only works for calls to DEV_BEGINIO(); and ABORTIO works only for DEV_ABORTIO(). You can use this if you need a distinction between AbortIO() and BeginIO() calls (DEVCMD doesn't make this)

INPUTS

<device> - a device alias defined by a **device** keyword. Note that forward referencing is not possible, so you need to specify a device before you can use it in a watch

<command> - the number of the command you want to see [decimal by default, use prefix \$ for hex numbers], which should probably always be a positive number. If you have defined a command number, you can use that define without having to dereference it with the \$<..> syntax. Note that commands are 16-bit, that's an internal exec.library limitation. If the <command> is -1, Snoopy will ignore it and always display this device command. You can use this to find out what is actually going on with your device (i.e. if you don't want to look for specific devices but rather find out what somebody else really does)

<IO data>,<template> - the <IO data> part describes how the IO data this device receives is to be examined before a call, whereas the <template> defines the output string. The concept is this : the <template> is a C-style format string like those you would use to call exec/RawDoFmt() or dos/VPrintf(). The order of the arguments to this <template> is specified by the <IO data> list, so that for each <template> format code the

corresponding <IO data> item is used. Now this <IO data> field understands different things than the Register Format. Actually, it should consist of elements in the form

.../<size>:[offset]>/...

where [size] is B for BYTE, W for WORD and L for LONG data, and the offset, well, is the offset in the IO block. This is the reason why you had to give the IO blocksize earlier in the DEVICE keyword; Snoopy can read this memory only if it makes a copy of the device IO request. DO NOT USE any register specifications, Snoopy doesn't flag an error but it won't display them

EXAMPLES

```
device=td,trackdisk.device,$<IOTD_SIZE>
```

```
devcmd=td,CMD_READ,L:IO_LENGTH/L:IO_OFFSET/L:IO_DATA,CMD_READ %ld bytes from $%lx to $%lx
```

The first line opens the trackdisk.device, the second line declares that

Snoopy should snoop into CMD_READ actions and display three 32-Bit-Data fields from the IO requests.

```
abortio=td,CMD_READ,L:IO_LENGTH/L:IO_OFFSET/L:IO_DATA,ABORTIO(CMD_READ) %ld bytes from $%lx to $%lx
```

same as above, but ONLY for AbortIO() calls

LIMITATIONS

You can only show values that are initialised prior to the IO performance; which means there are no "return values". Sorry folks, but you just will have to live without these until I find a better way of doing this...

SEE ALSO

[The DEVICE keyword](#) , [The WATCH keyword](#)

1.23 The RESOURCE keyword

NAME

resource - open a resource for snooping

SYNOPSIS

```
resource=<alias>,<resource>
```

FUNCTION

You have to give Snoopy a list of all resources you want to monitor so that Snoopy can manage the function monitoring effectively. This is done by using the RESOURCE keyword with the above syntax. Resources are treated just as librarys, which means you have to use the **WATCH** keyword to define which functions you want to snoop into. Note that resource functions normally start at offset -6, whereas librarys usually start at -30

INPUTS

<alias> - an abbreviation for the resource you want to monitor. Typically this will be the part before the '.resource'.

<resource> - the complete name of the resource you want to watch (which is directly transferred to OpenResource()).

EXAMPLES

resource=misc,misc.resource

will open the misc.resource that is used to get access to system level hardware

resource=CardRes,cardres.resource

well, read autodocs/cardres.doc

LIMITATIONS

You can only snoop into resources that define global functions. For instance, the FileSystem is a resource, but it has no callable functions (at least none I know of) so you cannot do much with it.

SEE ALSO

exec/OpenResource, [The WATCH keyword](#)

1.24 The ALIAS keyword

NAME

alias - define an alias for a string

SYNOPSIS

alias=<name>,<contents>

FUNCTION

This function defines an alias for a string. Aliases are different to mere **defines** in that a) they (can) contain text and b) they can contain references to other aliases. The main reason for introducing aliases is the support for looking into structures [especially device IO blocks] starting from Snoopy 2.0 onwards and because I liked the idea of including them (acid overload, you know ;-). Aliases can be referenced anywhere in the scriptfile lines using the syntax `$(<name>)` where <name> is the name of that alias (case sensitive). The input parser will replace any valid `"$(<name>)"` sequence with its contents, so typing `alias=rc,RDB0L` would define an alias named `rc` with the contents `RDB0L`

INPUTS

<name> - the case sensitive name of the alias

<contents> - its contents

EXAMPLES

alias=lib,watch=dos,

```
alias=regs,D0L/D1L/D2L/D3L/D4L/D5L/D6L/D7L/A0L/A1L/A2L/A3L/A4L/A5L/A6L/A7L
```

```
alias=regrow,%08lx %08lx %08lx %08lx %08lx %08lx %08lx %08lx
```

```
alias=out,$(regs),dos.library/Open()\nDx:$(regrow)\nAx:$(regrow)
```

```
$(lib)Open,$(out)
```

```
$(lib)Close,$(out)
```

the above script watches Open() and Close() from the dos.library, and displays all registers of these two functions

LIMITATIONS

Forward referencing is not possible, but backward references surely are:

```
alias=foo,.library
```

```
alias=dos,dos$(foo)
```

will result in "foo" containing ".library" and "dos" containing

"dos.library". You can use aliases even to redefine internal commands,

such as in

```
alias=son,segtracker=on
```

```
alias=soff,segtracker=off
```

```
...
```

```
$(son)
```

```
...
```

```
$(soff)
```

even the following will work

```
alias=this is a very stupid useless macro but I like it,alias=
```

```
$(this is a very stupid useless macro but I like it)yow,y
```

Note that you cannot make two aliases with the same name. Actually you

can, but only the first alias will work. Also, you cannot "construct" an

alias inside an alias : the following sequence would not work

```
alias=name,dos
```

```
alias=begin,$(
```

```
alias=end,)
```

will result in "\$ (begin)name\$(end)" = "\$ (name)", but since THE PARSER HAS

ONLY ONE RUN it won't be translated. You could use

```
alias=name,dos
```

```
alias=calldos,$(name)
```

which would make "\$ (calldos)" = "dos", but no other fancy tricks are

allowed. If you like tricky stuff, try experimenting with it and see if

you get any fun out of it... ;-). One more restriction (as if there

weren't enough already) you cannot define aliases for comment identifiers

such as ';' and '*', and neither for alias references '\$(')' or '\$< ' >'

SEE ALSO

[The DEFINE keyword](#) , [The BITDEF keyword](#) , [The ECHO keyword](#)

1.25 The DEFINE keyword

NAME

define - assign a logical name to a number

SYNOPSIS

```
define=<name>,<offset>
```

FUNCTION

This function is used to assign logical names to numbers, which normally represent function or structure offsets. Note that unlike **aliases**, defines are only numbers and thus restricted to contain only digits (decimal, hex or binary based upon the numbering scheme), but they are more purpose-specific; This keyword is most commonly used inside **includes** and is (a little bit) similar to #DEFINE statements in "C", or "EQU", "=" in assembler.

Some commands understand defines directly, such as the **WATCH** and **DEVCMDB** statements. However, you can reference defines **ANYWHERE** in the scriptfile using the syntax `$$[[name]]>` where [name] is the logical name of a number (case sensitive). The input parser will replace any valid `$$[name]>` sequence with their decimal contents, so typing `$$Open>` will place the string -30 in the text. If you prefix the name in brackets with another dollar sign, you will get the hex representation, so `$$Open>` will actually give you `-$1e`

INPUTS

`<name>` - The logical name, which is case sensitive and cannot contain any wildcards whatsoever. Note that you can use the same `<name>` more than once, Snoopy keeps some kind of "scope" in which a define is valid; in this case up to the next define keyword with the same `<name>`.

