User Guide

Whisker

A control system for research

by Rudolf Cardinal & Mike Aitken

www.whiskercontrol.com

Distributed by Campden Instruments Ltd (www.campden-inst.com)

Whisker

Printed: March 2024 in Cambridge, UK

Creator

Rudolf N. Cardinal

Design and Programming

Rudolf N. Cardinal

Michael R. F. Aitken

Legal Advisor (CUTS)

Adjoa D. Tamakloe

Sales (Campden)

Julie Gill

Contacting the authors:

For information about Whisker, visit http://www.whiskercontrol.com/.

If you have sales enquiries about Whisker, contact Campden Instruments Ltd at http://www.campden-inst.com/.

If you have comments or technical enquiries that cannot be answered by the sales team, contact the authors:

Rudolf Cardinal (rudolf@pobox.com)
Mike Aitken (m.aitken@psychol.cam.ac.uk)

Table of Contents

Foreword

WARNING

Whisker is a system designed for research purposes only, and should never be used to control medical apparatus or other devices that could endanger human life.

DISCLAIMER

The authors, copyright holders, and distributors disclaim all responsibility for any adverse effects that may occur as a result of a user disregarding the above warning.

Part I

Overview of Whisker

1 Overview of Whisker

1.1 Introduction

Whisker is a software system designed for controlling digital input/output devices. Its principal function is to control operant chambers for behavioural experiments. Whisker is based on a client–server software architecture: a **server** program is responsible for dealing with the hardware (i.e. provides digital or multimedia I/O **services**), while one or more **clients** can communicate with this server simultaneously to use these services. Each client would typically be implementing a **function,** usually a specific behavioural task.

In a typical research situation using operant chambers, the researcher will start the experimental day by running the WhiskerServer program and checking that all the equipment is working (simple clients can help with this). A client program that implements the required experimental protocol is then run, with the appropriate parameters set. The subject is placed into the relevant operant chamber, and the task started.

While this task is running, the researcher can use the WhiskerServer in any way — usually by running other tasks in other operant chambers — and the WhiskerServer will ensure that nothing influences the task in progress. This means that the common situation in which several subjects are being run simultaneously is no more difficult to program than when only one subject is involved — even if each subject is run under a different protocol: each task is run separately, and no special programming is required to ensure that the tasks can be run simultaneously.

While the tasks are running, the experimenter can monitor the process of the tasks from the WhiskerServer console, or from the client application, or from a remote computer using the status client, or a web browser.

WARNING. Whisker is a system designed for research purposes only, and should never be used to control medical apparatus or other devices that could endanger human life.

DISCLAIMER. The authors, copyright holders, and distributors disclaim all responsibility for any adverse effects that may occur as a result of a user disregarding the above warning.

1.2 Citing Whisker

CITING WHISKER

The reference publication is:

Cardinal RN, Aitken MRF (2010). **Whisker: A client–server high-performance multimedia research control system.** *Behavior Research Methods* 42: 1059–1071. PubMed ID: 21139173. DOI: 10.3758/BRM.42.4.1059.

The web site is:

http://www.whiskercontrol.com

1.3 History, acknowledgements, bibliography

Whisker was first used in the Behavioural and Cognitive Neuroscience Group of the Department of Experimental Psychology, University of Cambridge, UK. It was written in late 1999 because the alternative at that time was expensive and becoming obsolete; the opportunity was taken to design a system that was much more powerful and flexible than any previously available, and much cheaper.

Thanks to the following people:

- Rob Milner sparked the whole thing off by telling me how cheap the hardware was.
- The IVSA group told me what was needed — particularly Maria Pilla, Simon Howes, Dan Hutcheson and Barry Everitt. Maria also served patiently as the guinea pig for the first system.
- Barry Everitt approved the project and provided encouragement throughout.

The following books were useful during this project:

- Stevens WR (1998). *UNIX Network Programming, Volume 1. Networking APIs: Sockets and XTI.* Prentice Hall.
- Kain E (1998). *The MFC Answer Book: Solutions for Effective Visual C++ Applications.* Addison Wesley.
- Prosise J (1999). *Programming Windows with MFC, second edition.* Microsoft Press.
- Cline MP & Lomow GA (1994). *C++ FAQs: Frequently Asked Questions.* Addison Wesley.
- Musser DR & Saini A (1996). *STL Tutorial and Reference Guide.* Addison Wesley.
- Chapman D (1998). *Teach Yourself Visual C++ 6 in 21 Days.* Sams / Macmillan.
- Schwartz RL & Christansen T (1997). *Learning Perl, second edition.* O'Reilly.
- Wall L, Christiansen T & Schwartz RL (1996). *Programming Perl, second edition.* O'Reilly.
- Cohen A & Woodring M (1998). *Win32 Multithreaded Programming*. O'Reilly. (Includes the MCL and MCL4MFC multithreading libraries.)
- Mel HX & Baker D (2001). *Cryptography Decrypted*. Addison-Wesley.
- Howard M & LeBlanc D (2002). *Writing Secure Code*. Microsoft Press.
- Shamir A & van Someren N (1998). Playing hide and seek with stored keys. (Paper available from www.ncipher.com under White Papers; keyhide2.pdf.)

The following software was used:

- Microsoft Visual C++ 6.0 Professional (*http://msdn.microsoft.com/visualc*), and subsequently Microsoft Visual Studio 2008 Professional
- Inno Setup, by Jordan Russell (*http://www.jrsoftware.org/*); this replaced InstallShield.
- Microsoft Word 97 (*http://www.microsoft.com/office*)
- Adobe Acrobat 4.0 (*http://www.adobe.com/*)
- Adobe Illustrator 8.0 and later (*http://www.adobe.com/*)
- TextPad 3.2.5 and later (*http://www.textpad.com/*)
- Perl 5 and later for Win32 (*http://www.ActiveState.com/*)
- Microsoft Visual Basic 5.0 and 6.0 Professional (*http://msdn.microsoft.com/vbasic*)
- Paint Shop Pro 6.02 and later (*http://www.jasc.com/*)
- Help & Manual 3.1 and later (*http://www.helpandmanual.com/*)
- Araxis Merge 2001 v6.0 Professional (*http://www.araxis.com/*)
- CS-RCS Pro revision control software (*http://www.componentsoftware.com/*), subsequently replaced by Subversion (*http://subversion.tigris.org/*)
- The Crypto++ cryptographic library for C++ (*http://www.cryptopp.com/*)
- For video, DirectShow filter concepts and libraries including adaptations of GMFBridge (*http://www.gdcl.co.uk/*) and DirectShow Audio Transform (*http://www.chrisnet.net/code*).

Part II

TROUBLESHOOTING - FAQ

2 TROUBLESHOOTING - FAQ

TOUCHSCREENS AND DISPLAYS

Q. My touchscreen doesn't work.

For the touchscreen to work, it must communicate with UPDD. Make sure you have installed it as described in the instructions for installing touchscreens.

- Start the UPDD Settings program.
- Choose the "Status" view.
- Re-initialize the touchscreen. [Note: if you have multiple touchscreens, you have to switch to another panel of the UPDD Settings program, choose the correct touchscreen, switch back to the Status panel, and click "Re-initialize"; otherwise, it re-initializes the wrong touchscreen.]
  - If the display eventually says "OK", UPDD has basic communication established; proceed.
  - If not, either (1) the touchscreen has the wrong DIP switch settings; (2) Windows has the wrong COM port settings; (3) UPDD has the wrong COM port settings. Fix and repeat.
- Touch it a few times. If *Sync errors* appear, the touchscreen has incorrect DIP switch settings; fix these (with the touchscreen power cable disconnected) and re-initialize.
- Calibrate the touchscreen using the UPDD calibrator.
- Enable it in WhiskerServer (Configure hardware → Touchscreens).
- Try it on a Whisker test pattern.

Q. My touchscreen works - sort of. When I tell WhiskerServer to display a test pattern, it responds to my touches, but it interferes with my mouse.

Whisker isn't using the touchscreen; the mouse movement is coming from the UPDD driver. Choose the menu option Configure hardware → Touchscreens. Make sure the touchscreen is enabled and attached to the correct monitor. Restart the server.

This problem may arise if you re-install a touchscreen, as UPDD gives the re-installed touchscreen a new name.

Q. My touchscreen response appears abnormally prolonged.

Symptom: If I make a very short screen touch, while watching the server's debugging display, the touch-detector stays lit for longer than the screen is touched.

Problem: UPDD, the software that is an intermediary between Whisker and the touchscreens, is misconfigured. This problem arises if the "lift-off time" is set too high. In old versions of UPDD (version 2), a lift-off time of "40" meant 40 ms. In newer versions (version 3), a lift-off time of "40" means 40 x 20 ms = 800 ms (as lift-off is defined in units of 20 ms). Try reducing this value to 2 in the UPDD settings.

DIGITAL INPUTS AND OUTPUTS

Q. All digital input lines appear to be on

The computer may not be connected to the hardware properly. This is especially likely if many digital input lines are flickering on/off together. Check all cables. (Sometimes it can be difficult to plug in several Amplicon 78-way cables, if your computer's case gets in the way. We have once solved this problem by paring away some of the plastic with a sharp knife, but don't pare too

deeply – you might cut a cable!)

Q. Digital I/O devices behave... strangely

For example,

- Digital inputs are not always 'on' or 'off' but the optoisolator LEDs attached to input lines sometimes fade in and out.
- Levers get stuck halfway, or move on their own, or click slightly.
- Relays click spontaneously as they 'float'
- When you turn the power supply on after it's been off for a while, optoisolator LEDs light up (as stray capacitance discharges).

The devices are improperly grounded. There is almost certainly a loose or unconnected wire in the ground circuit of the digital I/O boards or of the operant chamber devices. Check all colour coding of wires!

Q. All devices switch on when the computer is reset.

This is a known problem with Amplicon digital I/O boards. Install a safety relay system.

Q. Components on an Amplicon EX230 input panel or EX221 mixed panel start smoking

If you haven't turned the power off, do it this instant. You have connected a +24V supply to the A/B/C 'VCC' line in block SK6. The optoisolator chip was hoping for 5V. The optoisolator chip has been destroyed. Oops. Our condolences; we've done it too. When you replace the blown Wickman fast-blow 19370K1A fuse (1.0 A) on the EX233 distributor board, you may find out that all the optoisolator lights are on. Buy another optoisolator board.

Q. All the optoisolator lights are on

Did you plug a 24V line into a VCC line connected to an optoisolator chip? You've destroyed the optoisolator chip. See above.

Q. I've installed a failsafe relay, but I can't work out how to tell the server about it

Choose the menu option Configure hardware → Configure failsafe outputs.

TESTING TASKS ON COMPUTERS WITHOUT FULL HARDWARE

Q. I want to run a task on my office computer, but it doesn't have an operant chamber/ touchscreen/second monitor... When I try to run my task, it complains that it can't claim a line/display/audio device...

For example, suppose you install Whisker on your office computer, and then install and run MonkeyCantab (one of our tasks). This task needs digital input/output (I/O) lines, to control pellet dispensers etc., and a touchscreen and audio device. Suppose it gives an error like this:

> Failed to claim display screen, unable to claim lines. Cannot start task.
> Error message: "Claim refused:box0:screen not recognized" received!

This indicates that there is no display device known as "box0:screen". Why?

A1. Ensure that your device definition file is set up correctly.

Have you actually told Whisker that you want to call your display "screen" in a device group called "box0"? This is the job of the device definition file. You need to create an appropriate definition file (for which, see the manual for your task) and tell the server to use that definition file.

A2. Ensure that you have appropriate digital I/O lines, creating "fake" ones if necessary.

If your task needs a lever and a pellet dispenser, your device definition file has been created to refer to one, and yet you don't have any such physical devices attached to your computer - and not even a digital I/O card plugged into the computer that would communicate one - then you need to tell the server to create "fake" (debugging) lines. Once the server has created fake lines, ensure your device definition file refers to the line numbers that correspond to your fake lines (looking at the server's line status display will allow you to see the line numbers of your fake lines).

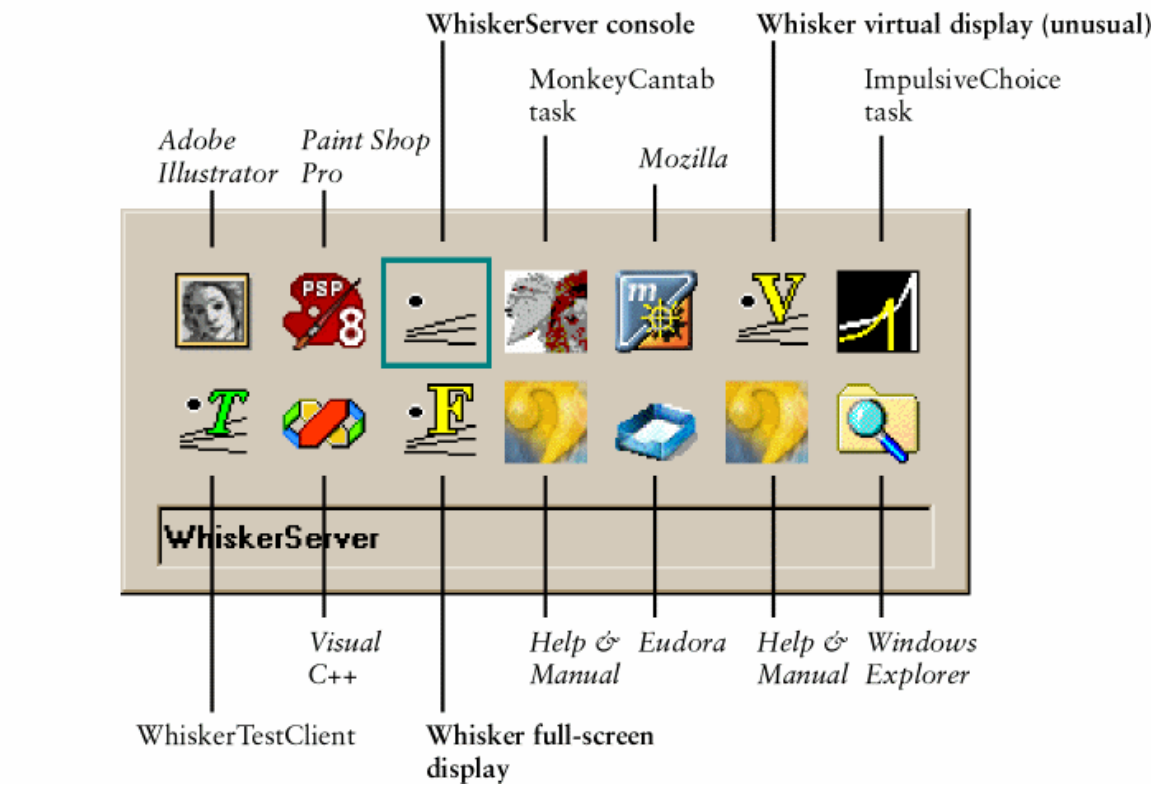
A3. Ensure that you have enabled your audio card.

Your task may require an audio device, in which case your device definition file should refer to it. But you also need to tell Whisker which audio cards to enable.

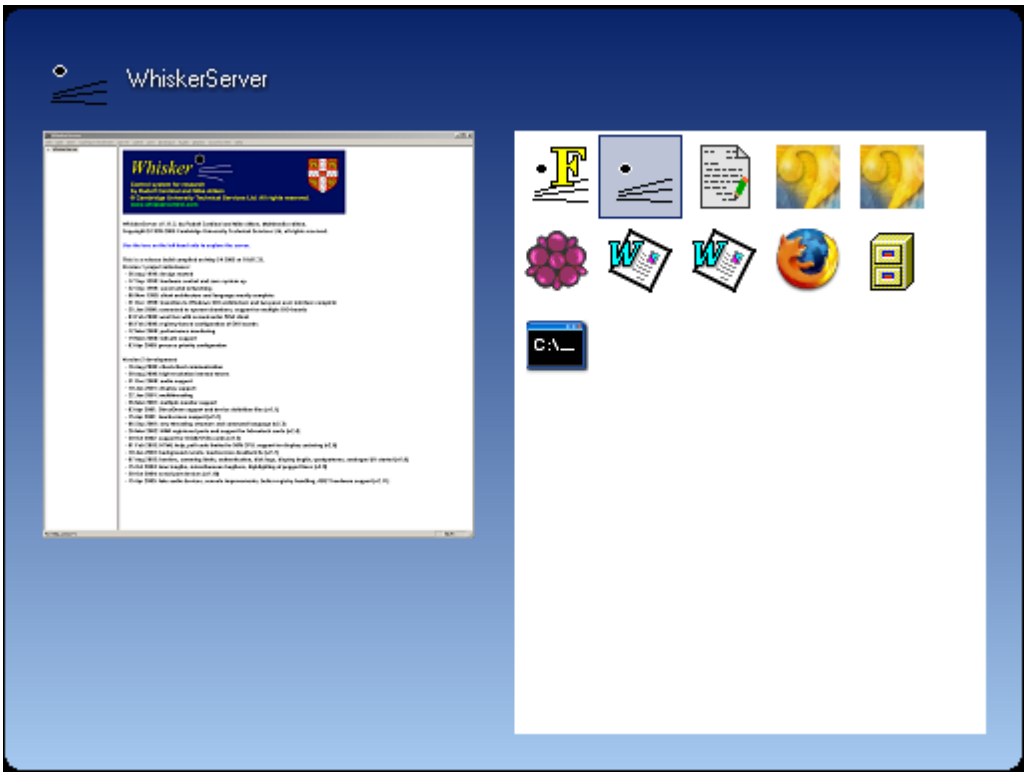
A4. Ensure that you have enabled your display device.

If your task needs a display device, you must tell Whisker to use a display device. If your office computer doesn't have a second monitor, you will need to use your primary display as the experimental monitor. In this case, when you restart the server, you'll see a large black window. Use **Alt-Tab** to switch between windows in case you can't see your Windows taskbar. You may see several Whisker-related icons. Here's an annotated real example, with some other programs also running from Windows 2000:

Example of pressing Alt-Tab to switch windows while running Whisker



Here's a similar one in Windows XP (which shows a preview of the application you can switch to as well). Here there is a Full screen display window, and the console window:



A5. Ensure that you have enabled your touchscreen.

If you have a touchscreen, Whisker needs to be told which touchscreen is attached to which display. If you don't have a touchscreen, then you can simply use the server's console copy of a display and enable mouse input, mimicking a touchscreen.

Part III

Obtaining Whisker

3 Obtaining Whisker

3.1 Editions of Whisker

Whisker is available in several editions. A synopsis of the differences between the editions is shown here:

Feature	Basic Edition	Multimedia (Full) Edition	Programmer's Edition	Embedded Edition for CANTAB
Full TCP/IP communications (supports any client)	√	√	√	× (CANTAB only)
Control of 'fake' (virtual) digital I/O lines for programming	√	√	√	×
Control of physical digital I/O lines via interface cards	√	√	×	√ (but for CANTAB only)
Graphical display and multimonitor support	×	√	√	√ (but for CANTAB only)
Touchscreen support (via UPDD driver)	×	√	×	√ (but for CANTAB only)
Touchscreen emulation using mouse	×	√	√	√
Mouse and keyboard input	×	√	√	√ (but for CANTAB only)
Audio output	×	√	√	√ (but for CANTAB only)
Digitally-signed data logs	√	√	√	√
Video	×	√	√	√ (but for CANTAB only)

See also

- Ordering Whisker

3.2 Ordering Whisker

Whisker is owned by Cambridge University Technical Services Ltd.
It is sold under licence by

Campden Instruments Ltd
http://www.campden-inst.com/

See also

- Editions of Whisker

Part IV

Installing Whisker software

4 Installing Whisker software

4.1 Software requirements

You will need:

- **Whisker**
- **A compatible Windows operating system.** WhiskerServer runs under Windows 2000, XP, and Vista. WhiskerServer Standard Edition will also work under Windows NT 4.0 (with Service Pack 3), but Windows NT 4 does not support multiple monitors and will therefore not run the Multimedia Edition of Whisker fully. Windows 95/98/ME is *not* suitable for running WhiskerServer. If you plan to use Vista, see also Notes on running Whisker under Windows Vista.
- **TCP/IP** installed. This is a network protocol that comes with Windows. TCP/IP needs to be installed to the point that 'ping 127.0.0.1' succeeds.
- **Driver software for the digital I/O cards.** These cards come with driver software, which must be installed separately.
- **Driver software for any other installed cards,** such as multimonitor cards or multi-way serial port cards.
- **Touch-Base UPDD software for touchscreens.** Touchscreen support is provided by means of the Touch-Base UPDD, which must also be installed.