`<offset>` - The (signed long) decimal offset, which should be negative for function offsets, positive for most structure offsets. Note that decimal numbers are default, but you can give hex numbers with a leading '\$', or dual numbers with a '%'

EXAMPLES

```
define=Open,-30
```

```
define=Open,-$1e
```

```
* "I wouldn't normally do this kind of thing" ;-)
```

```
define=ObtainQuickVector,-%1100010010
```

this example shows how you can use decimal, hex or dual numbers when defining library offsets

```
define=LN_SUCC,0
```

```
define=LN_PRED,4
define=LN_TYPE,8
define=LN_PRI,9
define=LN_NAME,10
define=LN_SIZE,14
```

the statements above define the node structure as used by exec (and device IO)

LIMITATIONS

forward referencing is not possible: you must define offsets before you can use them.

SEE ALSO

[The INCLUDE keyword](#) , [The INCDIR keyword](#) , [The ALIAS keyword](#)

1.26 The BITDEF keyword

NAME

bitdef - define an (unsigned) integer by its bit number

SYNOPSIS

```
bitdef=<name>,<bit>
```

FUNCTION

The main use for this function is the future: Probably the next Snoopy update will have some kind of "result analyser" that will replace register values by defines (you know, like defining those MEMB-bits and tell Snoopy to use them for register d1). Currently, BITDEF is excellent only for making cryptic scriptfiles (in the great tradition of FALSE - *the* language ;-). It works exactly the same as **DEFINE** , only that the contents of "name" are not "bit" but rather "1<<bit". Of course, BITDEFs are referenced the same as DEFINES are

INPUTS

<name> - The name for this define, which is case sensitive and cannot contain any wildcards whatsoever. Note that you can use the same <name> more than once, Snoopy keeps some kind of "scope" in which a define is valid; in this case up to the next define keyword with the same <name>.

<bit> - The decimal bit number (0-31). Note that decimal numbers are default, but you can give hex numbers with a leading '\$', or dual numbers with a '%' [although the later really is quite pretentious and probably used only by sucker MCs]

EXAMPLES

```
bitdef=chip,2
```

bitdef=funny,%11001

echo=\$<chip>,\$<funny>

will display a requester containing the message "4,33554432"

LIMITATIONS

Snoopy doesn't check if the bit number is less than 32. If you use numbers greater or equal to 32 you are a very very strange and troubled person and should probably get some kind of medical treatment before you shoot everybody up, start crying because nobody loves you and join the navy.

SEE ALSO

[The DEFINE keyword](#)

1.27 The INCLUDE keyword

NAME

include - include another scriptfile

SYNOPSIS

include=<filename>

FUNCTION

This keyword is used to include the contents of another scriptfile into the current scriptfile. For most purposes includefiles will contain function and structure defines for certain libraries or devices. The [offsets/](#) directory provided with this distribution has includefiles for all LVO-files I found on my harddisk. However, since the input parser doesn't make much of a difference between includefiles and scriptfiles, you can include whole scriptfiles to your main scriptfile. Therefore you can include any of the example scripts provided in the [scripts/](#) subdirectory (part of this distribution). If the Snoopy parser encounters an INCLUDE statement, it will try to open the file with the given name and process it just as it would process a normal scriptfile; which means that included scriptfiles can now contain ALL of Snoopys keywords

INPUTS

<filename> - the filename for the includefile, with or without additional path information. Snoopy will look for this file first in the current directory, then (if given) in the directory specified with the current [INCDIR](#) option.

EXAMPLES

include=dos

includes the file "dos" in either the current directory, or if it cannot be found there, wherever you have assigned your INCDIR to

include=dh2:snoopy/offsets/dos

includes the file "dh2:snoopy/offsets/dos" in the absolute path dh2:

(because if you specify a non-relocatable path, Snoopy has to stick to it)

LIMITATIONS

there can only be up to MAX_INCLUDELEVEL=16 includefile levels (of course, there is an unlimited number of includefiles). This applies to situations

where script A includes script B

script B includes script C

script C includes script D

... and so on ...

There is no "multiple-includes" feature, just this hierarchy protection, so that code like the below will eventually fail (after 16 steps)

script A includes script B

script B includes script A -> recursion

SEE ALSO

[The INCDIR keyword](#) , [The DEFINE keyword](#) , [The ALIAS keyword](#)

1.28 The INCDIR keyword

NAME

incdir - set directory for includefiles

SYNOPSIS

incdir=<path>

FUNCTION

If you use the **INCLUDE** feature of Snoopy you'll probably place all includes in one directory (like **offsets/** in this distribution). To make typing faster you can define include directories where Snoopy should look for all your includes (up to the next INCDIR statement). If you keep path empty, (that is, don't use this keyword or type something like "INCDIR=;") Snoopy will use only the current directory as the include path. Note that you can use more than one INCDIR statement, they are valid only up to the next INCDIR statement.

INPUTS

<path> - some valid AmigaDos path

EXAMPLES

incdir=snoopy:offsets

include=dos

incdir=dh2:offsets

include=graphics"

Snoopy first tries to load snoopy:offsets/dos, then dh2:offsets/graphics

NOTE

The current directory has priority over INCDIR directorys, so Snoopy will try to load from the current directory first, always.

SEE ALSO

[The INCDIR option](#) , [The INCLUDE keyword](#)

1.29 The HIDE keyword

NAME

hide - hide a specific task

SYNOPSIS

hide=<taskname>

FUNCTION

You can hide certain tasks from Snoopy. This is a very usefull feature because it reduces the amount of output given by Snoopy, thereby allowing you to look only for calls by specific tasks. An example: If you want to monitor critical calls to DOS functions like Open() and Close(), why bother about calls made from the Workbench ? They are harmless and probably ok - and besides you should think that the commodore programmers make some reasonable code in their own operating system (at least if you're not working with obscure beta release versions ("developers only")). By hiding the Workbench from Snoopy you will get a lot less calls to Open() and Close() (which are called, for example, on each icon when you enter a directory on your drive!). Besides, you probably still see all the important calls, don't ya ?!

INPUTS

<taskname> - the name of the task you want to hide. Unless the **MATCH** option is given, the <taskname> is actually an abbreviation (not case sensitive!) of the real task name; so for example hide=Work would hide both the tasks Workbench and WORKING CLASS HERO. If the task is a CLI, give the name of the command loaded in this CLI. You can get all the current task names by using the **TASKLIST** tool from the [support/](#) directory of this distribution.

Alternatively, you can specify the address of a task if it is too hard to find the task by name. You do this by giving a hex number starting with a dollar sign '\$'

EXAMPLES

hide=Workbench

will hide the Workbench (and everything that is named "similar")

hide=\$F3991E

will hide the task at the address 0xF3991E given that this address points to a task. Snoopy doesn't check it, it just ignores invalid addresses.

This was added especially because some tasks have either ridiculous names or no name at all

SEE ALSO

The **SHOW** keyword , The **MATCH** keyword

1.30 The SHOW keyword

NAME

show - set explicit tasks to look for

SYNOPSIS

show=<taskname>

FUNCTION

Sometimes it is more interesting to see calls of only one task (or: a selected list of tasks) instead of hiding all tasks that can be ignored.

For instance, if you are debugging task "A" and want to see calls relevant only for this task, you would have to hide all other tasks [and there are many on them on a multitasking machine like the amiga]. The **SHOW** keyword does just this; it allows you to give an explicit list of tasks you want to snoopy into. Note that **SHOW** disables the effects of any **HIDE** commands, it just doesn't make much sense.

INPUTS

<taskname> - the name of the task you want to show. Unless the **MATCH** option is given, the <taskname> is actually an abbreviation (not case sensitive!) of the real task name; so for example show=Work would show both the tasks Workbench and WORKING CLASS HERO. If the task is a CLI, give the name of the command loaded in this CLI. You can get all the current task names by using the **TASKLIST** tool from the [support/](#) directory of this distribution.

Alternatively, you can specify the address of a task if it is too hard to find the task by name. You do this by giving a hex number starting with a dollar sign '\$'

EXAMPLES

show=Workbench

will show ONLY the Workbench (and everything that is named "similar")

show=\$F3991E

will show only the task at the address 0xF3991E given that this address points to a task. Snoopy doesn't check it, it just ignores invalid addresses. This was added especially because some tasks have either ridiculous names or no name at all

SEE ALSO

[The HIDE keyword](#) , [The MATCH keyword](#)

1.31 The PRI keyword

NAME

pri - set running priority

SYNOPSIS

pri=<value>

FUNCTION

One very important thing about Snoopy is that you can change its running priority. This is VERY usefull [and therefor has been added in version 1.1] because some (=several(=many)) programs use local = volatile memory such as the stack for setting up their arguments. The problem is that if you place a string on the stack, the memory gets (probably) scratched by the time Snoopy is able to show it. If however you set Snoopy priority higher than the task calling the output is made BEFORE the memory is made invalid => lucky charm ! An example : tracking down calles by "internal" programs such as "alias" : if you use PRI=0 [default] you'll get scratched args for _Open(); if you use PRI=3 [recommended] you'll get the real thing! Great! Disko, Disko!

INPUTS

<value> - the numeric value (signed decimal) of Snoopys priority. You should use prioritys in the range -3 to +3. I cannot guarantee for any Snoopy function if you use prioritys outside this range : if your pri is too high, you might crash the event loop (especially if your pri is higher than the pri of the output handle (i.e."CON:")), if it is too low you might never see anything. I recommend using "PRI=3" for most purposes.

NOTE

if you use the **PRI command line argument** it takes priority over any PRI statement in a scriptfile, which means that any PRI statement in the script will be ignored

EXAMPLES

pri=3

sets the runtime priority to +3 which is the recommended level for most

scriptfiles.

pri=-1

If you are patient, use this priority to receive messages very slowly but with very very few system overhead.

LIMITATIONS

only one PRI statement per scriptfile is taken into account (actually: the last PRI statement in a scriptfile)

SEE ALSO

[The PRI option](#)

1.32 The ECHO keyword

NAME

echo - display a progress message

SYNOPSIS

echo=<message>

FUNCTION

If Snoopy finds this keyword, it will display the <message> in a requester.

If Snoopy doesn't accept one of your scriptfiles, you can use this keyword to somehow "debug" it. What you would probably do is: place "echo"s around statements that look suspicious, and see how far Snoopy goes.

Manytimes errors occur due to misuse of [aliases](#) and defines; with this option you can see what the aliases and defines really translate to (because they can be used in the <message>, of course). The requester has Ok and Abort buttons, which means you can break the scriptfile parser if you like to do so.

INPUTS

<message> - a string

EXAMPLES

```
alias=REM,\;
```

```
alias=RANDOM1,\$(REM)
```

```
alias=RANDOM2,\ \$(REM)
```

```
alias=RANDOM3,\ $(REM)
```

```
echo=Alias 1 is "\$(RANDOM1)"
```

```
echo=Alias 2 is "\$(RANDOM2)"
```

```
echo=Alias 3 is "\$(RANDOM3)"
```

will display three requesters with the following contents

```
Alias 1 is "\$(REM)"
```

```
Alias 2 is "\ \$(REM)"
```

```
Alias 3 is "\ "
```

SEE ALSO

[The ALIAS keyword](#) , [The DEFINE keyword](#) , [The BITDEF keyword](#)

1.33 The OUTPUT keyword

NAME

output - set another output handle

SYNOPSIS

output=<filename>

FUNCTION

You can give a filename to redirect Snoopys output to. This might come in handy if you know that there'll be lots of calls or you want to examine the output later (or whatever). Just give the name of the file you want Snoopy to print its messages to. If this argument is not given, Snoopy will open its standard output window, which in turn will be "con:0/0/640/150/Snoopy".

INPUTS

<filename> - the name of the file Snoopy should redirect its output to.

This can be a device name such as PRT: or SER:, a filename such as "ram:output" or a window definition such as "con:0/0/640/200/Snoopy"

EXAMPLES

output=prt:

will send all Snoopy messages for this scriptfile directly to your printer

SEE ALSO

[The OUTPUT/K option](#)

1.34 The Sticky keyword

NAME

sticky - enable/disable self-detaching mechanism

SYNOPSIS

sticky=<boolean>

FUNCTION

If you don't like Snoopy to detach itself, use this option. This was especially designed to enable arguments like **OUTPUT=*** (which redirects Snoopy output to the current CLI). You can also use it to be able to break redirected Snoopys with CTRL+C such as in "snoopy sticky output=ram:snoopy.output". Workbench ignores this command as it doesn't make any sense; if you start any program from Workbench, it will always be detached.

INPUTS

<boolean> - "YES", "TRUE", "ON", "ACTIVE" if activated, "NO", "FALSE", "OFF", "NOT ACTIVE" otherwise.

NOTE

Due to a bug (or a feature?) in dos/ReadArgs() it is not possible to know if a boolean CLI argument has been given on the command line or not.

Because of this Snoopy gives script statements a priority; which means that the **STICKY/S command line option** is overwritten by any STICKY statement in a scriptfile

EXAMPLES

sticky=active

will cause this scriptfile to be run only by sticky Snoopys ;-)

LIMITATIONS

Only the very last STICKY statement of a scriptfile is taken into consideration.

SEE ALSO

[The STICKY/S option](#)

1.35 The MATCH keyword

NAME

match - enable/disable exact string matching

SYNOPSIS

taskinfo=<boolean>

FUNCTION

If you use this statement in a script file, Snoopy will use case sensitive name matching instead of the default, case insensitive algorithm. This means, if MATCH is OFF, "disko" equals both "DISKO" and "disko LOVER", if MATCH is ON, it equals NEITHER OF BOTH. The Syntax is

INPUTS

<boolean> - "YES", "TRUE", "ON", "ACTIVE" if activated, "NO", "FALSE", "OFF", "NOT ACTIVE" otherwise.

NOTE

Due to a bug (or a feature?) in dos/ReadArgs() it is not possible to know if a boolean CLI argument has been given on the command line or not.

Because of this Snoopy gives script statements a priority; which means that the **MATCH/S command line option** is overwritten by any MATCH statement in a scriptfile

EXAMPLES

taskinfo=active

will enable taskinfo for that script

LIMITATIONS

Only the very last MATCH statement of a scriptfile is taken into consideration.

SEE ALSO

[The MATCH/S option](#)

1.36 The TASKINFO keyword

NAME

taskinfo - enable/disable taskinfo flag

SYNOPSIS

taskinfo=<boolean>

FUNCTION

If you use this statement in a script file, Snoopy will print out the address and the name of each task that called a patched function before the patch message itself. This helps you to determine who called which function in which particular order - a very usefull feature.