To take full advantage of Whisker (i.e. to write your own tasks), you will need a programming language that supports TCP/IP communications. This is discussed in greater detail later in this Guide, but **Visual C++** and **Visual Basic** are two useful programming languages.

4.2 Software installation

If you purchased WhiskerServer with hardware, this will all have been done for you. However, this section be useful if you need to reinstall the software for any reason, or if you are installing Whisker from scratch yourself.

Install Windows NT/2000/XP/Vista

If you don't know how to install this, find an expert. You will want to install Windows NT Service Pack 3 or greater (5 or greater for large hard disks) if you use NT4.
If you plan to use Windows Vista, see Notes on running Whisker under Windows Vista.

Tip

User accounts. When you install Windows NT, it creates a user account called Administrator, and you set the password for this account. Never lose this password. It is also a good idea to add a new user (e.g. "Rudolf") and subsequently do all your work in this user, so that you always have the Administrator account to fall back on. You may or may not want to give the new user the power and responsibility of an administrative account. Choose *Start → Programs → Administrative Tools (Common) → User Manager* to add users.

You should log in to Windows NT as an **Administrator** for the installation of the rest of the software — administrator authority is required to install TCP/IP, the Amplicon drivers and ODBC.

You must install and configure **TCP/IP** (found under *Control Panel → Network → Protocols*), even if you have not installed a network card, because Whisker uses TCP to communicate.

Tips

Rescue Disk. Whenever you make major changes to your computer's configuration, update the Rescue Disk once you are sure the new changes worked and the system is stable. Run `rdisk.exe` and choose *Update Repair Info*, then *Create Rescue Disk*.

Regional settings. After installation, choose *Control Panel* → *Regional Settings* and configure your computer for the country you live in. It makes dates more consistent.

Command prompt here. If you would like to be able to right-click a folder and bring up a command prompt already in that directory, run Windows NT Explorer and choose *View* → *Options* → *File Types*. Choose "Folder", and click *Edit* → *New*. In the *Action* box, fill in "Command prompt here". In the *Application used to perform action* box, fill in

```
C:\WINNT\system32\cmd.exe /k cd "%1"
```

or a different directory if you have installed Windows NT elsewhere. Click OK.

Install digital I/O drivers

Please see the CDs supplied with your cards for details. Installation may be slightly complex. At the time of writing, installation is different for ISA and PCI versions of the Amplicon cards, while Advantech cards require installation of the drivers *before* installation of the cards.

Install multimedia drivers (monitors, serial cards, sound cards, touchscreens)

Windows drivers for monitors, serial cards and sound cards should be installed according to the manufacturer's instructions. If you are using touchscreens, UPDD must be installed, and touchscreens configured and calibrated according to the UPDD manual.

All devices should be enabled for use in Windows (for monitors, all monitors should be enabled, and the desktop extended onto these monitors).

Install a database with ODBC support

This is optional. However, many commercial Whisker clients use the ODBC protocol to store data in a relational database (RDBMS). They will work without this facility, but it dramatically improves a lab's productivity. Microsoft Access 97 is one example of such a RDBMS. Windows 2000 comes with ODBC, but you will still need some form of database.

Install programming languages

Follow the instructions that accompany each product.

Install Whisker

Insert the Whisker CD and double-click the installation program for your edition (e.g. Basic, Multimedia). It will be named something like **Whisker_Multimedia_Setup.exe.** Follow the on-screen instructions.

Configure Whisker for your hardware

Once Whisker is installed, you can start the server by choosing *Start* → *Whisker* →

WhiskerServer. From the main menu, choose *Configure Hardware* and proceed according to the instructions in the rest of this guide.

Technical Note: WhiskerReset startup behaviour

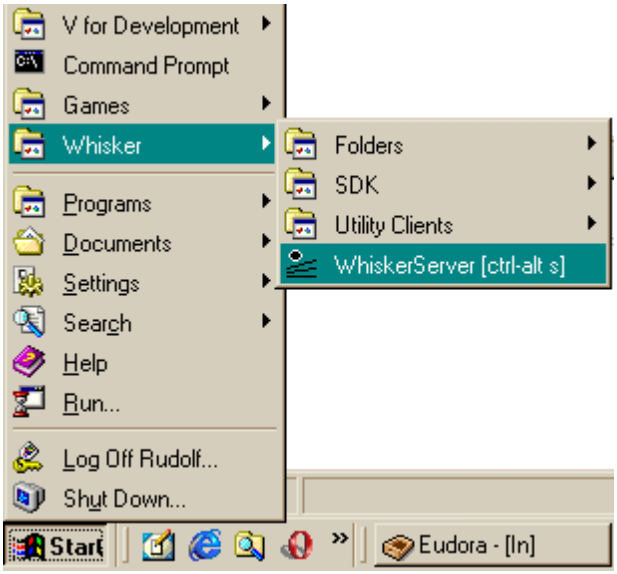
When Whisker is installed, it registers the WhiskerReset utility to be run when Windows starts. By default, this ensures that all devices are turned off (high, or 0V, on a conventional system) as soon as a user logs in to Windows. If you wish to modify this behaviour, you must edit the system registry. This should only be done by an experienced user. Choose *Start* → *Run* → *REGEDIT*. Edit the key as follows:

Key: HKEYLOCALMACHINE
SubKey: SOFTWARE\Microsoft\Windows\CurrentVersion\Run\
Value: WhiskerResetUtility
Type: REG_SZ
Data: Path to the WhiskerReset.Exe file.

Remove the value to prevent any reset. Append the option "ReverseOutputs" or "On" to reverse the behaviour at startup, i.e. to switch the lines all on ('low', or -24V, on conventional systems).

4.3 Overview of installed software

When Whisker has been installed, it appears here (*Start* → *Whisker*):



The Whisker suite includes the following software:

- **WhiskerServer,** the server program.
- **On-line documentation,** including this User Guide and a ReadMe file with reminders on installation.
- **WhiskerStatus,** a client used to find out the status of your subjects from another computer. Source code is supplied for this program.
- **WhiskerReset,** a small command-line utility to reset all your devices.

The following clients (behavioural tasks) are supplied, all with full source code:

- **Second-order schedules of reinforcement.** This package includes (1) the SecondOrder task, written in C++; (2) a relational database for automated data storage, which requires Microsoft Access 97 (part of Microsoft Office 97 Professional); (3) a spreadsheet for displaying cumulative record graphs, which requires Microsoft Excel 97.
- A simple **command-line C++ client,** as an example for you to extend.
- A simple **Visual Basic client,** as an example.
- A **library** for use in writing C++ clients.

The following third-party software is included:

- **Adobe Acrobat Reader 4.05.** Select the menu option to install Acrobat Reader. This is required to read the on-line version of the User Guide (this document).
- **Xitami** web server (optional)
- Library files required to run Whisker software.

4.4 Uninstalling Whisker

If you ever want to uninstall Whisker, click *Start* → *Settings* → *Control Panel* → *Add/Remove Programs*. Choose Whisker, and click *Add/Remove*.

4.5 Upgrading to a new version of Whisker

Log in as a user with administrator privileges.

Uninstall Whisker (*Control Panel* → *Add/Remove Programs*, etc.)

Install the new version of Whisker.

Note: some future upgrade versions may be installed without uninstalling previous versions. Please refer to any documentation that accompanies the upgrade.

4.6 Windows Vista and Whisker

Special considerations apply to Windows Vista, as follows.

INSTALLATION

Ensure you run the installer as an administrator (or it will be unable to intsall to \Program Files).

ALLOWING WHISKER THROUGH THE WINDOWS FIREWALL

When you run WhiskerServer, for the first time, you will see the following message:

```
Windows Security Alert
Windows Firewall has blocked some features of this program
---------------
Windows Firewall has blocked this program from accepting
```

```
incoming network connections. If you
     unblock this program, it will be unblocked on all private
networks that you connect to. What are
     the risks of unblocking a program?
     Name:                WhiskerServer
     Publisher:           Unknown
     Path:                C:\Program
Files\WhiskerControl\WhiskerServer.exe
     Network location:    Private network
```

Choose "Unblock" (rather than "Keep blocking").

If you get this wrong the first time, you can redo this using:
Start > Settings > Control Panel > Windows Firewall > Allow a program through Windows Firewall.
You can then choose either "Add program" (and select whiskerserver.exe, usually in C:\Program Files\WhiskerControl) or "Add port" (and choose port 3233 of type TCP, giving it an appropriate name such as "Whisker main port").

Explanation: WhiskerServer is a server program that must accept network connections to it, on port 3233 and other ephemeral ports. You do not have to let other computers have access to it, but you must at least allow clients on the same computer to have access.

CHANGES TO DEFAULT NETWORK BEHAVIOUR: INTERNET PROTOCOL VERSION 6

Note that Windows Vista by default accepts "localhost" to mean "this computer" for network connections. The old "loopback" is not accepted. Furthermore, Windows Vista runs IPv6. By default, "localhost" refers to the IPv6 local address of ::1, rather than the older IPv4 local address of 127.0.0.1. (This behaviour is all configurable by editing **C:\Windows\System32\Drivers\etc\hosts** if you wish to do so.)

As of January 2009, Whisker clients supplied via www.whiskercontrol.com default to "localhost". If you are using an older client, simply enter "**localhost**" or "**127.0.0.1**" instead of the old "loopback".

FEATURES OF WHISKERSERVER THAT REQUIRE IT TO RUN WITH ADMINISTRATOR PRIVILEGES

If you do not run WhiskerServer.exe as an administrator, Windows refuses to let it run as a real-time process; Windows silently fails (i.e. doesn't tell WhiskerServer that it won't allow this) and changes the real-time request to "High". Furthermore, it is unable to write to the hardware configuration section of the registry (see Whisker's use of the registry). **Therefore, WhiskerServer should be run with administrator privileges.**

To configure WhiskerServer to run as an administrator, this is the easiest way:

- left-click Start > Whisker
- right-click WhiskerServer
- left-click Properties > Advanced > tick "Run as administrator" > OK > OK

Windows will say:

```
Access denied
```

```
     You will need to provide administrator privileges to
changes these settings.
     Click continue to complete this operation
```

Click "Continue". Windows will say:

User Account Control
Windows needs your permission to continue
If you started the action, continue.
Save Shortcut Properties
Microsoft Windows

Click "Continue".

Now the usual WhiskerServer menu shortcut (and the Ctrl-Alt-S keyboard shortcut) will run WhiskerServer as an administrator. However, whenever you do this, a message will pop up saying:

```
     User Account Control
     An unidentified program wants access to your computer
     Don't run the program unless you know where it's from or
you've used it before.
     WhiskerServer.exe
     Unidentified Publisher
     > Cancel - I don't know where the program is from or what
it's for.
     > Allow - I trust the program. I know where it's from or
I've used it before.
```

Click "Allow" or press Alt-A.

So your quickest shortcut to starting WhiskerServer is now **Ctrl-Alt-S, Alt-A.**

Part V

HARDWARE installation

5 HARDWARE installation

5.1 Introduction

This section of the guide covers

(1) The hardware required for Whisker
(2) How to install the hardware devices that Whisker will recognize;
(3) How to install Whisker.
(4) How to configure Whisker to use hardware

This is intended to be a guide to help with setting up the hardware and software you need for a WhiskerControl system. It should be regarded as an addition to, rather than an a replacement for, the manuals which will be provided with the individual products. You must read any manuals and/ or readme files which come with the products, as information may become outdated very rapidly.

Although this guide is intended to help as much as possible, those without any experience in either using or administrating Windows NT systems or wiring electrical equipment would be advised to seek expert help.

5.2 Hardware requirements

Computer

You will need a PC-compatible computer capable of running Windows NT or 2000. You will need enough ISA or PCI slots to install your digital I/O cards (see below). For additional functions in Whisker Multimedia Edition, you may also need extra serial ports, or space for extra serial cards (for touchscreen input), sound cards (for sound output) and a multi-monitor card (for dedicated experimental displays).

We first used Whisker on an AMD K6-2/450 system with 128 Mb RAM and Windows NT4 (costing about £510 in September 1999). That was more than fast enough for the digital I/O features of Whisker. Multimedia users may benefit from faster computers, however, as this improves drawing speed.

Uninterruptible power supply

An uninterruptible power supply (UPS) is important to protect your system against mains power failure, especially if you are using potentially dangerous devices such as intravenous infusion pumps; in this situation, we consider a UPS essential. Using an UPS is also good practice because it prevents data loss.

Digital input/output devices

Digital I/O cards currently supported are

- Amplicon 272 series (72-line digital IO)
- Advantech 1753 cards (96-line PCI digital IO)
- BNC Controller (Campden Instruments Ltd)
- ICS / Advent
- National Instruments (selected models)

- Serial ports
- Lafayette CANTAB USB

Multiple monitors
(Multimedia Edition only)

If you wish to have more than one video output, then extra monitor cards will be required. **Multiple monitor cards** are provided by a number of manufacturers, such as Matrox.

Touchscreens
(Multimedia Edition only)

Touchscreen compatibility is provided through the Universal Pointing Device Driver by Touch-base Ltd. Any touchscreen supported by the UPDD is supported by Whisker.

For each touchscreen, a serial port is required. **Multiple serial cards** are available from a variety of sources.

Sound devices
(Multimedia Edition only)

Sound is supplied by using standard PC sound cards. Each card can be used as a single stereo sound outputs, or a pair of mono outputs.

5.3 Notes regarding large systems

A single digital I/O card will provide a limited number of lines: typically 96 (for Advantech cards) or 72 (for Amplicon cards).

If you need more lines than this, you will need more than one IO card. PC-compatible computers have a limited number of ISA and PCI slots: you will need some slots for sound, serial or multimonitor cards.

Even if you have enough lines in total, do you have enough inputs and enough outputs?

Input/output mixes on Amplicon cards

With 24V devices, which we use, lines are configured in blocks of 24. Each block of 24 may be all inputs, all outputs, or 16 inputs and 8 outputs. So each 72-line card can take the following input/ output mixes: 72/0, 64/8, 56/16, 48/24, 40/32, 32/40, 24/48, 16/56 (the only combination not possible is 8/64).

Input/output mixes on Advantech cards

Lines are configured in blocks of 8, to be input or output.

Very large operant control systems

Our intravenous self-administration boxes have up to 6 inputs and 8 outputs each. So a full 8-card system should be able to run 41 of these boxes, and if you have fewer devices attached, you'll get more boxes. So you could build systems that are very large by common behavioural neuroscience

standards.

For a system this large, it might be a bit cheaper to get an 'industrial' computer (1 @ <£1,500?) than the necessary number of smaller-capacity PCs (4–5 @ £500 each). On the other hand, I generally prefer to have lots of cheap PCs scattered around running smaller systems – if one system is damaged you lose less; also, you have more computers hanging around to do other things with.

Whether you can actually use up the software limit of 8 cards depends on the other devices in the computer. Each card will need a unique *I/O base address*, and very probably a unique *interrupt*. Base addresses and interrupts are also used by things like serial ports, Ethernet cards and so on. But if you use a PS/2 mouse and keyboard, disable the serial (COM) ports, use a networked printer and disable the printer (LPT/PRN) port, use PCI or on-board video, a PCI network card, and as many as possible of your PCI devices share interrupts, you should certainly manage 5 or 6 cards in an industrial computer (360 or 432 lines). Note that multimedia Whisker setups will often need to use PCI slots and interrupts for Audio outputs (one PCI slot for each 2 channel sound card), Display outputs (at least one slot for a multimonitor adapter), and Touchscreen inputs (one slot for a multi-serial card).

5.4 DANGER - safety with critical devices

We have discovered one potentially life-threatening eventuality. This relates to the use of devices that are potentially fatal if switched on inappropriately – particularly intravenous infusion pumps.

At the core of the Whisker system is a set of digital I/O boards that control your devices, and a server program that controls the I/O lines. Whenever you start or stop the server, it makes sure that all devices are off. While the server is running, it does its best to ensure that devices are never left switched on by mistake, using a number of safety features.

So far, so good. But **when the computer is first turned on (or hard-reset), output relays on Amplicon cards are switched ON** (see the *Technical Note* overleaf for an explanation). As soon as you run the Whisker server for the first time, everything is OK. But consider the situation: your tasks are running, there is a power cut; everything switches off. The power is restored; the computer powers up; all the devices are switched on, including i.v. pumps; before the computer finishes booting the animals are dead of an overdose.

We'd thought this through, and our computers had ATX motherboards that did *not* switch on when power is applied; they need someone to press the ON button. We though we were OK, but we had never tested the effects of very brief (<0.5 s) power cuts; this *did* cause the computer to reset, and all devices went on.

The solution we have adopted is threefold:

1. **Wire the power to the critical devices through a fail-safe devices.** Whisker supports and is aware of fail-safe devices. This removes the danger from power cuts.
2. **Install uninterruptible power supplies for the control computers.** This prevents the nuisance and data loss caused by power cuts, and removes the danger if somebody has forgotten to install a fail-safe device.
3. **Never turn on or hard-reset the control computer with subjects in the operant chambers and the operant chambers powered on.** Run the server software at least once

first. (You can turn the computer on if you switch the box power supply off until the server is loaded, or you can take the rats out until the server is loaded, or whatever.) This prevents subjects being confused by devices switching on and off, and removes the danger if somebody has forgotten to install a fail-safe device.

Whisker is a system designed for research purposes only, and should never be used to control medical apparatus or other devices that could endanger human life.

See also:

- Fail-safe devices
- Technical note: why does this happen?

5.5 Fail-safe devices

Every system that has a device that could be dangerous if accidentally switched on or off should have a fail-safe device installed.

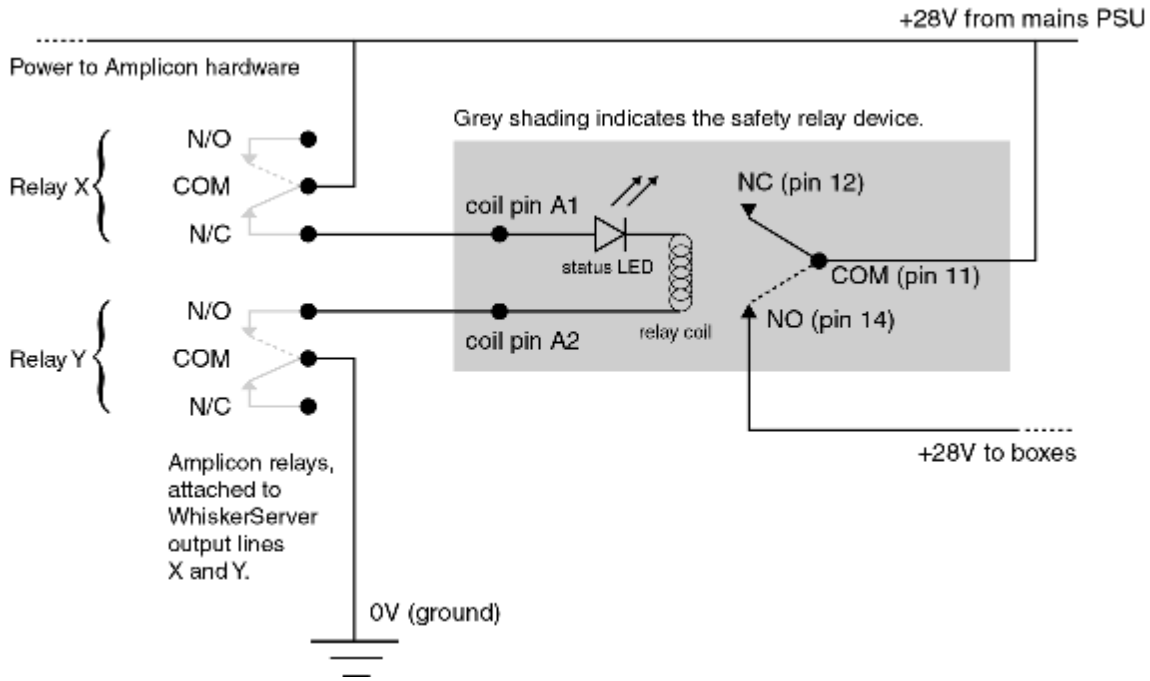
Note also that Whisker is a system designed for research purposes only, and should never be used to control medical apparatus or other devices that could endanger human life.