INPUTS

<boolean> - "YES", "TRUE", "ON", "ACTIVE" if activated, "NO", "FALSE", "OFF", "NOT ACTIVE" otherwise.

NOTE

Due to a bug (or a feature?) in dos/ReadArgs() it is not possible to know if a boolean CLI argument has been given on the command line or not. Because of this Snoopy gives script statements a priority; which means that the **TASKINFO/S command line option** is overwritten by any TASKINFO statement in a scriptfile

EXAMPLES

taskinfo=active

will enable taskinfo for that script

LIMITATIONS

Only the very last TASKINFO statement of a scriptfile is taken into consideration.

SEE ALSO

[The TASKINFO/S option](#)

1.37 The SEGTRACKER keyword

NAME

segtracker - enable/disable segtracker support for specific functions

SYNOPSIS

segtracker=<boolean>

FUNCTION

Snoopy is capable of SegTracker support, given that you have installed this program. If SegTracker is running, Snoopy will always display SegTracker messages. However, you can partially (or globally) disable or enable

SegTracker support for specific library function using this keyword.

INPUTS

<boolean> - "YES", "TRUE", "ON", "ACTIVE" if activated, "NO", "FALSE", "OFF", "NOT ACTIVE" otherwise. Note that you can use multiple SEGTRACKER statements to enable output for some, disable for others : the parser has some kind of scope mechanism which means that one SEGTRACKER setting is valid only up to the next SEGTRACKER statement

NOTE

Due to a bug (or a feature?) in dos/ReadArgs() it is not possible to know if a boolean CLI argument has been given on the command line or not.

Because of this Snoopy gives script statements a priority; which means that the **SEGTRACKER/S command line option** is overwritten by any SEGTRACKER statement in a scriptfile

EXAMPLES

```
segtracker=yes
```

```
watch=dos,Open,D1L/D2L/RBD0L,Open( "%s", $%lx ) = %s
```

```
watch=dos,Close,D1L/RBD0L,Close( $%lx ) = %s
```

```
segtracker=no
```

```
watch=dos,Lock,D1L/D2L/RBD0L,Lock( "%s", $%lx ) = %s
```

```
watch=dos,UnLock,D1L,UnLock( $%lx )
```

This example will enable SegTracker output for both Open() and Close(), but disable it for Lock() and UnLock()

SEE ALSO

[Segtracker Support](#) , [The SEGTRACKER/S option](#)

1.38 The ASSUME keyword

NAME

assume - set assumption for register templates

SYNOPSIS

```
assume=<assumption>
```

FUNCTION

This keyword can be used to set the default assumption for a register template; that is the default register size. If you don't specify explicit register sizes, Snoopy will use this keyword to determine what size extension you possibly want. Note that you nevertheless have to take care about the output yourself; that is: if you assume LONG you must use %ld instead of %d in the output template and so on...This keyword was added to make scripts more readable, because historically Snoopy defaults to 16-bit

sizes, but it seems to me that the vast majority of functions take 32-bit arguments, so why not use this....

INPUTS

<assumption> - LONG for 32-bit data, WORD for 16-bit, BYTE for 8-bit

EXAMPLES

assume=LONG

watch=.....D0/D1....%x %x

here, both d0 and d1 are assumed to be 32-bit data

assume=WORD

watch=.....D0/D1....%x %x

here, both d0 and d1 are assumed to be 16-bit data

SEE ALSO

[The register format](#)

1.39 The SKIPSIMILAR keyword

NAME

skipsimilar - enable/disable skipping of "similar" functions

SYNOPSIS

skipsimilar=<boolean>

FUNCTION

This statement was added so that you can "pack" the output. If this argument is on and the same function is called two or more times without any other function called inbetween, Snoopy will show only the first time the output was made. For instance, if you snoop into Lock() and try to load a file on your path, the system will call Lock() for each entry of that path, so that you can get some 50 Locks one after the other. Now while this is correct some people find it annoying - you miss out the important calls because all you see is Locks(). This keyword is what you've been waiting for: if enabled, Snoopy will show only the first of two or more consecutive similar calls (hence the name).

INPUTS

<boolean> - "YES", "TRUE", "ON", "ACTIVE" if activated, "NO", "FALSE", "OFF", "NOT ACTIVE" otherwise. Note that you can use multiple SEGTRACKER statements to enable output for some, disable for others : the parser has some kind of scope mechanism which means that one SKIPSIMILAR setting is valid only up to the next SKIPSIMILAR statement

EXAMPLES

skipsimilar=not active

```
watch=dos,Open,D1L/D2L/RBD0L,Open( "%s", $%lx ) = %s
```

```
watch=dos,Close,D1L/RBD0L,Close( $%lx ) = %s
```

```
skipsimilar=active
```

```
watch=dos,Lock,D1L/D2L/RBD0L,Lock( "%s", $%lx ) = %s
```

```
watch=dos,UnLock,D1L,UnLock( $%lx )
```

This example will show all output for both Open() and Close(), but skip similar output for both Lock() and UnLock()

1.40 The DELAY keyword

NAME

delay - set timeout delay for important functions

SYNOPSIS

```
delay=<ticks>
```

FUNCTION

If you use Snoopy to help you debugging you might want to pause after calls to certain functions. This keyword can be used to set a timeout delay for a function. If your application then calls this function, it will be delayed for as many <ticks> as you have specified (AFTER having sent the message to Snoopy, so you will always see the delayed function on the display). If you specify -1 as the ticks value, the CALLER will be put to sleep and wait for CTRL+C signals to arrive. This means: you **send CTRL+C** to the CALLER, NOT TO SNOOPY!!!! and the task continues. If you send CTRL+C to SNOOPY, Snoopy tries to end itself, which will not work because some patch is still active and as long as patches are working Snoopy cannot be removed. This function has a scope, so the timeout delay is used for all functions up to the next DELAY statement. use "DELAY=0" to disable the delay feature.

INPUTS

<ticks> specifies how many ticks (50 per second) to wait before returning control. If <ticks> is -1, the calling task will wait to receive a CTRL+C.

You can send this by pressing CTRL+C or using the CLI command BREAK if the caller is an AmigaDOS process (i.e. listed by "STATUS") or using my **BREAK** replacement from the **support/** directory.

NOTE

do NOT set a timeout delay for time-critical functions or for exec/Wait(), dos/Delay() and stuff like that.

NOTE

delays are caused by ALL calls : this means that even if you have hidden

most of the tasks, somebody may still call your delayed function and thus cause the delay to occur. This may be a problem if you use the BREAK feature, because you then do not know who caused the Wait() and cannot send the break signal to continue. Generally, use DELAY only for functions you are sure are safe to delay (i.e. use your intelligence, you're a human and not some stupid machine)

NOTE

delays need the dos.library and thus should only be set for functions that are called only from DOS processes. This means, if a function might be called from an interrupt, DO NOT set a delay for this function.