Parts required

- 24V 10A 3.5mm PCB power relay, RS part no. 229–3614, £2.75 each
- DIN rail mounting socket for 3.5mm PCB relay, RS part no. 400–4129, £3.07 each
- LED power monitor module for mounting socket, RS part no. 400–4056, £2.59 each

Wiring the system

Designate two dedicated outputs on your system for fail-safe outputs. Call them X and Y. Wire them as follows:



(Note: this high-power relay will not run from the 5V TTL outputs on the Amplicon EX233 distribution board.)

Controlling the power

- Wire the main box PSU to the COMmon pin of relay X.
- Wire the N/C (normally closed) pin of relay Y to one of the safety relay coil pins.
- Wire the other safety relay coil pin to the N/O (normally open) pin of relay Y.
- Wire the COMmon pin of relay Y to ground.

The N/O pin of relay X isn't used and mustn't be connected to anything.
The N/C pin of relay Y isn't used and mustn't be connected to anything.
Thus, when relay X is off and relay Y is on, current flows to the safety relay's coil.

Power to the devices

- Wire the main box PSU to the COMmon pin of the safety relay.
- Wire the N/O (normally open) pin of the safety relay to your boxes' 24V power line.

Thus, the boxes only get power when X is off and Y is on.

Configure WhiskerServer

The WhiskerServer program should be configured (Configure hardware → Configure failsafe outputs) so that

- the output line attached to relay X is OFF during operation;
- the output line attached to relay Y is ON during operation;
- some other state pertains when the server is not running (e.g. both relays off).

See also

- Danger with critical devices

5.6 Amplicon digital I/O hardware

Whisker supports I/O cards by Amplicon Liveline Ltd; http://www.amplicon.co.uk/.

This section is designed to help you to choose and install Amplicon I/O hardware for Whisker.

5.6.1 Buying Amplicon digital I/O hardware

The hardware comes from Amplicon Liveline Ltd (http://www.amplicon.co.uk; (telephone 0800–525–335, technical support 01273–608–331, next-day delivery is usually available).

Prices shown here may be out of date. Last checked August 2001.

For each 72 lines, you will need one I/O card (this can be PCI or ISA) and one distribution board. For high-voltage (>5V) devices, you will need up to three I/O panels, each with 24 lines. The vital components are

- **PCI272 PCI digital I/O board.** Interfaces the computer with up to 72 input/output lines. Part no. 960–035–23. £149 each.
- **PC272E ISA digital I/O board.** Interfaces the computer with up to 72 input/output lines. Part no. 909–562–33. £99 each.
- **78-way cable.** Connects one I/O card (PCI or ISA) to a distribution board. Part no. 909–663–49. £61 each.
- **EX233 distribution board.** A bridge between the computer and the high-voltage (24V) input/output panels. Can also accept 5V inputs and outputs directly. Part no. 909–663–33. £99 each.

If your inputs and outputs all use TTL voltages (5 V), that's all you need. If you want other voltage systems, as we usually do, you may connect up to three 24-line panels to each distribution board. Amplicon supply a 24-way output panel, a 24-way input panel and a mixed 16 input/8 output panel, which you may install in any combination. They are:

- **EX213 output panel.** Provides 24 outputs of several kinds (see below). Part no. 909-663-63. £199 each. *The manufacturer's worst-case switching speeds for these outputs are 8 ms to close the relay (of which 1 ms to turn on the logic circuit; 7 ms to close the relay itself) and 7 ms to open the relay (of which 5 ms is to turn off the logic circuit and 2 ms is to open the relay).*
- **EX230 input panel.** Provides 24 inputs of several kinds (see below). Part no. 909-663-73. £129 each.
- **EX221 mixed panel.** A combination of the above, with 16 inputs and 8 outputs. Part no. 909-663-83. £149 each.
- **37-way ribbon cable.** Connects each input/output panel to the EX233 distribution board. Part no. 908-920-05. £20 each.

You will probably also need:

- **A DC power supply.** The Amplicon output panels need a 24V DC power supply. (The

distribution board and input panels can draw power from the computer, and this is recommended.) You may already have a DC power supply for your operant chambers; if this is 24V and sufficiently powerful, it can drive the Amplicon output panels too.

- **DIN rail mounting.** Amplicon cards will clip onto DIN rail, which you can attach to something convenient, like a wall. **Deep top hat DIN rail (35 mm wide, 15 mm deep), punched,** from RS Components (http://rswww.com) is part no. 176-703 and costs £3.24 per metre.

5.6.2 Installing Amplicon digital I/O hardware and drivers

Turn everything off first! Never try to add/remove an expansion card to/from a computer that is on. If you wave 24V wires around, you may damage components by short-circuiting them accidentally.

Please note that I may confuse +24V and +28V in this discussion - this is because the precise voltage isn't terribly important. As long as everything's the same, you can use either. (Note for those using Med Associates power supplies: Med Associates persist in referring to +28V and -28V lines, giving the impression that there's a 56V potential in total. This is nonsense; what they refer to as -28V, or sometimes as "28V ground", is 0V.)

Proceed through the following steps:

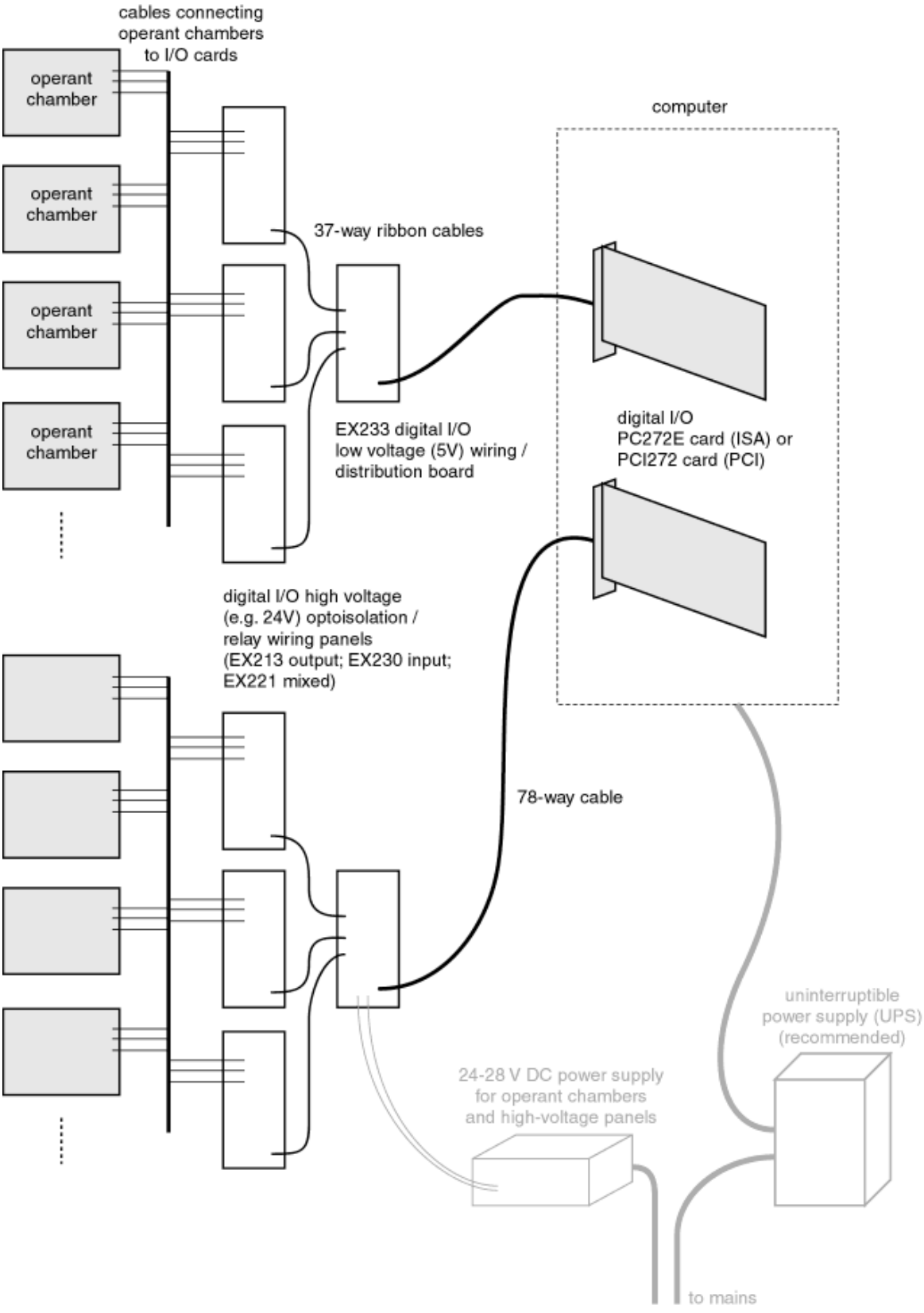
1. Configure digital I/O cards and install them into the computer.
2. Configure EX233 distributor boards.
3. Configure EX213 output panels.
4. Configure EX230 input panels.
5. Configure EX221 mixed panels.
6. Install a safety relay device.
7. Connect the components together; connect your devices; connect power.

You may need...

- Wire of two colours (e.g. red for +28V, black for 0V). Ensure that the wire will take enough current; I use 3A multicored wire (multiple strands of copper rather than a single core make the wire stronger and more flexible; 3A is easily more current that is needed and 3A copper wire isn't too thick to manipulate easily).
- Wire cutters/strippers
- Something hard and thin (like a small screwdriver) for opening up the lever-operated terminals to insert your wire
- A multimeter (voltmeter and continuity tester)

Schematic

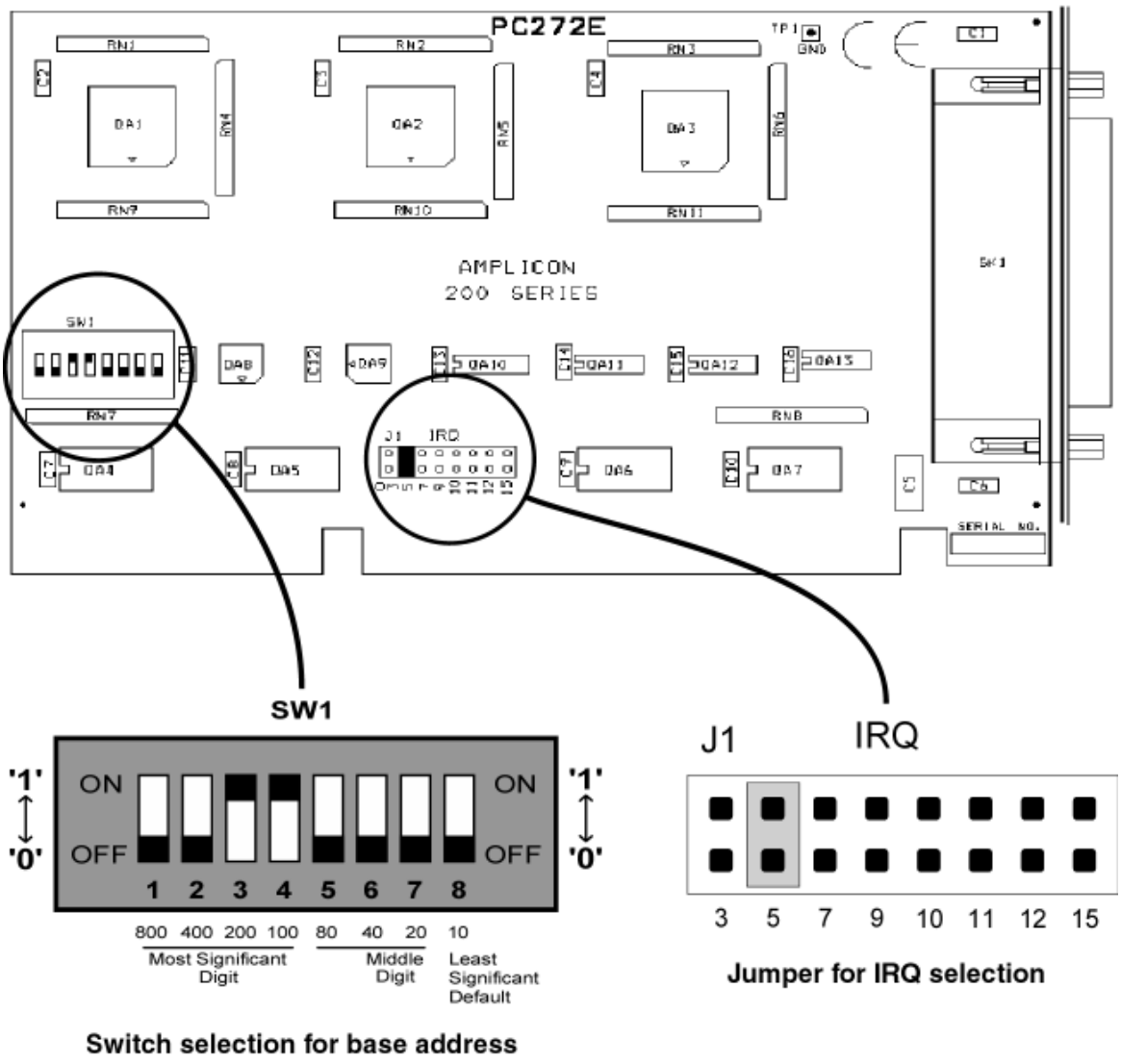
This diagram shows the sort of thing you're aiming at with Amplicon digital I/O hardware.



5.6.2.1 PCI272 and PC272E digital I/O cards

Configure the cards and install them in the computer.

ISA version (PC272E)



Switch selection for base address

There are two things to configure on the card: the I/O base address (default 0x300) and IRQ (default 5). See the electronic manual for details of how to set these (`\manual\pc272e.pdf` on the Amplicon CD-ROM), or the figure above.

These settings need to be chosen to avoid conflicts with other cards installed in the com-puter; if you use more than one Amplicon board you will certainly have to change the jumper settings for at least one board. In a plain PC system, a single-board installation will probably work without changing anything.

We generally use two PC272E cards, and as IRQ 5 is normally used by a network card in our computers, we use the following settings:

First card: IRQ 10, base address 0x300
Second card: IRQ 11, base address 0x320

You need to tell the Amplicon drivers what settings you have chosen.

Installing Amplicon drivers for the PC272E (ISA) card under Windows NT 4

Insert the Amplicon CD, which comes with the PC272E cards. It will auto-run.

Choose "Software drivers and manuals" from the main menu. Choose "32-bit drivers for the PC272E card" from the matrix presented. The drivers will be installed. Note that the manuals for the card are also available in this matrix. To read them, install **Adobe Acrobat Reader** from the CD. The drivers have their own, very detailed manual (ampdio32manual.pdf).

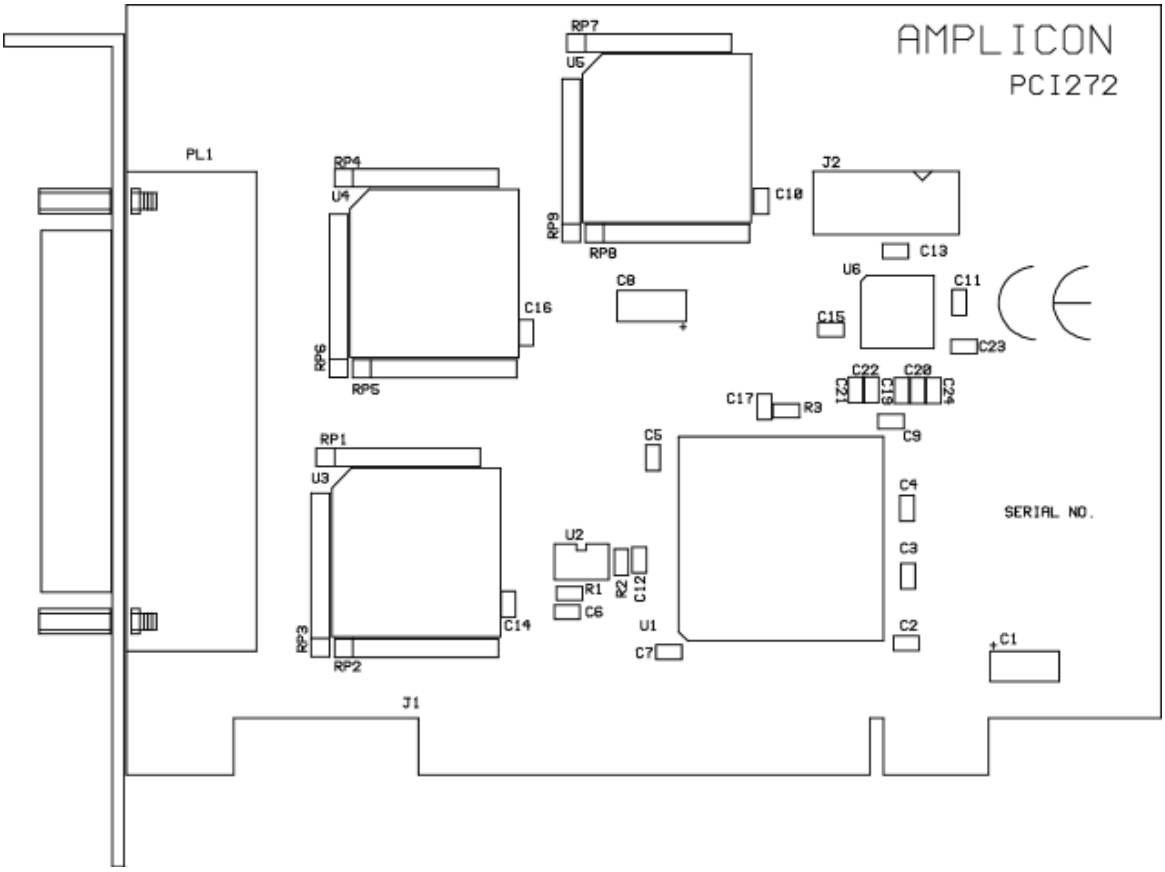
Run *Control Panel* → *Amplicon DIO* and configure each of your cards. The first card is called DIO 0, the second DIO 1, etc. They will be PC272E cards, and you should set the *interrupt* and *base address* settings to match those you physically configured the cards to. You will then need to reboot the computer.

Installing Amplicon drivers for the PC272E (ISA) card under Windows 2000

The procedure is essentially the same as for Windows NT 4, except that the CD installation is almost entirely automated. To configure the interrupts and port addresses for each card, use *Start* → *Settings* → *Control Panel* → *System* → *Hardware* → *Device Manager*.

Under Device Manager, there is an entry is called *Amplicon Analogue/Digital IO Counter Timer Cards;* this has a submenu which lists *PC272E Digital IO Card;* under *Settings* you'll find *DIO Port Number*, which may be set from DIO0 up to DIO7.

PCI version (PCI272)



You don't need to configure anything on the PCI card. Everything is done in software (from Control Panel).

Installing Amplicon drivers for the PCI272 (PCI) card under Windows NT 4

Installing Amplicon drivers for the PCI272 (PCI) card under Windows 2000/XP

Install the PCI272 card and turn the computer on. Windows will detect the card as "PCI Data Acquisition and Signal Processing Card" (or, if you've installed one of these before, as "PCI272 Digital IO card") and tell you that it wants to install drivers. Insert the Amplicon CD that came with the card; Windows will automatically search for and find the drivers.