EXAMPLES

```
delay=-1
```

```
watch=debug,SmartFunction,D0L/RD0L,d0 changes from %lx to %lx
```

```
delay=0 ;reset the timeout delay
```

this example will watch the SmartFunction() in your debug base and each time wait for you to send CTRL+C to it

SEE ALSO

[support/BREAK](#)

1.41 Debugging your own application code - concept & ideas

DESCRIPTION

Although there are several debuggers available on the Amiga (and good ones, too, such as the PowerVisor) many programmers use only ENFORCER because it finds all the serious bugs and why bother tracing down all your local calls. Especially when writing large assembler programs (such as Snoopy ;-)) using source-level debuggers is quite impracticable; because all you do is single-step thru kilobytes of code (and a kilobyte means a LOAD of instructions) or wait for breakpoints to happen (which is like "Waiting for Godot" if you know what I mean) and besides some errors just are too tricky to find out (what about this one : index overflow in one of the 127 loops, would you single-step or rather try something else?). Now Snoopy gives assembler programmers an easy yet powerfull way of debugging programmes (System level that is, fuck "org \$60000" hacker kids) : It incorporates a way of calling your assembler subroutines via Snoopy, so that all registers pre/post call are sent to Snoopy! This means, just as you would watch library functions, you now can watch your own assembler subroutines! The only difference is that your routine must close with RTS; and that your jump must be BSR or JSR, not BRA or JMP.

This debugging package consists of a single sourcecode SNOOPYDEBUG.S and a single includefile SNOOPYDEBUG.I which have to be included somewhere in your source (of course you could easily make it a link module). What you do is this:

- 1.include SNOOPYDEBUG.I at the top of your code [or each source if you have a project such as Snoopy]
- 2.include SNOOPYDEBUG.S somewhere at the bottom of your code [it is NOT an includefile but a source code; so place it somewhere safe]
- 3.change your BSR/JSR calls to the SNOOPYDEBUG macro defined in SNOOPYDEBUG SNOOPYDEBUG <Pointer To Label>,<Template>

where <LABEL> is the address of your subroutine. Remember to prefix this with # if its a real label, else you'll crash inevitably! <Template> is either #0 (which means no template, show all registers) or a pointer to a **template description** . Note: This is NOT the same as the register format template, its different because it has to be directly inserted in your code. SNOOPYDEBUG is defined as

SNOOPYDEBUG MACRO

```
move.l \2,-(sp)
move.l \1,-(sp)
jsr SnoopyDebug
ENDM
```

don't worry if you think this looks strange; SnoopyDebug() cleans up your stack so this code works perfect. The Subroutine at <LABEL> will be called with all registers except a6, and of course different condition codes. This means, the following code is crap, really :

```
SNOOPYDEBUG #ReadBlock,#0
beq.s BLOCK_READ
```

because the "beq.s" instruction reads undefined condition codes. Use a TST or something, it won't harm you, will it ?!. Note that a6 is NOT scratched to the outside, only to the inside; which means the main program doesn't have to preserve a6; only the called subroutine has to reload it. Ok, this is about as much as it takes to incorporate SnoopyDebug() in your code.

All calls made via SNOOPYDEBUG will be sent to a snoopy instance described somewhere in your code as

```
SnoopyPortName: dc.b "Snoopy Port.1",0
```

The ".1" is the instance number, so you can send the output even to different Snoopy windows. Note that it is even possible to start Snoopy with an EMPTY scriptfile (not WITHOUT any), which means that this particular Snoopy will receive only calls by your application!

EXAMPLES

```

SNOOPYDEBUG #ReadBlock,#0
will jump to the subroutine labeled "ReadBlock:" and show all registers
include include:exec/types.i
include snoopydebug.i
; main program
moveq #20,d0
SNOOPYDEBUG #Funktion,#0
rts
; subroutine that does some tricky stuff
Funktion: moveq #0,d0
rts
; enable debug support
include snoopydebug.s
; this is the port that should receive all messages
SnoopyPortName: dc.b "Snoopy Port.1",0
end

```

This example is a minimum sourcecode that shows all registers.

NOTE

Of course, SnoopyDebug() takes up some time. Its very short and considerably fast, but do not use SnoopyDebug() inside time critical stuff, especially not inside interrupts, when relying on timer values, or making synchronized time-critical IO. You CAN use it there, but just don't complain if your stuff doesn't work because its too slow : I warned you. While you are debugging your code it is probably more important that you find some error than that the error takes exactly 3.5 processor cycles CPU time [don't you just hate the "pixel war", lamer #48123 can do 23 pixels more per blit than lamer #48124 and thus "rules" - ha!]

NOTE

It is possible to nest SNOOPYDEBUG calls; which means you can use SNOOPYDEBUG inside functions called by SNOOPYDEBUG. The function is reentrant and pc-relative, so the code can be even relocated. However, you ***MUST*** keep SnoopyPortName in the same segment as the SNOOPYDEBUG.S code. Look at the code (its documented) if you have any doubts.

NOTE

Don't panic if you're a C-Programmer, the next version of Snoopy has C-support for SnoopyDebug() and some special C stuff; because of course normally you won't pass your arguments in C by register but by stack.

NOTE

The source has no "special effects" except for some very basic macros (written in "old syntax"). It is understood without any problems on MACRO68 and on ASMONE;
SEE ALSO

[Template Description](#)

1.42 SnoopyDebug() Templates

DESCRIPTION

You can define templates for SnoopyDebug(). However, these templates must be directly incorporated in your code, because you have to send the template to Snoopy (as part of the message SnoopyDebug sends). The Template is a varying sort of array, it looks something like this:

```
Template: dc.l <Pointer to Template String>
dc.w <Template Element>
...
dc.w -1 ;End of Template
```

Here, <Pointer to Template String> just points to the usual zero-terminated string you would define for WATCH-statement templates. However, the <Template Elements> are what is really new; they describe the registers directly. All elements are WORDs that contain a register value plus additional flags. They are defined in SNOOPYDEBUG.I as:

```
;----- additional flags
BITDEF REG,RESULT,6 ; return value
BITDEF REG,BOOLEAN,7 ; boolean style output
BITDEF REG,BYTE,8 ; size = byte
BITDEF REG,WORD,9 ; size = word
BITDEF REG,LONG,10 ; size = long
;----- the register values
REGISTER_D0 equ %001000
REGISTER_D1 equ %001001
REGISTER_D2 equ %001010
REGISTER_D3 equ %001011
REGISTER_D4 equ %001100
REGISTER_D5 equ %001101
REGISTER_D6 equ %001110
REGISTER_D7 equ %001111
REGISTER_A0 equ %010000
REGISTER_A1 equ %010001
```

```
REGISTER_A2 equ %010010
REGISTER_A3 equ %010011
REGISTER_A4 equ %010100
REGISTER_A5 equ %010101
REGISTER_A6 equ %010110
REGISTER_A7 equ %010111
REGISTER_I0 equ %100000
REGISTER_I1 equ %100001
REGISTER_I2 equ %100010
REGISTER_I3 equ %100011
REGISTER_I4 equ %100100
REGISTER_I5 equ %100101
REGISTER_I6 equ %100110
REGISTER_I7 equ %100111
REGISTER_DONE equ -1
```

For example, "dc.w REGISTER_D0|REGF_RESULT|REGF_LONG" would be the same as a "RD0L" element in a WATCH statement. I think you get the idea...

EXAMPLES

```
SNOOPYDEBUG #MyHandler,#Template
```

...

```
Template: dc.l .l
```

```
dc.w REGISTER_D0|REGF_LONG
```

```
dc.w REGISTER_A0|REGF_LONG
```

```
dc.w REGISTER_D0|REGF_LONG|REGF_RESULT
```

```
dc.w -1@
```

```
.l dc.b "MyHandler() gets %lx and %lx; returns %lx",0 ;NO LINEFEED!
```

would be the same as a "D0L/A0L/RD0L,MyHandler().." part of a WATCH statement.

NOTE

It is (of course) possible to define an empty register list:

```
Template: dc.l .l
```

```
dc.w -1
```

```
.l dc.b "CALLED MY FUNCTION!",0
```

SEE ALSO

[Debug concept & Ideas](#)

1.43 Boolean output

DESCRIPTION

Starting with 1.7, Snoopy can give out boolean values instead of the hex numbers because for many actions you just want to know if they succeed or not. To do this, you have to do two things :

- give option 'B' in the register template
- use '%s' in the regarding field of the output template.

EXAMPLE

```
watch=dos,Open,D1L/D2L/RD0L,Open( "%s", $%lx ) = $%lx
```

will return output such as

```
Open( "disko/ultraworld", $1234 ) = $6543210
```

whereas

```
watch=dos,Open,D1L/D2L/RBD0L,Open( "%s", $%lx ) = %s
```

will return output such as

```
Open( "disko/ultraworld", $1234 ) = YES
```

or

```
Open( "disko/ultraworld", $1234 ) = NO
```

NODE

You ***MUST*** use %s in the output template to see the success/failure message.

1.44 Segtracker support

DESCRIPTION

Starting with Version 1.7, Snoopy supports SegTracker to help you debugging. If you don't know what SegTracker is all about, read the Enforcer documentation first because I really cannot give you all the details here. From this version on Snoopy will automagically know if SegTracker is in your system and add a detailed Segment/Hunk/Offset/SegList description to the display. You will get a line looking something like this :

```
$<address>-<filename>:Hunk <number>,Offset $<address>,SegList $<address>
```

Note that Snoopy will use "(pc)" to find the segtracker address, this way the segment address is where the jump occurred. Note also that since not all SegTracker versions support the SegList feature, a SegList of \$0 means no SegList found. Also note that internal commands do not have a SegList, so don't expect one.

Thanks to Christian Rattei for pointing SegTracker out to me...

SEE ALSO

The **SEGTRACKER** keyword , enforcer documentation

1.45 Running multiple instances of Snoopy

DESCRIPTION

There are a few problems involved when running multiple instances of Snoopy. Some of these have been fixed for version 2.0, others are unfortunately not so easy to remove because they result from the way Snoopy handles things. The first new thing is that now every instance has its own number, starting with 1. This means, that if you have three Snoopys running at the same time, you'll get three tasks named

Snoopy Task.1

Snoopy Task.2

Snoopy Task.3

Note that Snoopy displays the instance number in [] brackets at startup;

the first line of the Snoopy output should be something as

```
"Welcome to Snoopy 2.0 (Dec.16,1993)[1] - written by G.Kurz"
```

^

Instance number

All Snoopy instances are fully self-content, which means that they don't really know of each others existence : each does what it has to do with the scriptfile provided. If you use different scriptfiles (more precisely: watch different library functions) you can remove any copy at any time.

However, if you use the same scriptfile (or better, watch the same functions in scriptfiles) you have to remove the instances in the order they were created. If you try to remove an "older" copy while a "younger" one is still running you will get the same warning message as if something else would have patched library functions.

SNOOPYS "SAVE CLOSING" mechanism

The above is what is called Snoopys "save closing" mechanism that now has a little more to it than before. It generally works like this :

- "Snoopy Task.<i>" (Instance = <i>) patches functions a(),b(),c()
- "Someone else" patches one of these functions, lets say b()
- If you remove "Snoopy Task.<i>" without having removed "Someone else" before, Snoopy will give you a warning message telling you that it couldn't remove the function with the template for b(). Note that a() will be successfully removed anyway, whereas c() will still be there (because the mechanism stops the first time it encounters such a problem).

Now, "Someone else" is, most prominently, another instance of Snoopy, or something else like SnoopDos. As you see the only time Snoopy really cares about multiple instances is when a) creating its task/port names and b)

when removing itself.

If you use the QUIT command line option, you have to specify the instance number, since it is not (yet) possible to remove all instances in one command. Also note that the QUIT option works only if you specify a number and only if Snoopy can find this instance. If the instance cannot be found, Snoopy will issue a warning telling you about it.

EXAMPLE

```
start instance 1
```

```
1> Snoopy
```

```
start instance 2
```

```
1> Snoopy
```

will result in an error, since both instances use "s:snoopy.script"

```
1> Snoopy Quit=1
```

this is what you have to do

```
1> Snoopy Quit=2
```

```
1> Snoopy Quit=1
```

As if you didn't know already : It is of course much easier to remove a Snoopy instance by sending CTRL+C to its window. This option is really usefull if you redirect Snoopys output, but its a drag if you use the standard windows.

KNOWN PROBLEMS

1 - it is possible that SegTracker output loses segments because by the time Snoopy asks for the segment details the memory might be invalid again. Although Snoopy asks for the PC segment address this can be a problem if you run two copies at high priority and have some call at the end of file; by the time the system unloads the file (probably InternalUnLoadSeg()), the PC is -of course- invalid so SegTracker doesn't find anything. Well, you can't have them all...

2 - it is a very problematic issue to run multiple instances with different prioritys, especially when the one with the higher priority is "always busy" you might not see any calls for the other Snoopy instance **ALTHOUGH INTERNALLY THEY ARE MADE**. This means, that the low-pri instance just gets more and more messages (because the system message mechanism still works) but can process them only by the time the hi-pri Snoopy can wait again; which in turn means that the low-pri Snoopy eats up memory [because Msg memory is freed after it has been displayed]. There is no real "work-around" I can offer for this, my only suggestion is that you should write better scriptfiles : hide tasks you don't really need, and don't write "conflicting" scriptfiles, or write just one scriptfile for all the

things you want to see. Of course, when you get right down to it, this is a general problem resulting from the way the priority mechanism works on the amiga.

3 - DON'T use instances of older Snoopy versions together with instances of version 1.8 or higher : older versions had some "severe" bugs in the "instances mechanism" which resulted in strange crashes -> you don't want that, do you.

By the way, did anyone of you know that using CreateProc() with a NULL pointer for the process name causes an enforcer hit ?

1.46 Advanced concepts

DESCRIPTION

Now, since some folks are really using Snoopy (you wouldn't believe it, would you), here are some tips on what you should remember when writing your own scriptfiles

HOW SNOOPY HANDLES ITS SHOW/HIDE MECHANISM

The SHOW/HIDE mechanism gives you nice output but keep in mind that it DOES NOT change the way Snoopy patches library functions, it just changes the way the resulting messages are displayed. This means, that (of course) you CAN patch into all of dos.library's functions at once and SHOW only your own program : but Snoopys port will still receive warnings for EACH message, and it has to PROCESS EACH of these messages, which in this case means A LOT OF messages. The result is that it may take a long time until you see the output for a function call, because Snoopy has to skip the hidden tasks first. So, try to make "tailormade" scriptfiles in that you look out only for stuff you really need to see (like, if you know the only things you need from dos is Open(), Close(), Read() and Write(), why look at stuff like "IsFileSystem()").