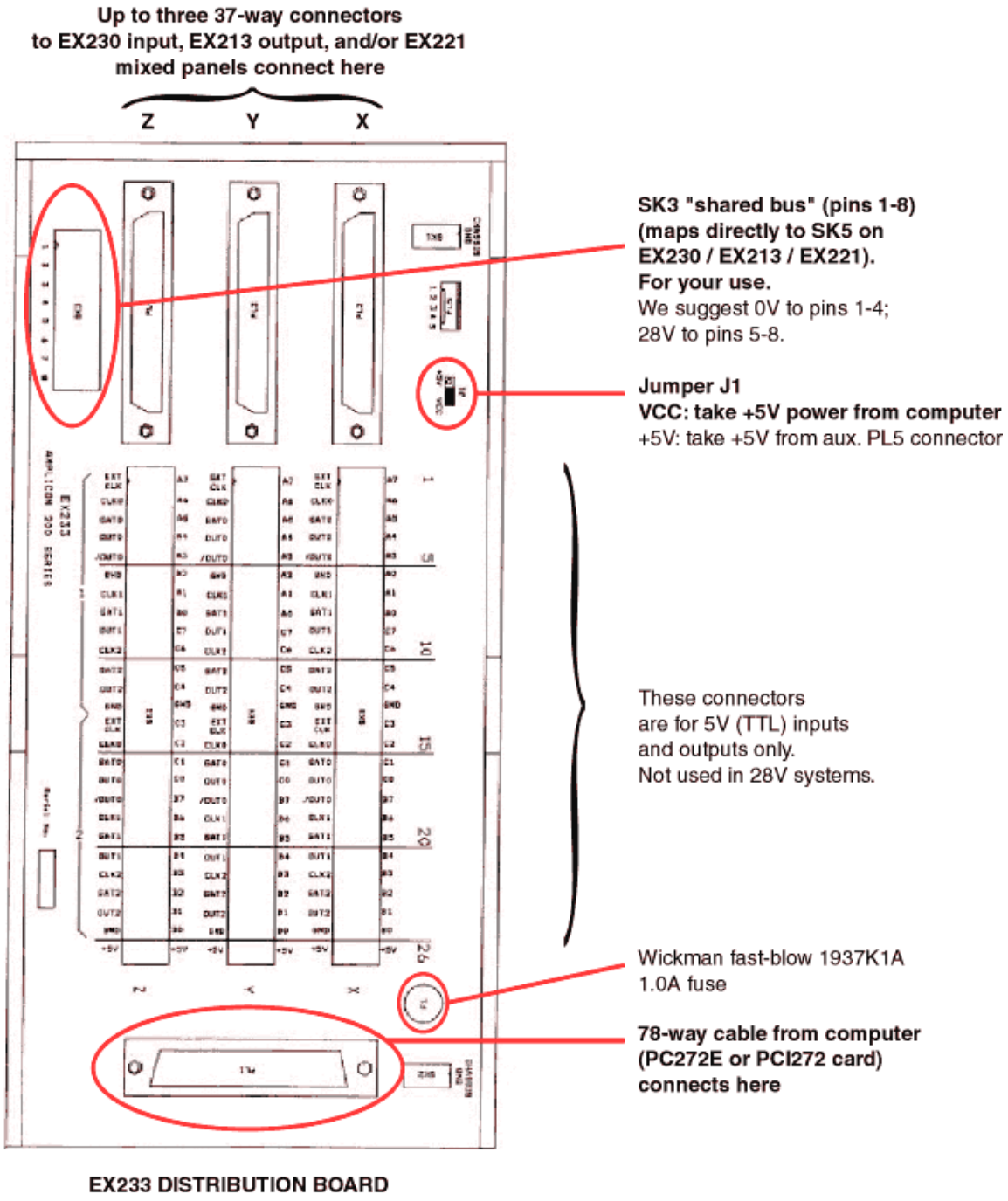
To configure the card, choose *Start* → *Programs* → *Administrative Tools* → *Computer Management* (or *Start* → *Settings* → *Control Panel* → *Administrative Tools* → *Computer Management*). Then choose *Device Manager*. In the right-hand tree, you'll see an entry for *Amplicon Analogue/Digital IO Counter Timer Cards*, under which you'll find *PCI272 Digital IO card*. Right-click this and choose *Properties*. Under the *Settings* tab you can choose the DIO Port Number, which determines the order the cards are seen by Whisker.

5.6.2.2 EX233 distributor boards

There's not a lot to configure on this board. See the diagram below.

- **Ensure J1 is set to VCC.** (This connects a 5V power signal from the computer down all of the 37-way cables. This will be used to power the optoisolator optical detection chips on input boards.)

- To simplify wiring, use the "shared bus" facility of the distribution board. Eight connectors on the EX233 board, collectively labelled SK3, are provided for your own use. Whatever you connect to these is connected via the ribbon cables to an equivalent block of 8 connectors on any input/output panels that you connect to the distribution board (where they are labelled SK5 to confuse you). We need to share out +28V and 0V to all the panels. So **wire 0V (GROUND) from the power supply to pins 1-4 of block SK3 on the EX233 board, and +28V from the power supply to pins 5-8.** ***(This is an arbitrary choice, but I will assume you've done this from now on.)***



EX233 DISTRIBUTION BOARD

5.6.2.3 EX213 relay output panels

Med Associates devices are turned on by shorting their control lines to ground (0V).

- Ensure **J1-J3 are connected,** as per the diagram below.
- Ensure **J4-J11 are set to the RIGHT (source driver mode),** as per the diagram below.
- To use relay outputs, **set J12-J35 to the RIGHT.**
- To supply ground to all the relay circuits, **connect ground (e.g. SK5 pins 1-4) to "A GND", "B GND" and "C GND"** on the block labelled SK6.
- To provide power to operate the relays, **connect +28V (e.g. SK5 pins 5-8) to "A VCC", "B VCC" and "C VCC"** on SK6.

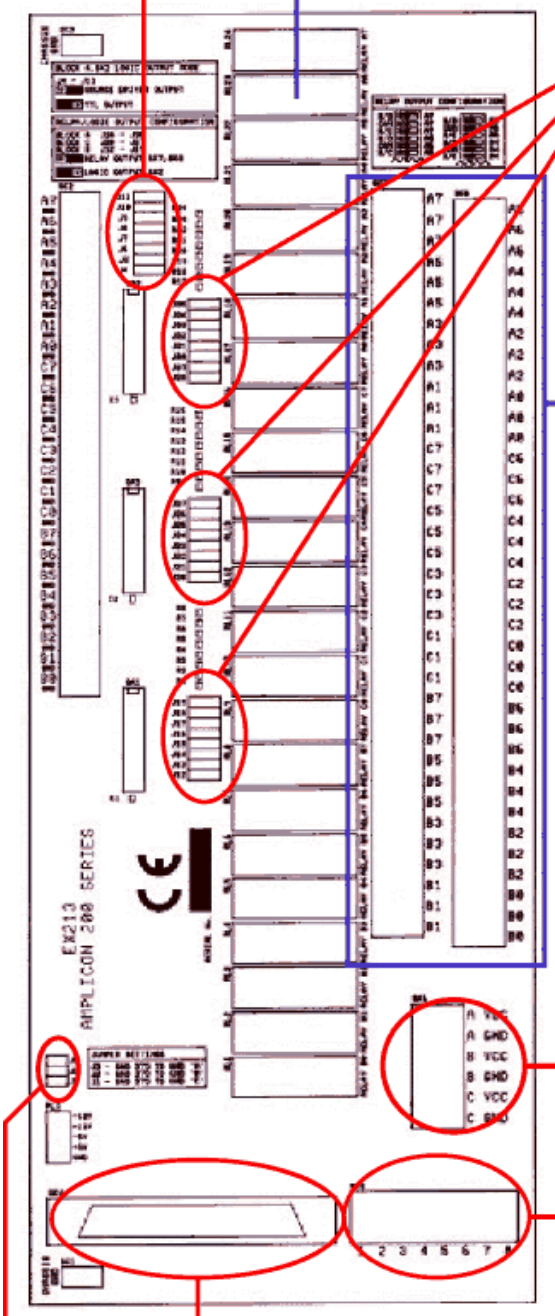
Relays can be thought of as follows. They have a common (COM) pin. When there is no power applied to the relay, the COM pin is connected to the normally closed (N/C) pin, but not to the normally open (N/O) pin. When power is applied, this reverses (so N/O is connected to COM and N/C is disconnected). All the relays act independently and have separate COM lines, one per relay, but to simplify things for Med Associates devices, we want a shared ground:

- **Connect all the COM pins together** on the two long blocks (A0-7, B0-7 and C0-7). **Then connect these lines to a ground line** (e.g. pins 1-4 of SK5).

Make sure that your boxes use the same ground, too (i.e. that they run off the same power supply). Then you can **connect your device control wires to the N/O pins.** This will keep your apparatus *off* by default. If you want it on instead — for example, if your supplier has wired a dipper so it's up by default and you want it down — you can wire it to the N/C pin instead.

J4-J11 (apply to block A only)
Left = TTL mode
Right = source driver mode (needed for relay operation)

Relays



J12-J35
Left = logic outputs
Right = relay outputs

Relay connections.
Each has a N/O (top), COM (middle), and N/C (bottom) pin. Connect N/O to devices that should normally be off. Connect N/C to devices that should normally be on. Connect all COM lines together and to 0V (ground) for Med Associates devices.

EX213 OUTPUT PANEL

SK6. Power/ground to the relays.
Connect "A VCC", "B VCC", "C VCC" to +28V
Connect "A GND", "B GND", "C GND" to 0V
(e.g. via SK5 pins 1-4 [0V], 5-8 [+28V]).

SK5 bus (matches SK3 on EX233)
Following our suggestion,
pins 1-4 are 0V; pins 5-8 are +28V.

37-way connector from EX233 attaches here

J1-3. Connects logic ground of the computer to the relay's ground circuits. Leave them connected for proper operation.

5.6.2.4 EX230 optoisolator input panels

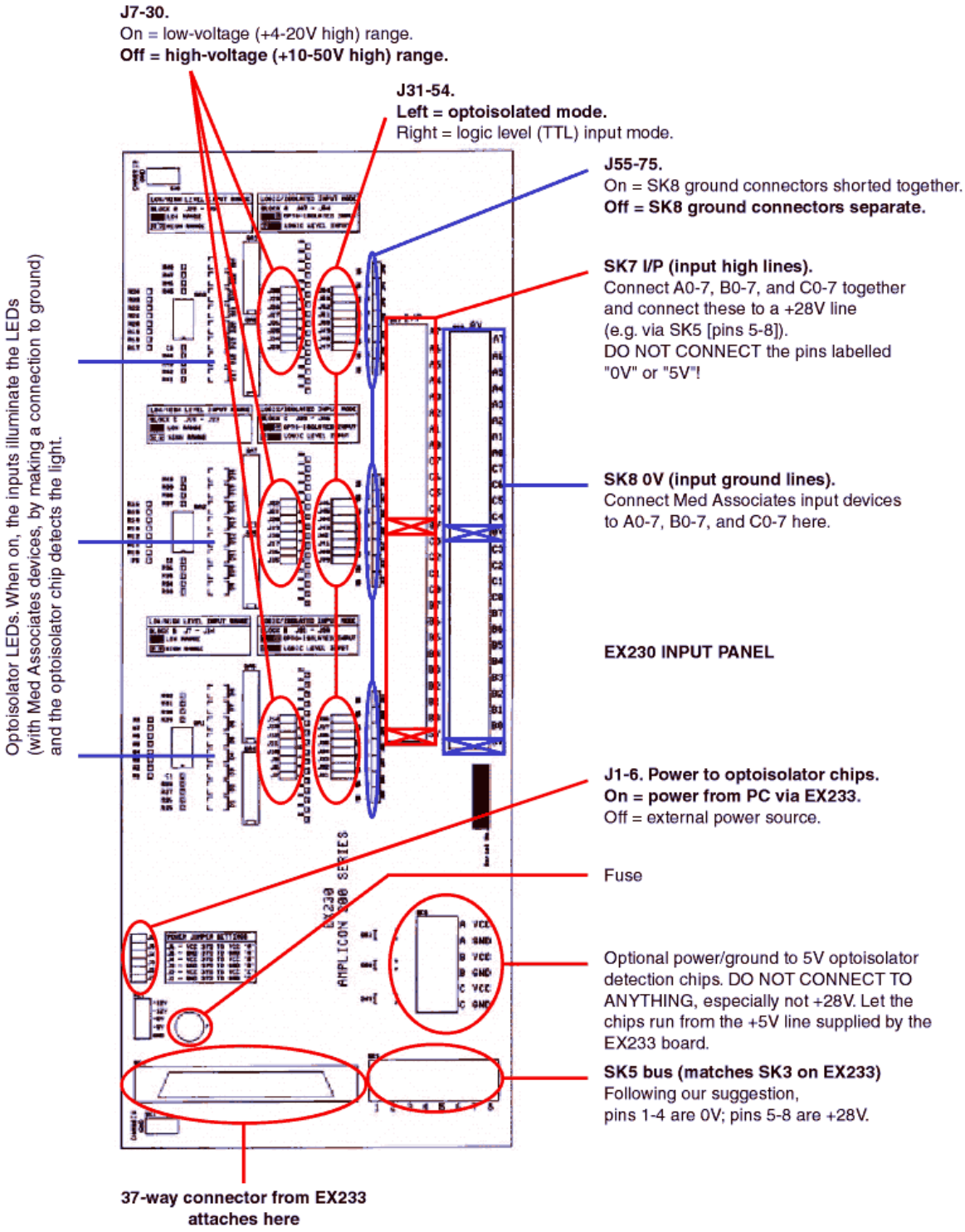
I will give examples for Med Associates inputs, which, when activated, short their control lines to ground (and share a common +28V line). Here's how we arrange things:

- So the board takes power from the computer, **ensure jumpers J1-J6 are all CONNECTED.** (See diagram below.)
- To use 24V inputs, **disconnect jumpers J7-J30.** Don't lose the jumpers; attach them to one pin only.
- To use optoisolated inputs, **ensure jumpers J31-54 are all set LEFT.**
- So the ground connectors are all independent, **disconnect jumpers J55-75.** (Again, don't lose the jumpers!)
- So the +28V lines are all shared, **connect A0-7, B0-7 and C0-7 together on the "SK7 I/P" block.** Pieces of paperclip are great for this, or you could use a single piece of wire. **Then connect those lines to a +28V line.** For example, pins 5-8 of SK5 should be attached to +28V if you have wired up the EX233 board as I suggested — but *don't connect up the pins marked "0V" or "5V" in this block!* You'd be shorting 0V or 5V to 28V, probably with some sparks.

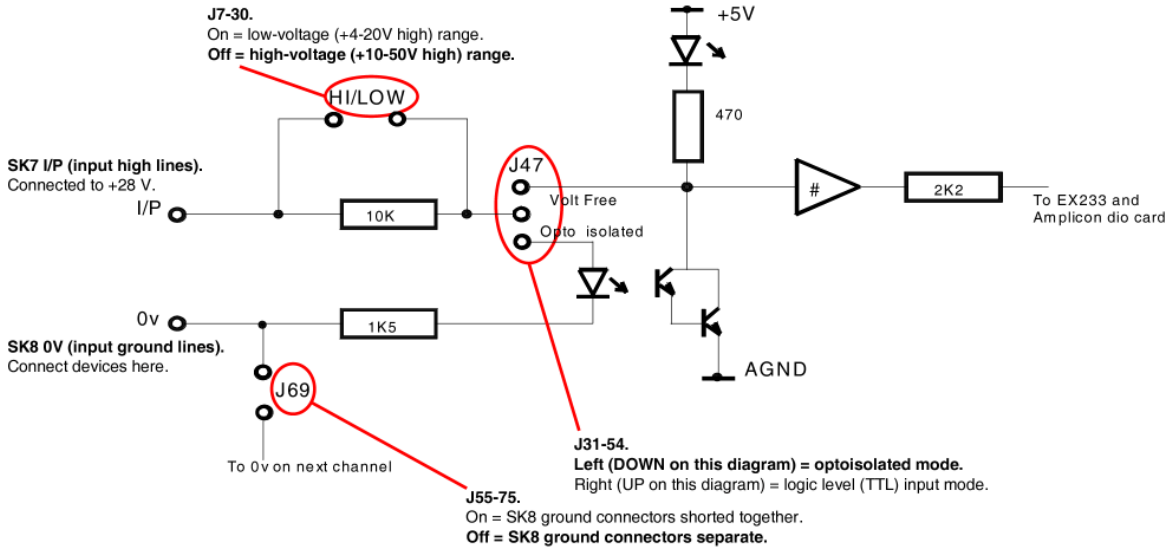
This is not as neat as we'd like — there's wire everywhere — but this is because the engineers at Amplicon expect people to share a ground wire and have separate +28V control lines, not the reverse. As optoisolators don't work if you plug them in backwards, we have to share the +28V lines by hand.

You will then be able to **connect the data lines** from the Med Associates devices to A0-7, B0-7 and C0-7 on the SK8 0V block (one line per device).

Ensure that the Med Associates devices use the same ground as the rest of the system.

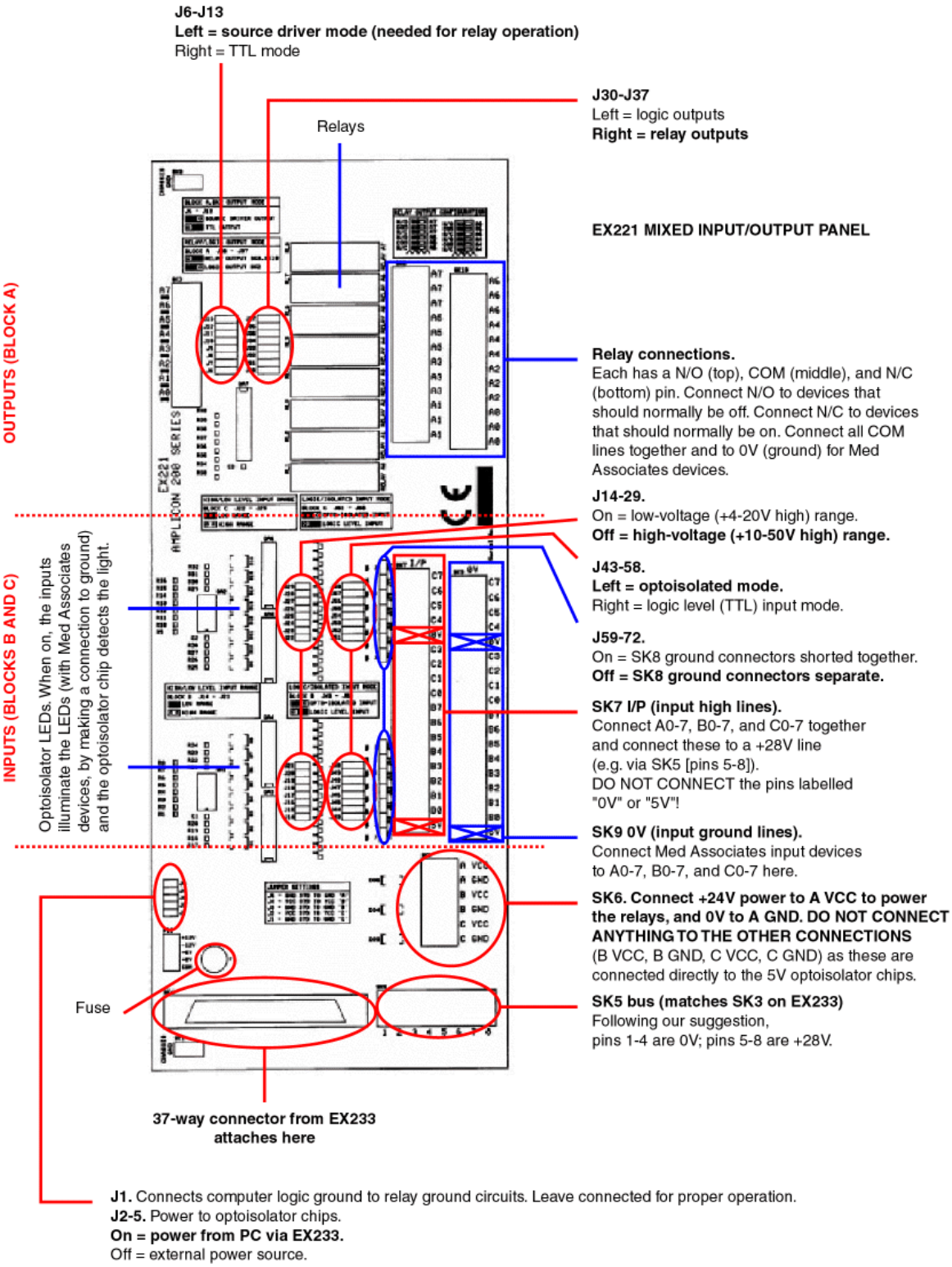


Here's an annotated circuit diagram of a single input:



5.6.2.5 EX221 mixed input/output panels

The first 8 lines (block A) are outputs, and the last 16 (blocks B and C) are inputs.