PATCHING OF CRITICAL SYSTEM FUNCTIONS

be carefull -VERY carefull- when looking into functions vital to the OS such as exec "AllocMem()/FreeMem()", dos "DoPkt()", or (worse yet) exec "Wait". In these cases you are STRONGLY URGED to use an explicit SHOW statement, because there is currently no way Snoopy can detect an endless-recursion trap ("from hell") that results NOT from system-internal library calls BUT FROM calls from Snoopy itself; see "memory.script" for some example on how this can be a problem and on how you can fix it. Note that "Endless recursion" does not necessarily really means "recursion", it is enough of a problem if e.g. some interrupt calls a system function on

and on, which means trouble (like some internal system functions (especially Workbench) seem to do...probably, I am not sure about this, maybe some real amiga guru can fill me in on some details here). The reason why this means trouble is that once Snoopy gets one message it tries to read as much messages as it can, up to a NULL message, then it falls back to its wait state (exec/Wait()). Also note that I do NOT have any details on whether or not system functions are "priority critical" in that they need to be executed at a certain priority level. I think I did implement everything the save way, but of course you never know. NO, I cannot give you a list of which function you can safely watch and which not, just try and see if it works (really it should work most of the time) but just don't complain if you watch things like "exec/Alert()" and get crap (or less) in return.

HOW TO HANDLE MULTIPLE LIBRARYS FOR ONE SCRIPTFILE

If you use Snoopy for debugging I suggest you write ONE scriptfile for all librarys used in ONE application. As you know you can add a nearly unlimited number of "BASE=" statements; so try to keep everything in place, it prevents you from problems resulting from the multiple instances Link MultipleINSTANCES} problems plus it is (possibly) a little bit faster.

SPECIAL HINT

Note that you can use registers more than once in the output template :

```
...,D0L/D0L,Register Hex $%lx = Decimal %ld
```

is perfectly ok.

1.47 AUTHOR

AUTHOR

Hey! If you use this and you find any bugs or have any ideas, please send me a note! I got very few real "bug reports", which leads me to suspect that you guys probably think stuff thats free doesn't have to be maintained (if Snoopy was shareware I'd probably get more response than now). Funk dat! Send your bug reports, ideas, love letters, fresh'n'phunky vinyl to

Gerson Kurz

Karl Köglspurger Str.7 / A303 (yes,appartment 303,not ROLAND TB303 ;-)
80939 München

West Germany

or call 089/3119421 (human voice only, SUCK EMAIL)

This tool is dedicated to the **ULTRAWORLD** (Destroy Sperrstunde!)

GREETINGS TO

The ULTRAWORLD (munichs BEST rave - 100% pure mind-fucking techno) plus all munich DJs, sound-systems, the food-4-thought-suppliers RECORD STORE, DELIRIUM (I feel it, yeah)!, OPTIMAL, all munich techno ravers, my friends in the industry, RAVE SYSTEM K plus GABBERMAN of DISKO LOVERS INTERNATIONAL, Stefan Nocon, Jumbo of The Wizards, Stefan Scherer plus ACID DEESIGN and all other friends & ravers.

BETA TESTING BY

Christian Rattei and GABBERMAN.

AND ALWAYS REMEMBER

breakbeats! deconstructivism! system riots! real ravers never sleep!

1.48 DISTRIBUTION

** Snoopy is I-WANT-TO-BE-FREE-WARE **

FREELY DISTRIBUTABLE!

see me - feel me - hear me - love me - touch me!

dedicated to the global house & techno scene

written by [Gerson Kurz](#)

LICENSE

Snoopy may be used by and FREELY distributed with any application be it commercial or public domain. There are no pagan users fees, Trump-esque licenses, or other forms of rabid capitalist trickery associated with using this tool and its support files. You do not even have to acknowledge the secret of your superb and efficient whatever-it-is result you gain by using Snoopy. The only limitation is that you may not alter the actual executable of Snoopy, nor sell the product and its support files as a distinct product (i.e. represent it as such). Fred Fish is specifically given permission to include Snoopy in his fine disk collection.

I don't give any guarantee for the fitness of Snoopy for any purpose: Use this software at your own risk. The author will not be liable for any damage arising from the failure of this program to perform as described, or any destruction of other programs or data residing on a system attempting to run the program. While no damaging errors are known, the user of this program uses it at his or her own risk.

1.49 HISTORY

DESCRIPTION

This page tells you about the changes that came with the various release versions of Snoopy

2.0 Major update

- snooping of DEVICES and RESOURCES added!!!
- ALIASes!!
- SnoopyDebug() added!!!
- structure macros
- fixed several bugs concerning multiple instances
- icons now understand a LOT more things
- rewrote everything, which makes Snoopy maintenance much more funky
- rewrote this documentation
- fixed MANY bugs: for instance, 1.7 crashed if Segtracker wasn't installed, any previous version crashed if multiple instances were misused, and so on.. (real shitloads of bugs, I tell you)

1.7 Some additional features

- SegTracker support
- added boolean output ("SUCCESS" or "FAILURE")
- bugfix for INCLUDES and DEFINES! now the current directory will be searched always
- Snoopy now has a "save closing" feature : If "something" else patches system vectors, Snoopy will refuse to close to prevent crashes
- some buffer overflow checking added
- some new scriptfile statements : MATCH, TASKINFO
- removed QUIET option since nobody needs that kind of crap anyway

1.6 Bugfixes

- WARNING! Register Indirect was COMPLETE CRAP! It didn't work a bit, and in those few cases it worked it sure-as-hell crashed! Funk dat! I completely rewrote all the stuff related to this, so now (ax) is enabled ALWAYS (expert mode skipped); thus :
- NO LONGER ANY ENFORCER HITS!!
- INCLUDES and DEFINES and INC_DIRS!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
- added MATCH option
- removed snoopy.doc
- rewrote some of the SUPPORT/ files, added MAKEOFFSETS and "offsets/"

1.5 Bugfixes

- assumed having fixed bug in 1.4 causing 68000 machines to crash, but of

course it still was there... ;-)

- added STICKY option
- added QUIET option
- added special notes on ENFORCER and other MMU tools
- no more Printf() for errors, uses EasyReq() instead (because after detaching the DOS window might be closed)

1.4 Major update

- enables register indirect "due to popular demand"
- workbench interface
- detaches itself automatically
- added snoopy.doc

1.3 internal bugfixes only

1.2 bugfixes

- BASE keyword now can handle a specific library version
- SHOW keyword added
- rewrote this document to fit what one might call "the latest developments in the wide field of the English Language" ;-)
- tested with ENFORCER and found no hits (lucky me!)

1.1 minor update

- added PRI keyword

1.0 First ever released version

(legendary, could be worth a fortune in a few years time when I'm rich and famous ;-)

1.50 The offset/ directory

DESCRIPTION

The directory "offsets/" contains **includefiles** for as many LVO files I could find. Note that these were the most recent includefiles I got: you really shouldn't use V39 functions on your 1.3 kickstart amiga, believe me... They are distributed under the terms of the Snoopy **license**

1.51 The scripts/ directory

DESCRIPTION

The directory "scripts/" contains severel example scriptfiles that explain various details of Snoopy. In the subdirectory scripts/libraries are a lot of edited scriptfiles for common **libraries** , in scripts/resources you will find scriptfiles for all **resources** I could manage to find; and in the

subdirectory scripts/devices the same has been done for **device** scriptfiles. Because of the nature of Snoopys include mechanism, you can directly **include** these if you need to debug a device or a resource and are too bored to make a tailormade script. The main scripts/ subdirectory contains some general-purpose sample scriptfiles

EXAMPLES

snoopdos.script - emulates some of SnoopDos behaviour
snoopdos2.0.script - emulates SnoopDos plus many 2.0/3.0 DOS calls
memory.script - tracks down AllocMem()/FreeMem() [has important notes!]
tooltype.script - lets you find out about tooltypes understood by programs
window.script - track down screens/windows/requesters
trackdisk.script - monitors floopys (exemplary device usage)
miscres.script - examines the misc.resource (exemplary resource usage)
virus.script - well, it just watches SetFunction(), SumLibrary() and some other functions that could be possibly misused by viruses
All scriptfiles are distributed under the terms of the Snoopy **license**