5.6.2.6 Connecting everything up

Connect each of the input/output panels to the EX233 distribution board with the 34-way cables

The three columns on the EX233 board are labelled X, Y and Z; you may find it useful to label the cables going to the input/output panels accordingly. It helps when wiring things up later.

Connect your apparatus

Plan your wiring diagram before starting (see Amplicon wiring diagram). To remind you:

- Connect Med Associates inputs to A0-7, B0-7 and C0-7 on the SK8 0V block of the EX230 input panel.
- Connect each output control wires to either the N/O or the N/C pin of one of the relays on the EX213 output panel.
- Ensure that output ground wires are (eventually) connected to the COM pin of the same relay.

Plug the computer into the uninterruptible power supply (UPS)

Make sure the system looks OK

It should look something like the schematic shown earlier.

Ensure you do not place subjects in the operant chambers until...

... you have installed the system, tested the hardware and read the discussion of life-threatening devices and safety systems.

5.6.3 Amplicon digital I/O board wiring map

If you have an Amplicon-based system, the first 144 lines as seen by Whisker correspond with the physical Amplicon maps as follows (note that this information is also available on the Line Status view):

```
Wiring panel
|       Line on panel
|       |       Software line# (first DIO board) (board 0)
|       |       |       Software line# (second DIO board) (board 1)
|       |       |       |

X       A0      0       72
        A1      1       73
        A2      2       74
        A3      3       75
        A4      4       76
        A5      5       77
        A6      6       78
        A7      7       79

        B0      8       80
        B1      9       81
        B2      10      82
```

	B3	11	83
	B4	12	84
	B5	13	85
	B6	14	86
	B7	15	87
	C0	16	88
	C1	17	89
	C2	18	90
	C3	19	91
	C4	20	92
	C5	21	93
	C6	22	94
	C7	23	95
Y	A0	24	96
	A1	25	97
	A2	26	98
	A3	27	99
	A4	28	100
	A5	29	101
	A6	30	102
	A7	31	103
	B0	32	104
	B1	33	105
	B2	34	106
	B3	35	107
	B4	36	108
	B5	37	109
	B6	38	110
	B7	39	111
	C0	40	112
	C1	41	113
	C2	42	114
	C3	43	115
	C4	44	116
	C5	45	117
	C6	46	118
	C7	47	119
Z	A0	48	120
	A1	49	121
	A2	50	122
	A3	51	123
	A4	52	124
	A5	53	125
	A6	54	126
	A7	55	127
	B0	56	128
	B1	57	129
	B2	58	130
	B3	59	131
	B4	60	132
	B5	61	133
	B6	62	134
	B7	63	135

C0	64	136
C1	65	137
C2	66	138
C3	67	139
C4	68	140
C5	69	141
C6	70	142
C7	71	143

5.6.4 University of Cambridge: Hubert's boxes (25-way connector style)

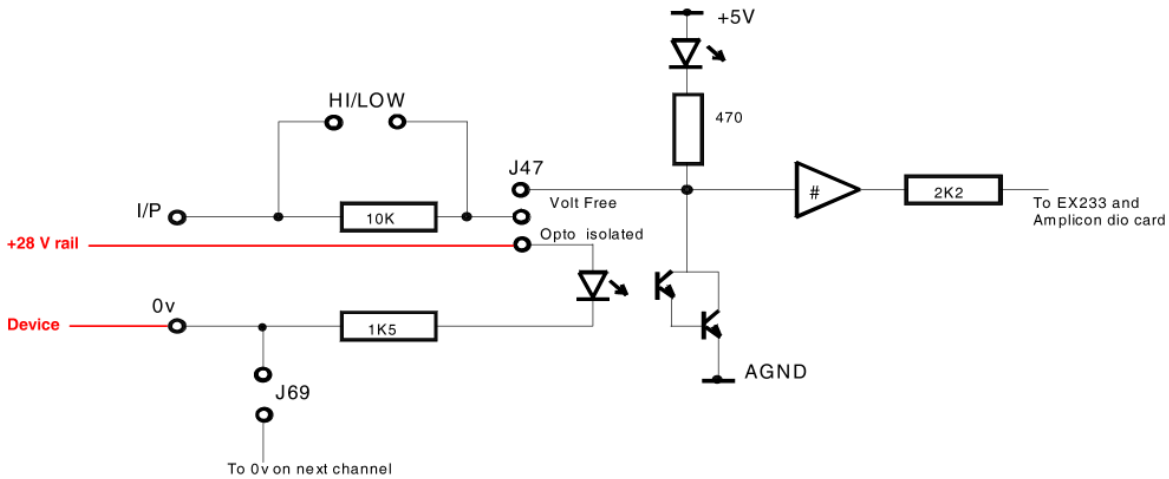
Hubert Jackson has created boxes for the University of Cambridge that enclose all the Amplicon hardware. They have the following connections:

- +28V DC in
- 0V DC in
- 78-way cable from Amplicon card in the computer
- LED to indicate when power is being delivered to the front panel
- 4mm connectors on the front panel. These are:
  - one **black:** ground (0 V) to devices
  - one **red:** switched power (+28 V)
  - **white** connectors: inputs (from devices to computer) - shorted to ground by the device to signal that it's on
  - **blue** connectors: outputs (from computer to device) - shorted to ground to switch on a device

Lines **66 and 67** (or **138 and 139** if it's the second box attached to the computer) are failsafes. The server should be configured to turn 66 (or 138) **on** during operation and 67 (or 139) **off.**

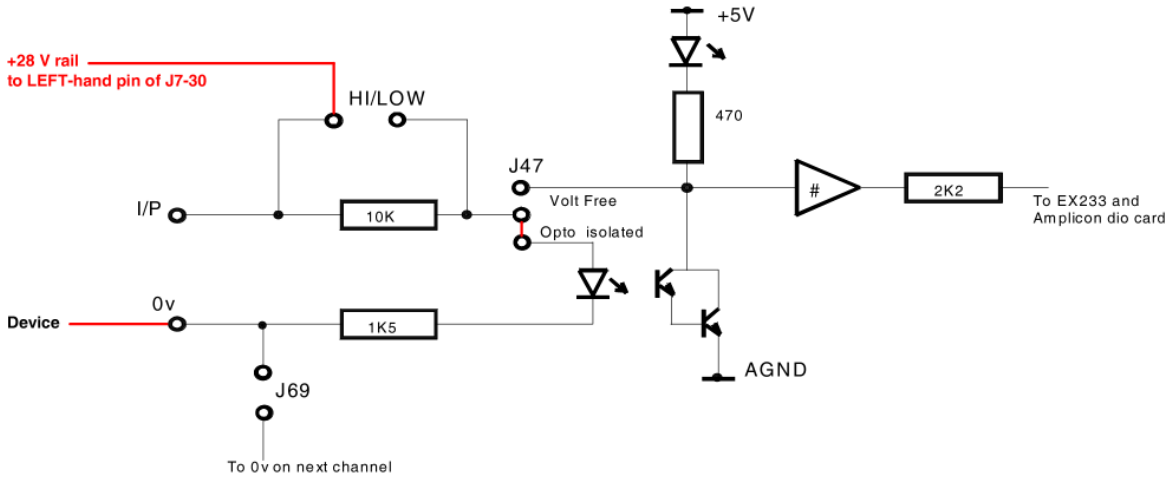
Input (EX230) board wiring

Hubert actually wired up the boards by removing the J31-54 jumpers and connecting a +28V rail to the left-hand pin of J31-54. See the circuit diagram. That effectively gives this:



It certainly seems to work much of the time. Yet we do seem to have a high failure rate on the input boards. I'm a little concerned that this circuit bypasses the high-voltage protection resistor. I

would have used a +28V rail to the left-hand side of J7-30, and had J31-54 connected left (as usual) and J55-75 open (as always), like this:



5.6.5 Technical note - Amplicon power-on behaviour

When the computer is turned on, all devices connected to an Amplicon PC272E are turned on. This can be dangerous.

Why does this problem occur?

When the Amplicon PC272E cards power up, they reset their 82C55 controller chips. When the 82C55 chip resets, it places all its lines into input (high-impedance) mode. When a line is set to input mode, a set of pull-up resistors on the PC272E card bring the voltage high, and if an output device is connected to this line, it will be turned on. These resistors are not configurable (unless you feel like resoldering the board), and the net result is that the outputs stay on until the software reconfigures the lines for output. *Sources: Peter Adams (Amplicon Technical Support, tel. 01273–601331) and the Appendices for the 71055 and 8255 controller chips on the Amplicon CD-ROM.*

These are not valid solutions:

- Swapping all outputs from the "normally open" to the "normally closed" relay pins. In this situation, short power cuts (which reset the computer) are safe, but now long power cuts (which may leave certain computer motherboards switched off) are dangerous: the box power supply comes on but the computer stays off.

What are the possible solutions?

- Use different digital I/O cards. [Impractical for us.]
- Rewire the existing cards. [Difficult.]
- Use a motherboard that guarantees never to power on unsupervised. [Very hard to guarantee.]
- Plug the computer and/or DC power supply into a trip switch, so a mains power cut cuts power to everything until reset by a human. [Possible with an "industrial self-testing plug", £48 each from RS, but not the best solution.]
- Wire the DC power supply for the critical devices through a fail-safe mechanism. Probably the best is to wire it through two relays such that one relay must be ON and the other must be OFF in order for power to reach the devices. This is extremely unlikely to happen by chance (all the problems described so far switch all the relays on or off together). The server software provides support for assigning some relay outputs to this function. The only criterion that must be fulfilled is that the relays can take the current: the relays on the EX213 board can switch AC to 60VA and DC to 28W (i.e. 1A at 28V). [This is a very good solution, and can be combined with an UPS.]

Further to this: the relays on the EX213 board will take 8A, but the tracks on the circuit board will probably handle only 1A (Russell English, Amplicon tech. support, pers. comm. 1 Dec 2000). This is enough for individual devices, but not for the entire power supply for a system. The TTL outputs on the EX233 distribution board will provide about 2.5 mW; the high-voltage (24V) logic outputs on the EX213 board will provide 100 mW; the TTL-capable outputs on the EX213 board will probably provide 100 mW too (though nobody seems absolutely convinced of this). As a typical 10A relay takes about 250 mW to drive its coil (so approx. 10 mA at 24 V; coil resistance approx. 2.4 kW), we opted for a high-power relay that is itself switched by a relay on the EX213 board.

- Install an UPS for the computer. This eradicates the problem and gives data protection. If you're willing to spend money on a very powerful UPS, you could also guarantee DC power to the operant chambers, so your tasks would not even be interrupted by a power cut. [This is a good solution and can be combined with the fail-safe relay device.]

See also

- Danger with critical devices
- Fail-safe relay system

5.6.6 Technical note - input and output modes for Amplicon panels

Input specifications

Inputs operate in "low voltage" mode (-5V to +1.5V low, +4V to +12V high) or "high voltage" mode (-15V to +2V low, +10V to +30V high), with or without optoisolation, chosen for each input individually. Optoisolation is recommended, and the voltages quoted are for the optoisolated mode.

The Amplicon EX230 input card

SK5 is a 'user' connector block with 6 connections on it. It is connected via the 37-way ribbon cables to the equivalent block on the EX233 card. This is handy for providing external power and data lines that are shared between all cards. SK6 provides power and ground to the board itself (and is usually not needed as the 37-way connector provides power; see notes for J1-J6 below). SK7 is the I/P (input) connector block. Beware: there's a stray 0V connector in the middle, and a 5V connector at one end. SK8 is the 0V (ground) connector block. J1-J6 connect power and ground lines from the 37-way D connector to the VCC/GND connectors that provide power for the board itself. J7-J30 select high or low voltage ranges for each of the 24 inputs. J31-J54 select optoisolated or logic mode for each of the inputs. J55-J75 short the ground lines together. They are in three blocks of 7, which connect the ground lines of the three blocks of 8 connectors together.

The optoisolators do not work if you reverse +28V and 0V.

Input voltage ranges

In the 'high-voltage' range, 10-30V is considered 'on' or 'high', and anything less is considered 'off' or 'low'. If you want to use 5V inputs instead (where the range 4-12V is considered 'high'), connect the jumpers.

Med Associates inputs

Activating a device shorts its data lead to ground.

So the Amplicon board needs to be set to high-voltage, optoisolated mode. Then all the +28V lines need to be connected to +28V (it's easy to do this by inserting paperclips into the SK7 I/P connectors!); then the ground (SK7, 0V) connectors need to be disconnected from each other by removing J55-75; the data leads can then be wired to the ground connections (SK7). Thus when the device shorts its data lead to ground, the circuit is completed and the optoisolator LED illuminates.

I can't think of an alternative, but it may be worth asking Amplicon if they can redesign the card so shorting the I/P lines together is possible.

Output specifications

The output lines can operate in several modes. (1) "TTL" mode provides TTL voltages (approx. 0V low, +5V high) for low-power devices, such as transistors. If you need to switch high-power devices electronically, you can use these lines to drive the transistor; this is what happens in the Cenes output panels. (2) "Source driver" mode does something slightly mysterious and will be ignored. (3) "High level logic output" mode provides +0.7V (off) or +23V (on) and will drive devices up to 100 mA directly. (4) "Isolated relay contact output" mode switches a relay on the circuit board. When the output is switched on, the relay contact is closed. (There is a second relay contact that behaves oppositely.) This can obviously switch high-power devices that use their own power supply, and is the easy way to switch devices that need more than 100 mA.

Outputs on the EX213 panel

The alternative to relays is logic output. For blocks B and C, this is always high-voltage (0 off, 24V on). For block A only, you can choose high-voltage or TTL voltage (0 off, 5V on) using SK2. The text assumes you are using relays.

5.7 Amplicon analogue I/O hardware

This section is **in development.** It covers

5.7.1 Buying Amplicon analogue hardware

Amplicon Liveline Ltd (http://www.amplicon.co.uk (telephone 0800–525–335, technical support 01273–608–331, next-day delivery is usually available). Prices shown here may be out of date. Last checked November 2002.

PCI 230 analogue I/O card

This card has **16 analogue input** lines (which can be paired to give 8 differential inputs); it will take input signals in the range ±10V. It can sample inputs at up to 312 kHz (across all channels, i.e. one channel at 312 kHz, two channels at 156 kHz, etc.) using a FIFO. It has **two analogue outputs** but these do not have FIFO buffers, so outputs must be driven directly by the software. The output resolution is 12-bit, in the range 0–10V or ±10V; therefore, the best output resolution is 2.4 mV. **The outputs are therefore unsuitable for electrochemistry,** which needs 1 mV output resolution (O'Neill 1994). The board also has **24 digital I/O lines,** and **three timers/counters,** but the timers/counters are not made available for users by Whisker.

- Amplicon PCI 230 board, product 909-893-83. **£450**.
- 50-way breakout board, product 909-651-32, **£59.**
- 50-way cable to connect card to breakout board, product 909-663-59, **£56.**
- Disposable anti-static wrist band, product 700-145-12, **free.**

PCI 260 analogue I/O card

This is similar to the PCI 230, but it does not have the analogue output channels or the digital I/O channels.

PCI 224 low-precision high-speed analogue output card

This card has **16 analogue output** lines, each with 12-bit resolution (giving 4096 levels of output voltage). Its range can be set to ±10V, ±5V, ±2.5V, ±1.25V (bipolar), or 0–10V, 0–5V, 0–2.5V, 0–1.25V (unipolar). Therefore its best resolution (using the 0–1.25V range) is 0.3 mV. This makes it suitable for electrochemistry control. Its worst resolution (using the ±10V range) is 4.9 mV. It has no input channels.

- Amplicon PCI-224, product 909-893-93, **£699**
- 37-way cable, product 909-561-09, **£46.**
- 37-way breakout board, product 908-919-50, **£35.**
- Disposable anti-static wrist band, product 700-145-12, **free.**

PCI 234 high-precision high-speed analogue output card

This has **four output channels** with 16-bit resolution (65535 levels). The available output ranges are ±10V and ±5V (bipolar) only. The best resolution (at ±5V) is 0.15 mV; the worst (at ±10 V) is 0.3 mV. It has no input channels. (It is also rather more jumper-dependent and less software-configurable than the PCI-224.)

- PCI-234 is product 909-894-03, £749.

You may also need:

- **DIN rail mounting.** Amplicon cards will clip onto DIN rail, which you can attach to something convenient, like a wall. Deep top hat DIN rail (35 mm wide, 15 mm deep), punched, from RS Components (http://rswww.com) is part no. 176-703 and costs £3.24 per metre.
- **Oscilloscope with signal generator.** Not vital, but helpful for development purposes! For example, GW Instek GOS-620FG 20MHz analogue oscilloscope with 1MHz function (signal) generator; comes with probes, etc.; www.goodwill.com.tw/english and www.maplin.co.uk, £349.99 (Maplin code NZ96E).

5.7.2 PCI 224 analogue output card

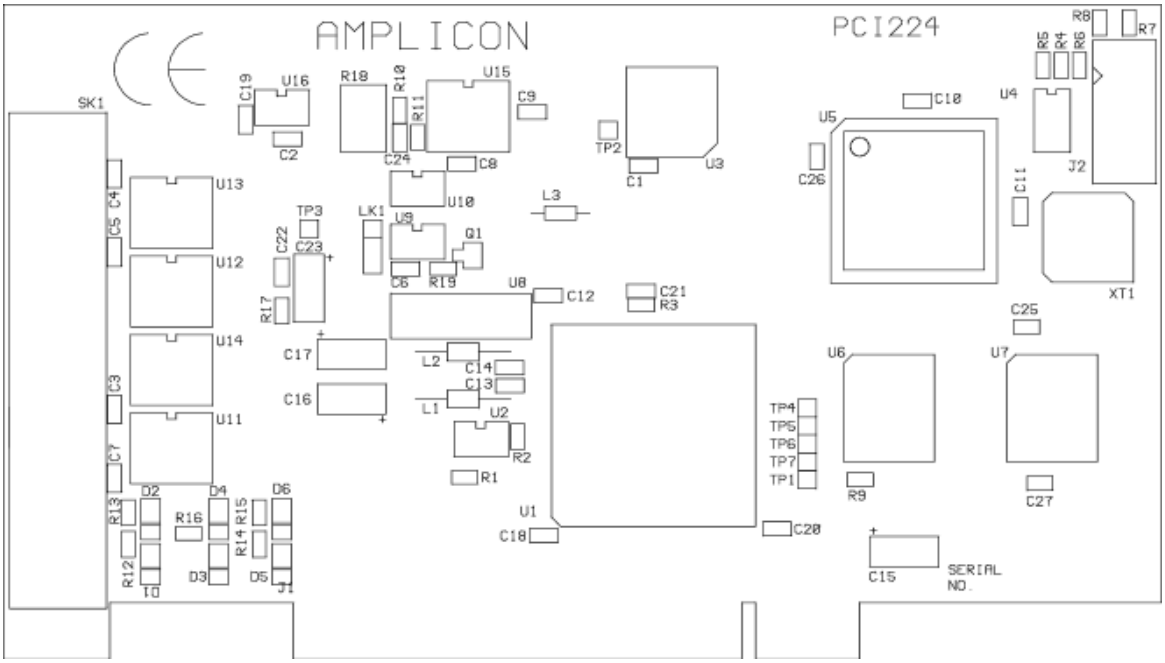
This is a card that is sensitive to electrostatic discharge.
You must, as a minimum, wear an earthed wrist strap when handling the board outside its protective bag.

About the PCI224 card

This card has 16 analogue output lines, each with 12-bit resolution (giving 4096 levels of output voltage). Its range can be set to ±10V, ±5V, ±2.5V, ±1.25V (bipolar), or 0–10V, 0–5V, 0–2.5V, 0–1.25V (unipolar). Therefore its best resolution (using the 0–1.25V range) is 0.3 mV. Its worst resolution (using the ±10V range) is 4.9 mV. It has no input channels.

Installing the PCI224 card

Ensure the computer is off. Install the card into the computer.



Installing the PCI224 under Windows 2000

Turn the computer on; it will detect the card. Insert the Amplicon driver CD when Windows asks to search for a suitable driver (and ensure that the CD-ROM is ticked for it to search). Windows will assign an IRQ (interrupt request) number to the card automatically. To check this, choose *Start* → *Control Panel* → *System* → *Hardware* → *Device Manager*. Ensure *View devices by type* is ticked. Under *Amplicon Analogue/Digital IO Counter Timer Cards*, you should find the PCI224 card. Click *Properties*. Check that it has been assigned an IRQ and a DIO number and there are no resource conflicts.

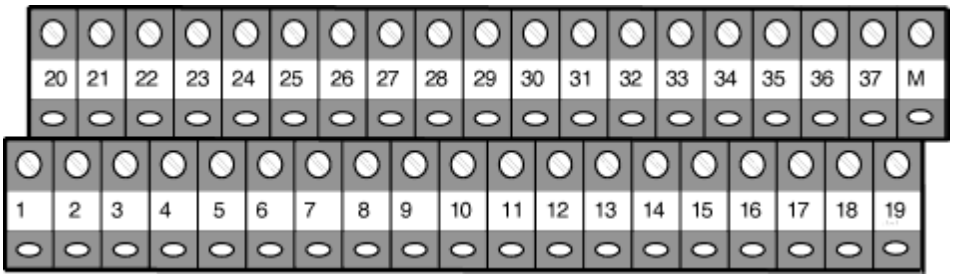
Connecting the PCI224 to devices

Connections for the PCI224 and the optional screw terminal assembly are shown below.

	Pin		
EXTCLK	1		
EXTTRIG	2	20	OUT2
EXTREF	3	21	DGND
DAC0	4	22	AGND
DAC1	5	23	AGND
DAC2	6	24	AGND
DAC3	7	25	AGND
DAC4	8	26	AGND
DAC5	9	27	AGND
DAC6	10	28	AGND
DAC7	11	29	AGND
DAC8	12	30	AGND
DAC9	13	31	AGND
DAC10	14	32	AGND
DAC11	15	33	AGND

DAC12	16	34	AGND
DAC13	17	35	AGND
DAC14	18	36	AGND
DAC15	19	37	AGND

(EXTTRIG - external trigger input for DACs; DAC0-15 - digital to analogue converter outputs; EXTCLK - external clock input; OUT2 - timer 2 output; DGND - digital ground; AGND - analogue ground.) The analogue outputs and ground are paired (e.g. DAC0 is paired with pin 22 AGND, DAC1 with pin 23 AGND, etc.).



On the screw terminal assembly, terminal M is connected to the shell of the D-type connector (which, on the PCI224 itself, is connected to the computer's chassis ground). Terminal C (which may be present to the left of terminal 20 - not shown above) is not connected.

- Use separate ground wires for analogue output, digital I/O, and power lines. Use heavy-gauge wire for the ground connection. Try to have the ground point close to the PCI224 terminal block to minimize ground impedance. If output lines are long, shield them.
- **All devices connected to the analogue outputs must have a minimum load impedance of at least 2 kΩ.**
- The output impedance of the PCI224 is 50 Ω and this should be matched on the user's equipment.

5.7.3 PCI 230 analogue I/O card

This is a card that is sensitive to electrostatic discharge.
You must, as a minimum, wear an earthed wrist strap when handling the board outside its protective bag.

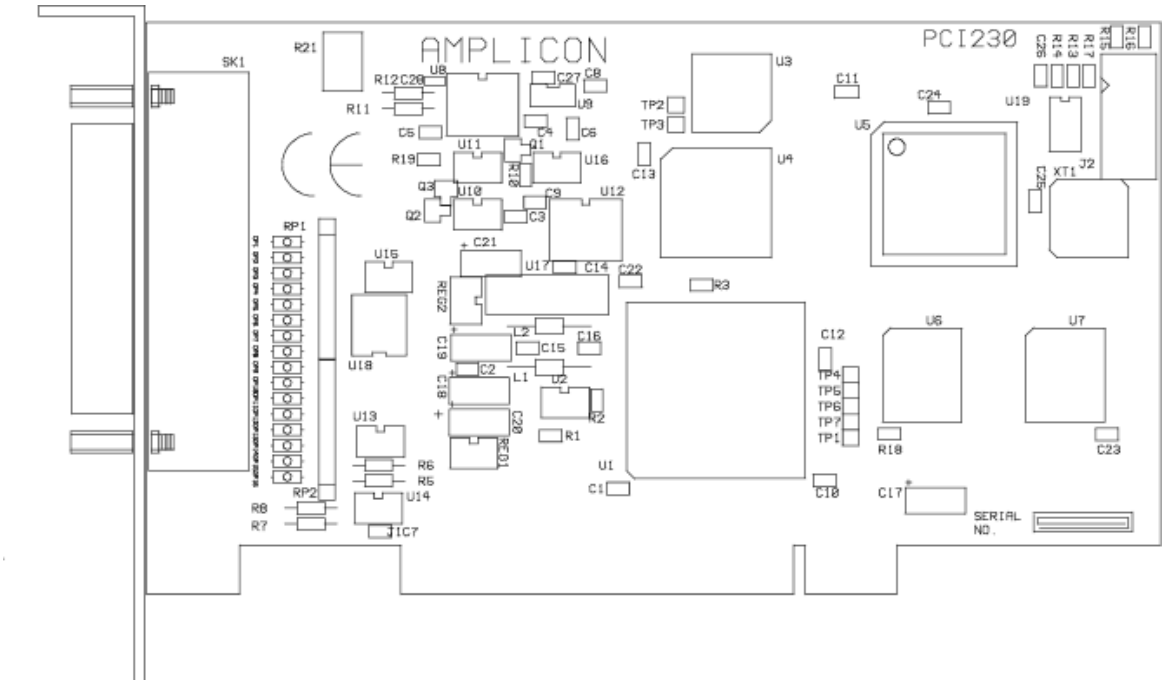
Additionally, analogue input voltages must never exceed ±15V when the system is powered on, or ±2V when it is off.
You must therefore connect unused inputs to ground before installation (see below).

About the PCI230 card

This card has 16 analogue input lines (which can be paired to give 8 differential inputs); it will take input signals in the range ±10V. It can sample inputs at up to 312 kHz (across all channels, i.e. one channel at 312 kHz, two channels at 156 kHz, etc.) using a FIFO. It has two analogue outputs but these do not have FIFO buffers, so outputs must be driven directly by the software. The output resolution is 12-bit, in the range 0–10V or ±10V; therefore, the best output resolution is 2.4 mV. The board also has 24 digital I/O lines, and three timers/counters, but the timers/counters are not made available for users by Whisker.

Installing the PCI230 card

Read the notes above and below. Ensure the computer is off. Install the card into the computer.

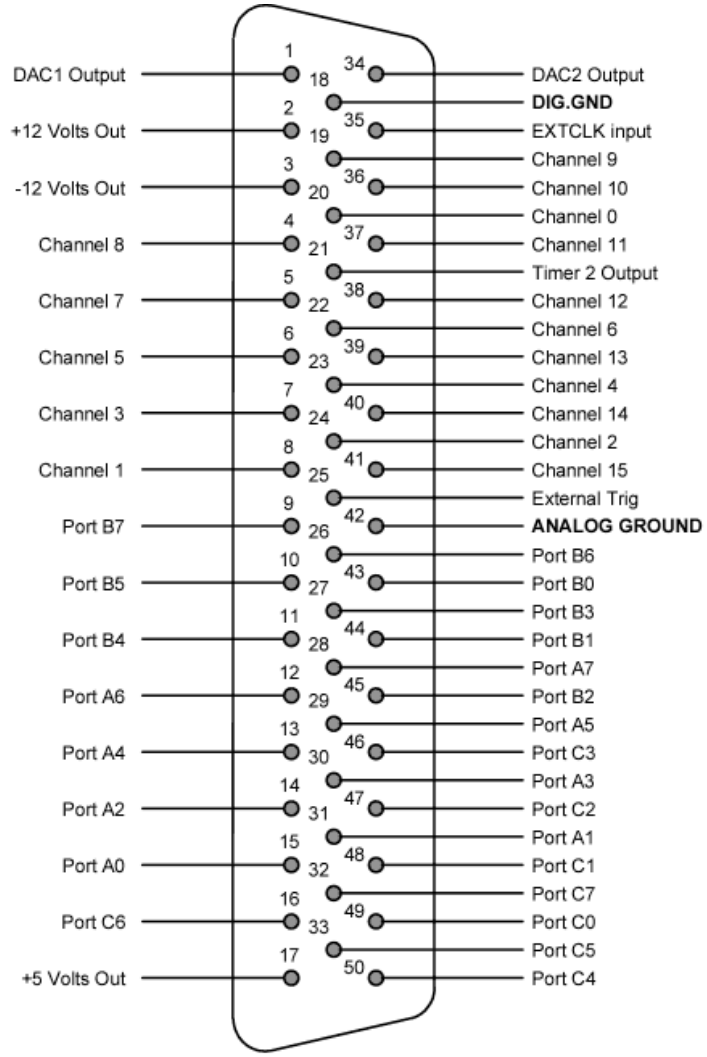


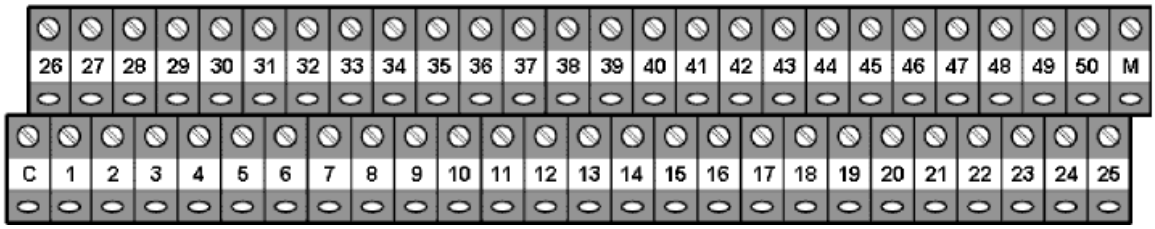
Installing the PCI230 under Windows 2000

Turn the computer on; it will detect the card. Insert the Amplicon driver CD when Windows asks to search for a suitable driver (and ensure that the CD-ROM is ticked for it to search). Windows will assign an IRQ (interrupt request) number to the card automatically. To check this, choose *Start → Control Panel → System → Hardware → Device Manager*. Ensure *View devices by type* is ticked. Under *Amplicon Analogue/Digital IO Counter Timer Cards,* you should find the PCI230 card. Click *Properties*. Check that it has been assigned an IRQ and a DIO number and there are no resource conflicts.

Connecting the PCI230 to devices

Connections for the PCI230 (viewed from the rear, i.e. looking at the socket) and the optional screw terminal assembly are shown below. "Channel" refers to analogue input channels; "Port" refers to digital I/O ports.





On the screw terminal assembly, terminal M is connected to the shell of the D-type connector (which, on the PCI230 itself, is connected to the computer's chassis ground). Terminal C is not connected.

- **Analogue input voltages must never exceed ±15 V when the system is powered on, or ±2 V when it is off. Therefore, you should connect any unused (floating) input channels to the analogue signal ground. You should also avoid having a signal present on an input when the computer is being switched on or off.** Specifically, this means that apart from any pins corresponding to channels you are using, you should connect pins 4, 5, 6, 7, 8, 19, 20, 22, 23, 24, 36, 37, 38, 39, 40, and 41 (channels 0-15, not in order) to pin 42 (analogue ground).
- Use separate ground wires for analogue input, analogue output, digital I/O, and power. Use a heavy-gauge wire for the ground connector. Try to keep the ground point close to the PCI224

terminal block to minimize ground impedance. If your input lines are long, shield them.
- Devices connected to the analogue inputs must have a source impedance of less than 250 Ω.
- **Analogue inputs must not exceed the voltage for the selected input range.**

5.8 Advantech I/O hardware

Physical installation information for Advantech card installation is not provided here.

See *Configure hardware / Advantech / BNC controller* for how to configure Whisker for Advantech/BNC devices.

See also
- www.advantech.com

5.9 ICS Advent I/O hardware

This section explains how to install ICS Advent digital I/O cards.

The specific card that is supported (the ICS Advent PCDIO24B-P ISA card) is an example of a card containing a simple Intel 82C55A I/O controller chip. Therefore, this section applies to any 82C55A controller card without its own drivers.

This hardware is supported as it was/is used by the original Monkey CANTAB system (supplied by Cambridge Cognition Ltd). The original Monkey CANTAB was a DOS-based program that expects a single 82C55A chip at I/O address 0x300.

THIS HARDWARE IS DEPRECATED. The driver is poor, because it is unsafe; it fails to guarantee protection for other devices installed in the computer. Whisker cannot provide that protection retrospectively.

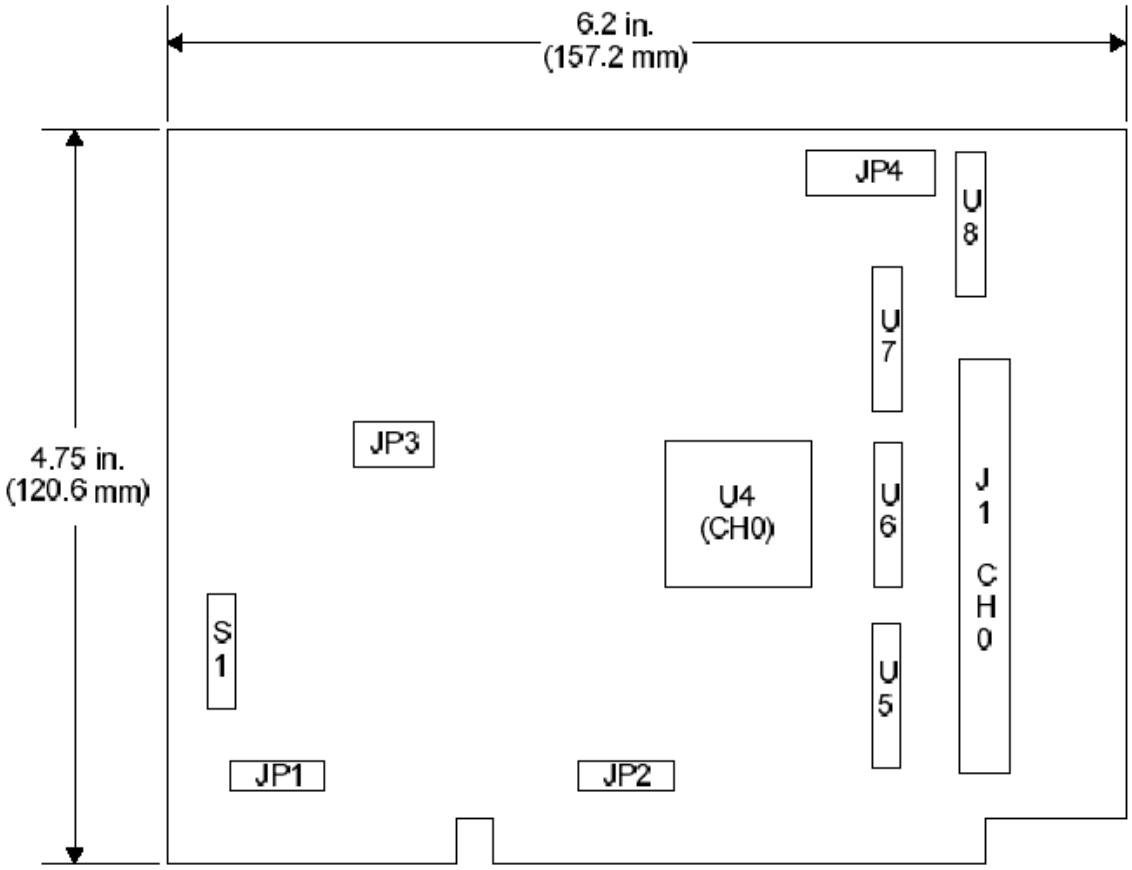
See also:

- Configuring ICS hardware within Whisker
- www.icsadvent.com (manufacturer's web site) - the company looks like it's now known as Kontron, and is also accessible as www.kontron.com

5.9.1 ICS Advent PCDIO24B-P card

Installing the ICS Advent PCDIO24B-P card

The specific legacy device we support is the **PCDIO24B-P card from ICS Advent;** this is an ISA card with 24 I/O lines, one 82C55A controller chip, and a 50-pin header port on the back. The label on the chip says PCDIO24C. It looks like this:



- **Jumpers JP1 and JP2 set the interrupt level** (options on JP1: N/A, IRQ14, IRQ15, IRQ12, IRQ11, IRQ10; options on JP2: IRQ9/2, IRQ3, IRQ4, IRQ5, IRQ6, IRQ7) (default: IRQ5, i.e. with nothing set on JP1). **Set this to N/A to disable interrupts** ("N/A" selected on JP1; nothing selected on JP2).
- **Jumper JP3 sets external interrupt triggering.** It has 5 jumpers (1-5), which can each be set to A (up) or B (down). (Default is 1 open, 2 open, 3 B, 4 B, 5 B.) **Remove all connectors to disable external interrupts.** Whisker will poll the card in software instead.
- **Jumper JP4 sets the operation mode for port C.** It is an 8-way jumper labelled "Channel 0"; each jumper can be set to A (top) or B (bottom) (default: B). To use mode 0, leave it on B - that's what we want (see below). For modes 1 and 2, you'd set this to A.
- **Switch block S1 sets the I/O base address.** It has 8 switches labelled "A9" to "A2"; the "on" direction is marked on the board (on is away from the 50-way connector, i.e. to the left in the diagram above). Bizarrely, when the switch is ON, that sets a 0 in the binary address number, and when the switch is OFF you get a 1. An example is shown in the following table, in which the address 0x300 is set. **Set this to an I/O address that's available in your computer** (see below).

ISA address line	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
Binary	0	0	1	1	0	0	0	0	0	0	0	0
Hex	3				0				0			
Switch setting			OFF	OFF	ON	ON	ON	ON	ON	ON	ON	ON

Note that this switch block allows you to set addresses from 0x000 to 0x3FF. As the drivers supplied with this card are very dangerous (they don't detect whether the address is used, or in

conflict with another device), I've [RNC] restricted Whisker to talking to a smaller set of addresses that aren't usually used for anything else. The PCDIO24B-P has a 4-byte address space (i.e. if you set a base address of 0x300, the card uses addresses 0x300 to 0x303 inclusive). So I'll allow the following:

(*Very dangerous things typically below 0x200.*)
VALID: 0x200-0x203 ... 0x270-0x273
(*ISA PNP typically at 0x274-0x279; LPT2 at 0x278-0x27F; etc.*.)
VALID: 0x280-0x283 ... 0x2E4-0x2E7
(*COM4 typically at 0x2E8-0x2EF; COM2 typically at 0x2F8-0x2FF.*)
VALID: 0x300-0x303 ... 0x370-0x373
(*Secondary IDE channel typically at 0x376; LPT1 typically at 0x378; all sorts of trouble from 0x380 up.*)

Find out what address space is free. In a Windows 2000 system, use *Start → Settings → Control Panel → Administrative Tools → Computer Management*. Then in the tree on the left-hand side, choose *Computer Management (Local) → System Tools → System Information → Hardware Resources → I/O*. This shows you a list of I/O addresses in use (click on *Address Range* to sort by I/O address). Ensure that whatever address you choose for your new card doesn't conflict with an address range already in use.

Installing ICS Advent drivers for Windows NT/2000

From the installation disk, run ICSPPNT\SETUP.EXE and follow the instructions.

5.9.2 Technical notes on ICS Advent 82C55A cards

Datasheet is 1143.pdf or PCDIOxB-P_1143.pdf. Manual is 00431231.pdf or "PCDIOB Series_00431231_rev2C.pdf".

Drivers are very simple (but potentially dangerous); you peek and poke the registers of the 8255 controller chips. For a 24-line card (the only kind we're using), the following register assignments apply. Base address + 0 = Port A (read/write); base address + 1 = Port B (read/write); base address + 2 = Port C (read/write); base address + 3 = control register (read/write). For cards with >24 lines, new 8255s are addressed from (base address + 4) onwards.