1.52 The support/ directory

DESCRIPTION

The directory "support/" contains several tools you will find usefull in your work with Snoopy (given that you see any sense in doing so ;-). These tools are provided together with their sourcecodes under the terms of the Snoopy **distribution** . They are :

- **MAKEOFFSETS** - build includefiles from assembler includes
- **BUILDWATCH** 1.1 - build watch commands for an FD file
- **TASKLIST,DEVLIST,RESLIST** - show the names of all existing tasks/devices/resources
- **BREAK** - an improved version of the CLI command BREAK that handles not only process but also tasks by name or address

1.53 MAKEOFFSETS

NAME

support/makeoffsets

DESCRIPTION

This tool is used to generate **includefiles** for Snoopy. It does so by analysing assembler LVO-style includefiles [although I probably could add some FD-style option sooner or later]. The command line arguments are **MAKEOFFSETS FILES/A/M,INCPATH/K,INCSUFFIX/K,SKIPPREFIX/K,OUTPATH/K,OUTSUFFIX/K** where <FILES/A/M> are one or more files you want to read. By default,

Snoopy will try to load

```
include:<filename>.i
```

and generate the output in a file called <filename> in the current directory. You can change all those naming conventions by the options, so that names are built like this :

```
<INCPATH><filename><INCSUFFIX>
```

and output files are built as

```
<OUTPATH><filename><OUTSUFFIX>
```

which makes MAKEOFFSET fit for any system settings (I hope). Since most function definitions start with "_LVO", MAKEOFFSETS tries to skip these automatically, so you don't have to write "_LVOOpen" when you could simply write "Open". You can change this using the <SKIPSUFFIX> option.

MAKEOFFSETS is distributed under the terms of the Snoopy [license](#)

EXAMPLE

```
l> makeoffsets dos incpath=include:lvo/ outpath=snoopy:offsets/
```

will read "include:lvo/dos.i" and create the includefile "snoopy:offsets/dos"

SEE ALSO

[The INCLUDE keyword](#) , [The INCDIR keyword](#) , [The DEFINE keyword](#) , [The Offsets/ directory](#)

1.54 BUILDWATCH

NAME

support/buildwatch

DESCRIPTION

In case you have trouble making [script files](#) for your libraries, try this tool. It takes as input the base name for a FD file, that fits a certain name description. Buildwatch is started from the CLI with

```
buildwatch FILES/A/M,FDPREFIX/K,FDSUFFIX/K,RESOURCE/S,DEFONLY/S,USEDEFS/S
```

where <FILES/A/M> are one or more names of FD-files. If you start

BUILDWATCH without any other arguments, it will try to load

```
FD:<filename>_lib.fd
```

and print out all [watch](#) -statements for this file. You can use the

<FDPREFIX/K> option to choose another directory, and <FDSUFFIX/K> to choose

another extension, so that the complete name would be

```
<prefix><filename><suffix>
```

If you set the RESOURCE switch, Snoopy will create output for a resource rather than a library (the only difference is that instead of a "base="

command a "resource=" is issued). The flag USEDEFS can be used to create output that contains the defined names instead of the decimal values (i.e.

"Open" instead of "-30"), and DEFONLY will create those defines (=output a file that contains only "define=" statements). This means, that DEFONLY effectively implements for FD files what MAKEOFFSET does for assembler includefiles.

The results are given to stdout - so you must redirect this if you want to get a script file. You can then hand-edit the result to take only the functions you wish to include. Also note that a script generated by BUILDWATCH is not sensible to the actual data types - BUILDWATCH assumes all data to be LONG registers; so it doesn't recognize STRPTRs for example.

Probably what you would do is : Generate a scriptfile with BUILDWATCH make your own copy of this scriptfile and manually edit it to contain

- a) only those few functions you are interested in (its not very reasonable to Snoop into all of Execs stuff at the same time, for instance...)
- b) proper output formats (e.g. replace %lx with %s for strings and so on)

BUILDWATCH is distributed under the terms of the Snoopy [license](#)

EXAMPLE

```
1> Buildwatch dos fdprefix=sc:fd/ fdsuffix=.fd
```

will try to load "sc:fd/dos.fd" and redirect the output to the current CLI

```
1> Buildwatch misc defonly >offsets/misc
```

will build all defines for the misc.resource

```
1> Buildwatch misc resource usedefs >offsets/resources/misc
```

will create the file offsets/resources/misc for the misc.resource

SEE ALSO

[The Watch keyword](#) , [Script file syntax](#) , [Register Format](#)

HISTORY

1.1 - added RESOURCE/S,DEFONLY/S,USEDEFS/S options

1.0 - first release

1.55 TASKLIST,DEVLIST,RESLIST

NAME

support/tasklist

support/devlist

support/reslist

DESCRIPTION

These are three very small tools that give you a list of the names of all tasks, devices, or resource currently active in your system. If you don't have any other monitor program at hand, you can use any of these to find about the possible names of structures in your system. No fancy options, just start these programs and see what happens

TASKLIST,DEVLIST and RESLIST are distributed under the terms of the Snoopy [license](#)

SEE ALSO

[The SHOW keyword](#) , [The HIDE keyword](#) , [Script file syntax](#) , [The RESOURCE keyword](#) , [The DEVICE keyword](#)

1.56 BREAK

NAME

support/break

DESCRIPTION

This is a small but very usefull improvement to the standard Break command of V37. In addition to the full functionality provided by the original CBM file, this version

- is able to send bit masks
- can specify tasks by name or by address
- can handle multiple tasks / processes

This way you can send break signals to tasks or processes that are not started as CLI commands (i.e. stuff that "C:STATUS" doesn't show) or send breaks by task names instead of silly boring numbers. The Command line syntax remains fully compatible but has some new argument keywords.

break PROCESS/N/M,NAME/K,TASK/K,MASK/K,ALL/S,C/S,D/S,E/S,F/S
where <PROCESS/N/M> specifys one or more process' by a number (gained by 'STATUS'), <NAME/K> specifys a task by name (must be case significant, see [tasklist](#) for tasknames) and <TASK/K> specifys a task by address. Address is given in hex numbering by default, but can be prefixed to decimal with '#', to octal with '@' and to binary with '%'. <MASK/K> specifys a signal mask given in hex numbering, but can be prefixed to decimal with '#', to octal with '@' and to binary with '%'. The <ALL/S> flag is used to set all four break-bits, C/S toggle CTRL-C break-bit, D/S toggles CTRL-D break-bit, E/S toggles CTRL-E break-bit, F/S toggles CTRL-F break-bit. The break-bit options follow this hierarchy :

- set all bits, if flag ALL is set
- if mask is given, OR in those bits
- if the special break-bits are given, TOGGLE these

This means, that you can both set and clear bits in the signal mask in a very easy yet powerfull way.

BREAK is distributed under the terms of the Snoopy [license](#)

NOTE

This BREAK replacement is written as fully reentrant code and thus can be made resident just like the original CBM command.

EXAMPLE

```
break 5
```

sends CTRL+C to the command in CLI #5

```
break name="Snoopy Task.1"
```

will send a break signal to the Snoopy instance #1 (i.e. try to quit it)

SEE ALSO

[support/tasklist](#) , The QUIT keyword, [The DELAY keyword](#)