In a DOS system, the manual suggests checking IO address availability by running DEBUG.EXE and typing (for example) **I 300,** which should return "FF" (0xFF) if the address if free; in our example, you'd need to check addresses 300-303 in turn and ensure they're all free.

To control this card, we need to refer to the 82C55A data sheet...

- The control register has the following bits.

Bit	Meaning
D7 (MSB)	mode set flag (1 = active)
D6 and D5	Group A - mode selection (00 = mode 0; 01 = mode 1; 1X = mode 2)
D4	Group A - Port A (1 = input, 0 = output)
D3	Group A - Port C, upper half (1 = input, 0 = output)
D2	Group B - mode selection (0 = mode 0, 1 = mode 1)
D1	Group B - Port B (1 = input, 0 = output)

D0 (LSB)	Group B - Port C, lower half (1 = input, 0 = output)

- We always want mode 0 (basic input/output), not mode 1 (strobed input/output) or mode 2 (strobed, bidirectional bus input/output).

From the information on the "Peek and Poke driver" from ICS Advent:

(extract from "PCDIOB Series_0431231_rev2C.pdf")

The Peek and Poke driver for Windows 95/NT allows developers to write Win32 programs that access hardware I/O ports and physical memory on ICS Advent products. This driver simplifies the testing of hardware components because they can be accessed without using a specific driver for each product.

Caution. The Peek and Poke driver gives application-level access to areas of the hardware and memory that can crash the operating system or corrupt data. Take care to access only known memory or I/O ports.

Under Windows NT, **pplibnt.lib** and **pplibnt.dll** are the drivers (pplibnt.lib for C++ static linking; pplibnt.h is the header).

Note: To make sure that libraries are thread safe, you must use the ics_pp_open() call in your initial thread, before creating any other threads. Then make sure that all other threads are terminated before calling ics_pp_close().

`BOOL ics_pp_open(void)`
Opens the Peek and Poke driver. Returns TRUE if successful. This must be called before any calls are made to other library functions.

`void ics_pp_close(void)`
Closes the driver. Should be called before the application exits.

`void *ics_pp_make_pointer(int page, int length)`
This function allows access to a particular region of physical memory by a Win32 application. *page* is the starting page of the physical memory; *length* is the size of the region in pages. For example, for a pointer to a region of physical memory starting at 0xA000 and 64k long, use `void *ptr = ics_pp+make_pointer(0xA),0x10);` **[note misprint there; need to be v. careful when testing this!]** The pointer can then be treated as a standard C/C++ pointer (see below). **Note:** Be sure to release this memory region back to the system as follows: `ics_pp_release_pointer`.
[Another version of that excerpt:] For example, for a pointer to a region of physical memory starting at 0xA0000 and 64k long: void *ptr = ics_pp_make_pointer (0xA0, 0x10);

`void ics_pp_release_pointer(void *address, int length)`
This function releases a memory mapping made with ics_pp_make_pointer. You must release such pointers back to the system. Failure to do so could affect the way the system runs even after the application has exited. *address* is the address returned by the ics_pp_make_pointer function; *length* is the size of the mapped region in pages.

```
int ics_pp_outp(USHORT port, int data)
USHORT ics_pp_outpw(USHORT port, USHORT data)
ULONG ics_pp_outpl(USHORT port, ULONG data)
```
These functions output *data* to the given *port*. Use ics_pp_outp for byte width, ics_pp_outpw for word width, and ics_pp_outpl for double word width.

```
int ics_pp_inp (USHORT port)
USHORT ics_pp_inpw (USHORT port)
ULONG ics_pp_inpl (USHORT port)
```
These functions return *data* input from the given *port*. Use ics_pp_inp for byte width, ics_pp_inpw for word width, and ics_pp_inpl for double word width.

5.10 National Instruments / Lafayette ABET hardware

For details of physical installation and configuration of National Instruments I/O cards and Lafayette/Campden Instruments ABET hardware, see *Configure hardware / National Instruments / Lafayette ABET hardware*.

5.11 Serial port (COM) devices

For details of serial port device configuration, see *Configure hardware / Serial port (COM) devices*.

5.12 Lafayette CANTAB USB device

For details of Lafayette CANTAB USB device configuration, see *Configure hardware / Lafayette CANTAB USB hardware*.

5.13 Berlin network controller

For details of Berlin network controller configuration, see *Configure hardware / Berlin network I/O controller*.

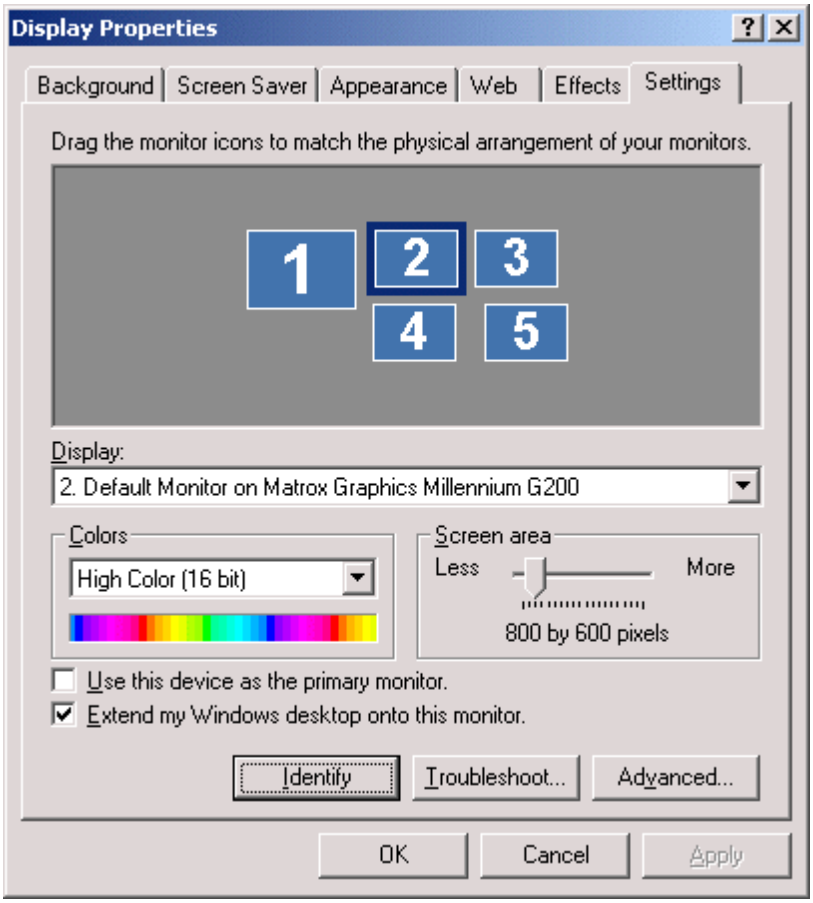
5.14 Sound card installation

Whisker will detect all sound cards that Windows is aware of. For details of sound card configuration, see *Configure hardware* → *Audio devices*.

5.15 Multimonitor installation

Video card installation is not covered by this guide. For details of configuring multiple monitors for use with Whisker, see the menu *Configure hardware* → *Display devices*.

The specific point to note is that within *Control Panel* → *Display* → *Settings*, every monitor you wish to use with Whisker must have 'Extend my Windows desktop onto this monitor' ticked. (This is not ideal, but is a consequence of what appears to be a DirectDraw bug.)



5.16 Touchscreen installation

Whisker supports touchscreens via the Universal Pointing Device Driver (UPDD) from TouchBase (http://www.touch-base.com). It does not support the use of touchscreens with the drivers from touchscreen manufacturers.

Therefore, the installation sequences is:

1. Set any hardware switches on the touchscreen and plug it into your computer.
   - Intasolve Interact 400 touchscreens

2. If your touchscreen is plugged into a serial port, ensure that the port has the correct settings. For example, here's how to configure ports for XON/OFF handshaking.

3. **Configure, enable and calibrate the touchscreen via the UPDD configuration utility**
   - Configuring the UPDD driver (version 2)
   - Configuring the UPDD driver (version 3)

Tip

Note that the touchscreen must be enabled and controlled via UPDD before you configure Whisker.

*Do **not** install drivers that come with the touchscreen - let UPDD control it for you. The touchscreen drivers will usually turn your touchscreen into a **mouse** as far as Windows (and therefore Whisker) is concerned.*

At this stage, you should see all of the touchscreens from the UPDD control panel.

4. Once UPDD communicates successfully with the touchscreen, Whisker will detect it. To check this, choose *Configure hardware* → *Touchscreens*. All touchscreens that UPDD has detected should be listed; if not, check the server event log for errors. Assign touchscreens to display devices.

5. Test display devices: *Display* → *Show test pattern on all displays.* Now touch the touchscreens!

5.16.1 Intasolve Interact 400

This is a serial touchscreen; it is supported by UPDD. Notes as of 16 April 2001.

How do I configure the touchscreen itself?

1. Turn it off.
2. Open it up.
3. Find the block of 8 DIP switches in one corner. Set them as follows:
   - If you are using UPDD v3:
     - If you have a modern touchscreen, **all off except 3.**
     - If you have an older touchscreen with custom Cenes firmware, **all off except 3, 5.**
   - If you are using UPDD v2, **all off except 3, 6.**

Troubleshooting

See Troubleshooting.
In particular, if *Sync errors* appear on the UPDD status screen (see UPDD v2 or UPDD v3), try different combinations of switches 5 and 6.

What do the DIP switches in the touchscreen do?

- The touchscreen has 8 DIP switches, as follows

Switch	Effect
SW1-1	ON origin right, OFF origin left
SW1-2	ON origin bottom, OFF origin top *For UPDD v2, both should be OFF. With UPDD v3, neither SW1 or SW2 are relevant; its calibration procedures determines where the effective origin is.*
SW1-3	ON for XON/OFF (software) handshaking. --- *We want XON/OFF handshaking (see below).* OFF for CTS (hardware) handshaking.
SW1-4	OFF for size report on power-up or reset.
SW1-5 SW1-6	Set as a pair (SW5-SW6): • OFF-OFF for superlabel mode (point, stylus status off, cr/lf after report on). • OFF-ON for touchbase driver mode (continuous mode, stylus status off, CR/LF after report on).

	• ON-OFF for Intasolve windows driver mode (stream mode, stylus status character on, cr/lf after report off). • ON-ON for touch-off mode (operating mode none; stylus status off, CR/LF after report off). *Modern Interact 415 touchscreens need these both OFF to operate with UPDD v3.* *Older versions (particularly those with special versions of the firmware supplied to Cenes / Cambridge Cognition Ltd offering backward compatibility with pre-1990 touchscreens) probably need 5 ON and 6 OFF to work with UPDD v3.* *For UPDD v2, we're not quite sure; we and Touch-Base suspect 5 OFF, 6 ON.*
SW1-7 SW1-8	Set as a pair (SW7-SW8): • OFF-OFF: 9600 bps --- *9600 bps is normally the correct setting.* • OFF-ON: 19200 bps • ON-OFF: 1200 bps • ON-ON: 2400 bps

What is handshaking? [SKIP IF YOU'RE NOT INTERESTED]

- RS232 and similar serial communications systems always require a minimum of three wires: RX (receive), TX (transmit), and GND (ground). The RX pin from one end of the link (e.g. the computer) is connected to the TX pin of the other end (e.g. the touchscreen) and vice versa; the two GND pins are connected together.
- If one end sends data faster than the other end can process it, data can be lost. Handshaking is how serial ports say to each other "hang on a second; don't send data yet, I'm busy". There are different ways of doing this:
  - One is RTS/CTS hardware handshaking. Each end sets its RTS (ready-to-send) pin high when it is ready to receive data. Each end watches its CTS (clear-to-send) pin; if this is high, the other end is ready to receive data. If it isn't high, no data should be sent. The RTS pin from one end should be wired to the CTS pin on the other, and vice versa.
  - An alternative form of hardware handshaking, less commonly used, is DTR/DSR handshaking (see below).
  - The other is XON/XOFF software handshaking. This only uses the standard three wires (RX, TX, GND). Each end may send an XOFF (^S, ASCII 19) signal to say "hang on! I'm busy". The other end should then stop sending data and wait until it receives an XON (^Q, ASCII 17) signal.
- In general, hardware handshaking is better, because it's faster.

Why do we use XON/OFF handshaking? [TECHNICAL]

- The PC and Archimedes 9-pin serial ports have the following pin configurations: 1 DCD, 2 RX, 3 TX, 4 DTR, 5 GND, 6 DSR, 7 RTS, 8 CTS, 9 RI. We've discussed five of these; the others are DCD (data carrier detect; computers listen on this pin and modems set it high to indicate that they have detected a carrier signal from another modem), DTR (data terminal ready; computers set this high to tell modems that they're generally in business; dropping it low instructs modems to change mode, hang up, or something like that), DSR (data set ready; computers listen on this pin and modems can use it to indicate a connection state), and RI (ring indicator; computers listen on this pin and modems use it to indicate that an incoming call is arriving).
- Some Intasolve touchscreens have no internal plugs; others have RJ11/DIN plugs. The touchscreen RJ11 internal connector is 1 DTR (from touchscreen), 2 TX (from touchscreen),

3 GND, 4 GND, 5 RX (from computer), 6 CTS (from computer). This is connected directly to a 9-pin DIN plug: 1 DTR, 2 TX, 3 RX, 4 CTS, 5 GND.

- CTS handshaking is reported [by MRFA] not to work with UPDD v2 and multiport serial card. This is/was a UPDD bug, as QModem communicates fine in this situation.
- Many of our touchscreens came from the manufacturer/supplier with mis-wired cables as shown below. The mis-wiring prevents proper hardware handshaking.

```
  PC/Arc     Touchscreen
  1 DCD    < DTR 1
  2 RX     < TX  2
  3 TX     > RX  5
  4 DTR    > CTS 6
  5 GND    - GND 4
  6 DSR
+-7 RTS
+-8 CTS
  9 RI
```

Not ideal. The RX/TX/GND connections are fine (note that < and > indicate the direction of information flow). The touchscreen raises DTR and the PC can detect this with its DCD pin, but no PC uses DCD as a handshaking pin, so this wiring assumes that the touchscreen can always process data from the PC as fast as the PC can send it. The PC will probably hold DTR high all the time (it would need to be in DTR/DSR handshaking mode to use DTR as a handshaking pin, and this is not a very common mode); therefore, the touchscreen will effectively assume that the PC can process data as fast as the touchscreen can send it. The RTS and CTS pins are soldered together on the computer end of the cable, meaning that this form of handshaking can't be used. Result: no hardware handshaking.

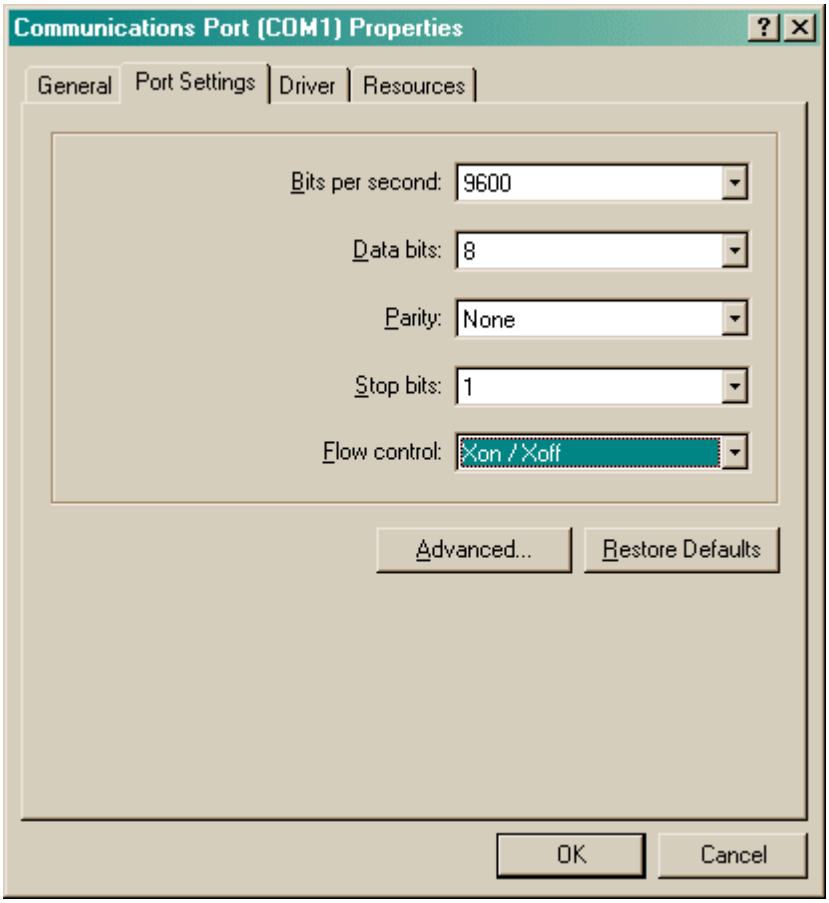
```
     PC/Arc
Touchscreen
     1 DCD    <
     2 RX     < TX
     3 TX     > RX
     4 DTR    >
     5 GND    - GND
     6 DSR    <
   +-7 RTS    > CTS
   +-8 CTS    < DTR
     9 RI     <
```

Better; still not perfect. This would do RTS/CTS handshaking (assuming the touchscreen uses its DTR pin like an RTS pin) if the computer's RTS and CTS pins were not soldered together!

- Better XON/XOFF than no handshaking at all!

How do I tell the computer that it's to use XON/XOFF handshaking?

- In addition to setting the DIP switches on the touchscreen for XON/OFF handshaking, you also need to configure your serial (COM) port for XON/OFF. Click *Start* → *Settings* → *Control Panel* → *System* → *Hardware* → *Device Manager*. In the device tree view, find the appropriate port (e.g. *Communications Port (COM1)*) under *Ports (COM & LPT)*. Right-click it and choose *Properties*.



Any other known problems?

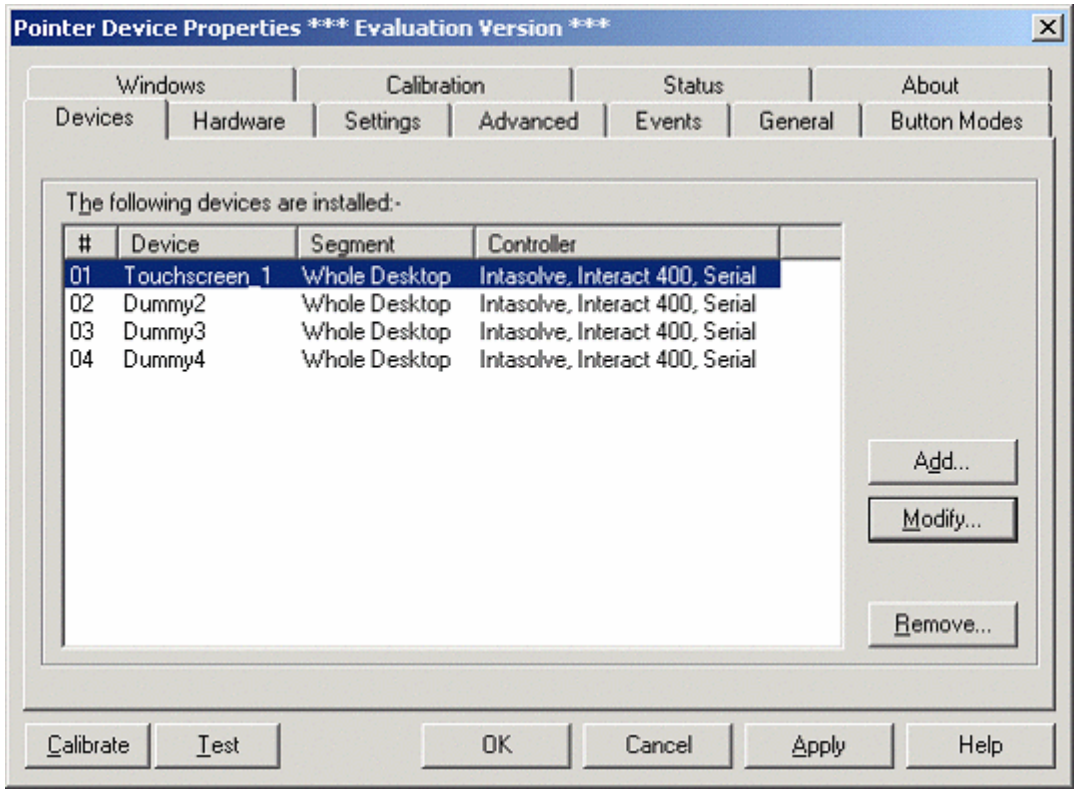
- With a BrainBox multiport serial card, UPDD v2 fails to recognize the extra serial ports once it's been installed, so you must select the touchscreen as you install UPDD (the setup program does recognize all ports). (UPDD v2 bug. Touchscreen was on COM3. COM4 and higher not recognized once installed.)

Now see

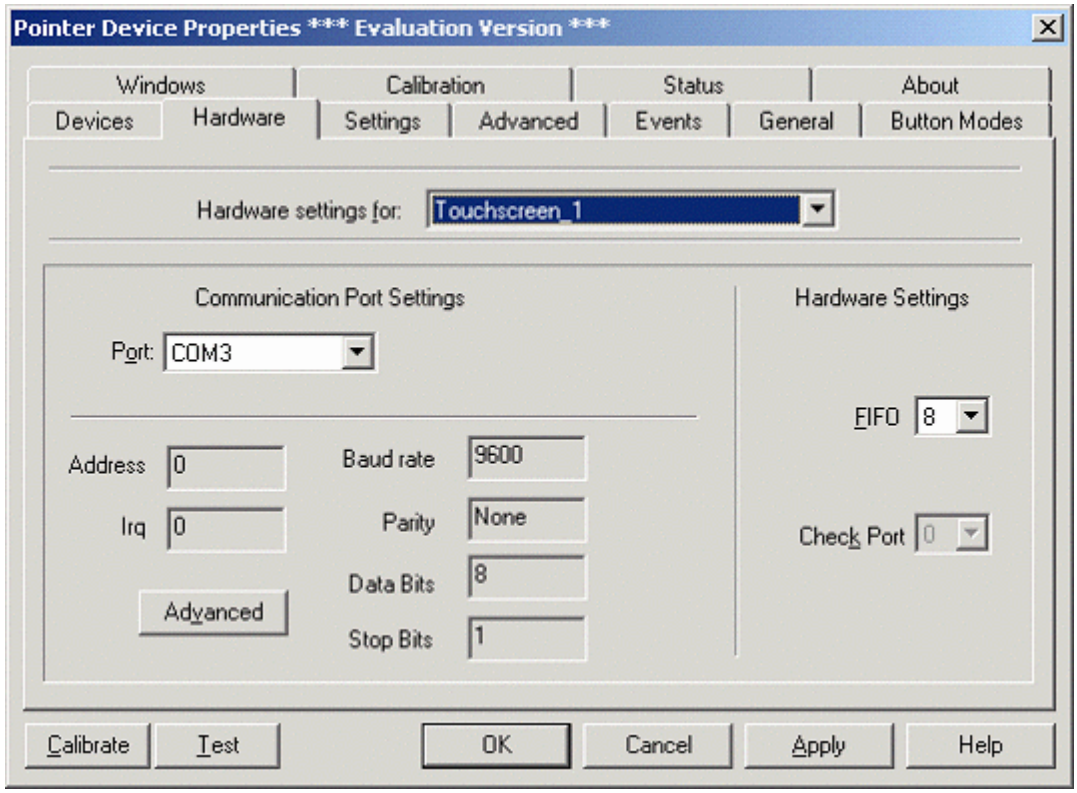
- Configuring the UPDD driver

5.16.2 Configuring UPDD version 2 touchscreen drivers

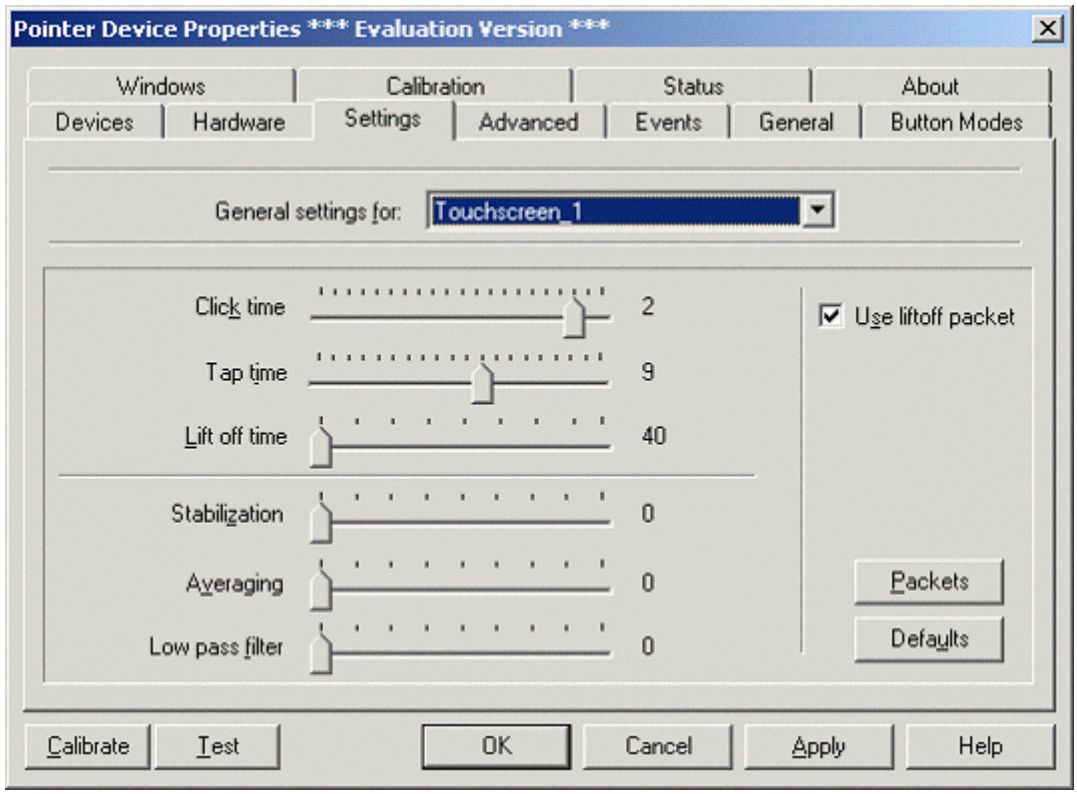
The UPDD configuration utility has the following windows:

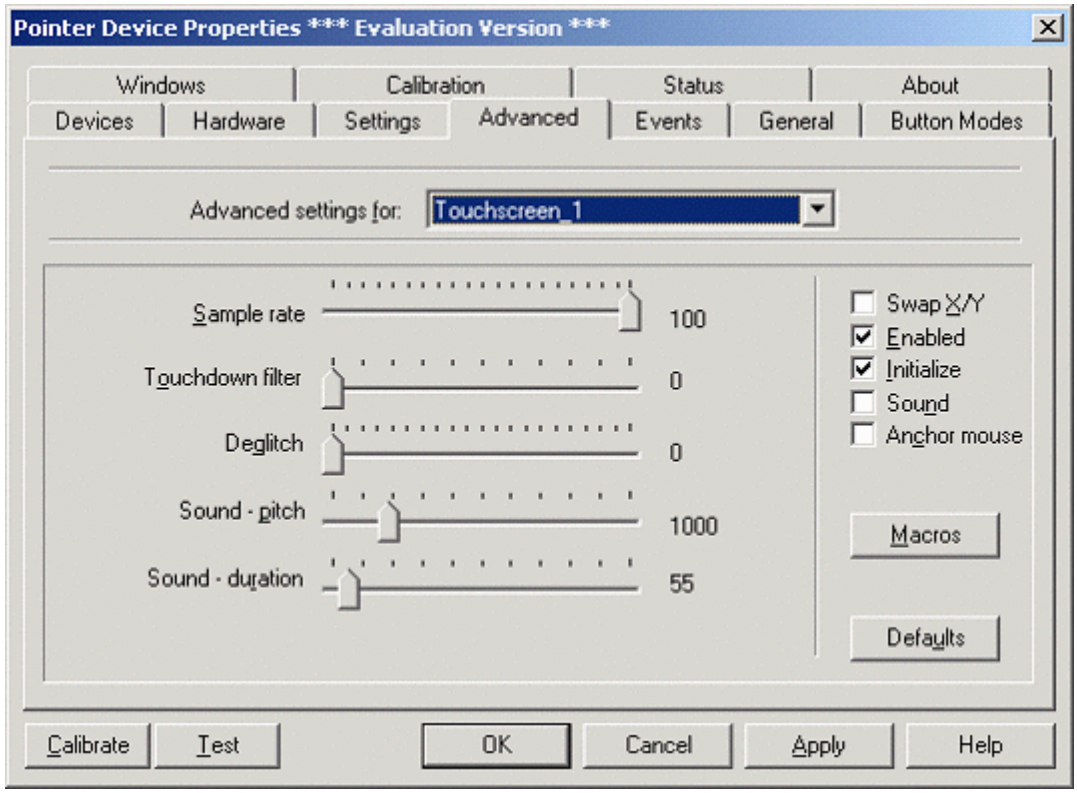


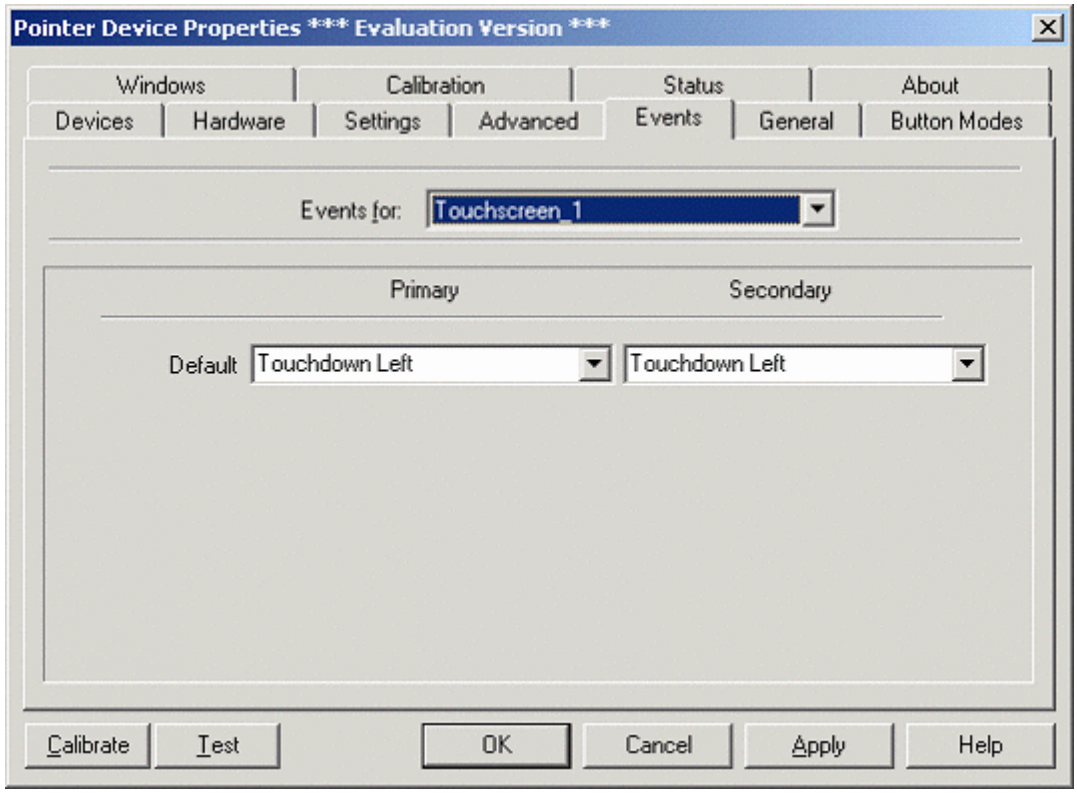
As the UPDD version 2 drivers don't support multiple monitors properly, set the touchscreen to 'Whole Desktop'. (Touchscreen_1 is the genuine touchscreen in this example.)



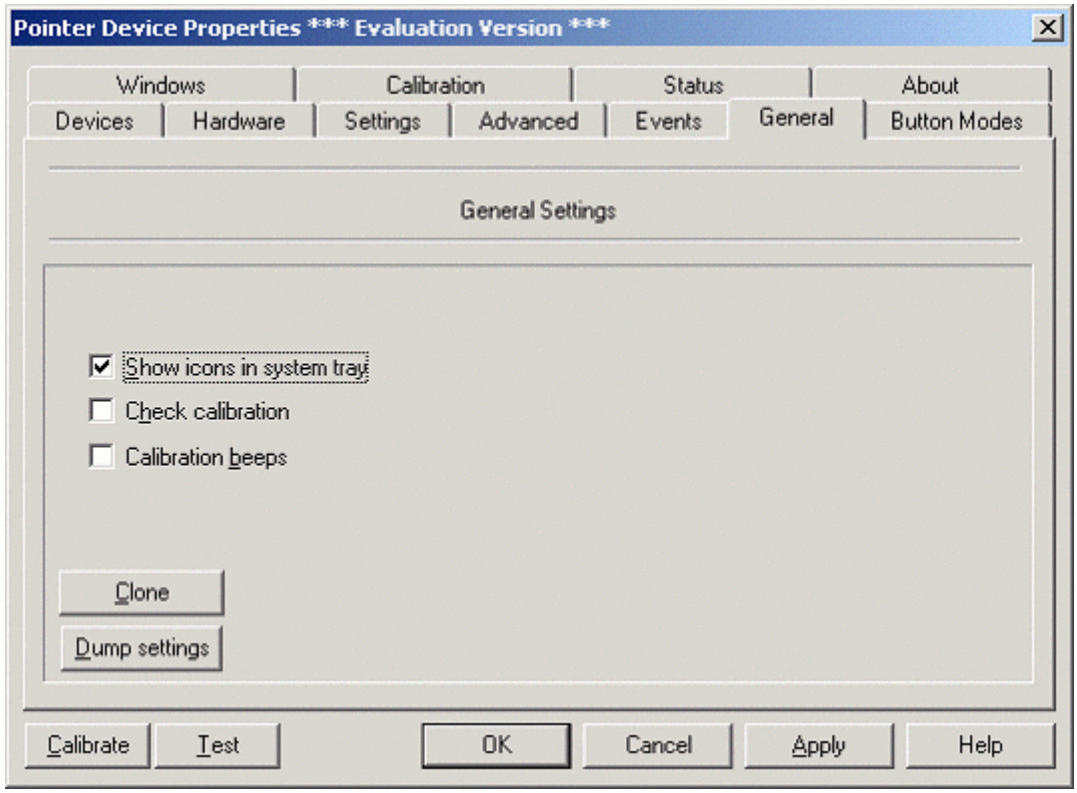
COM port settings should match the hardware configuration.

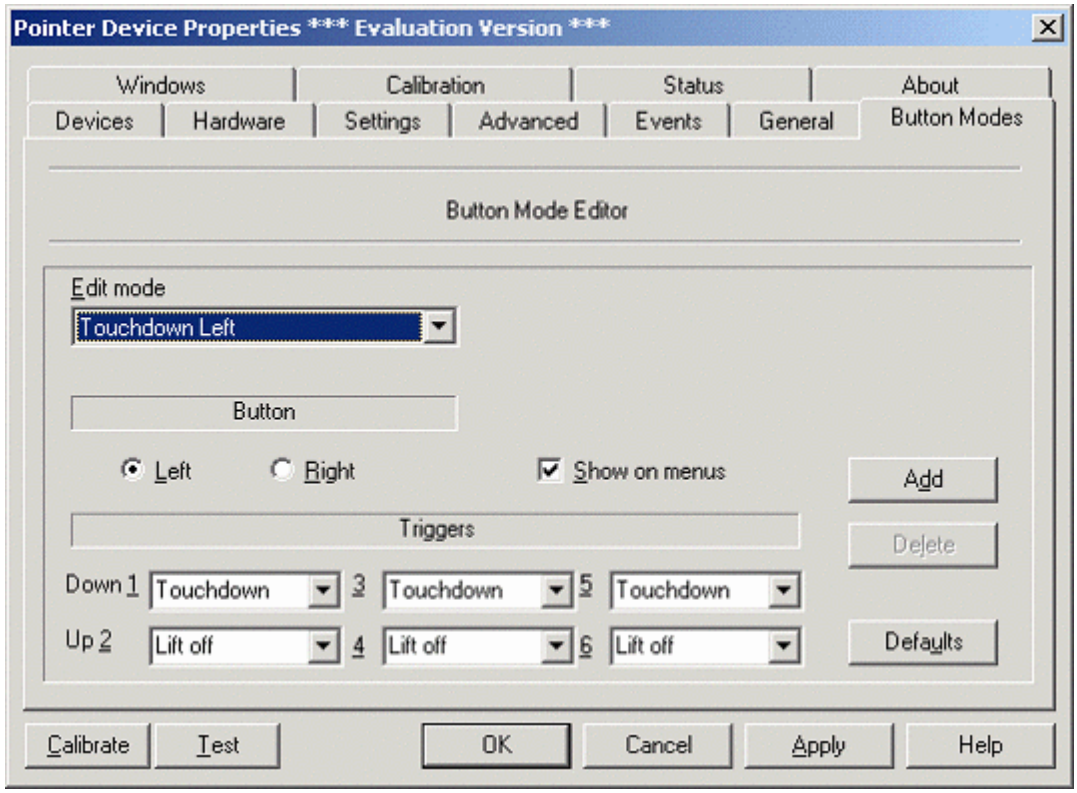


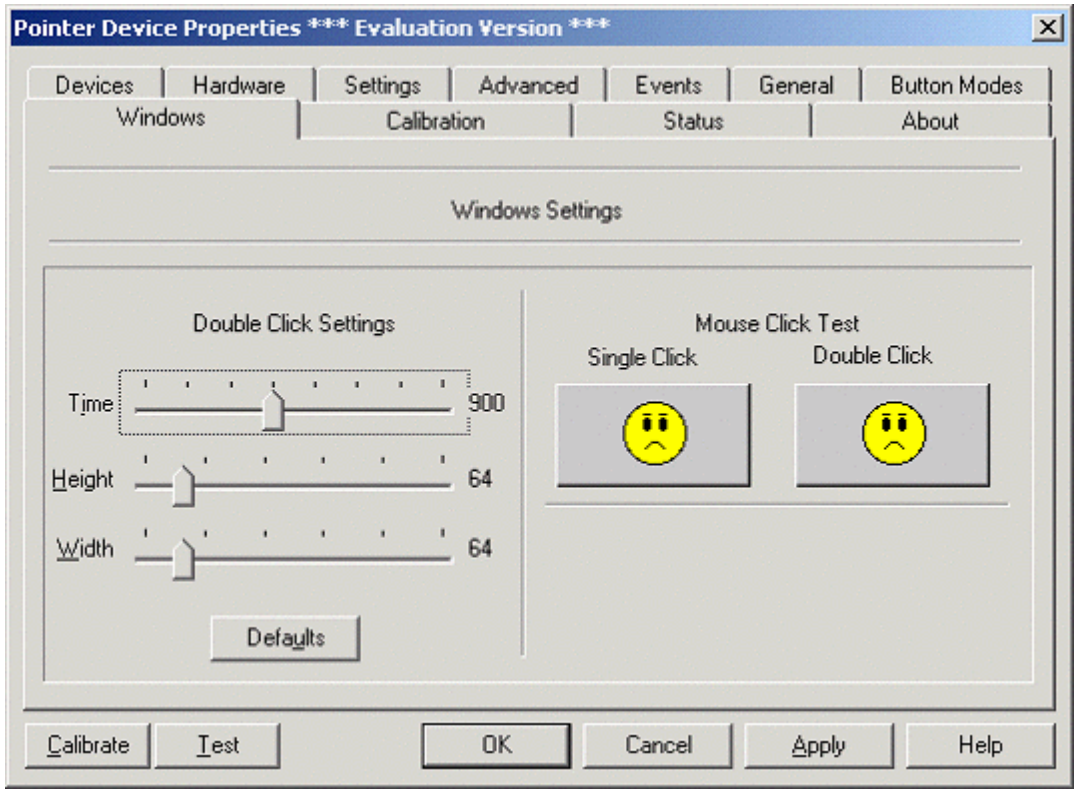


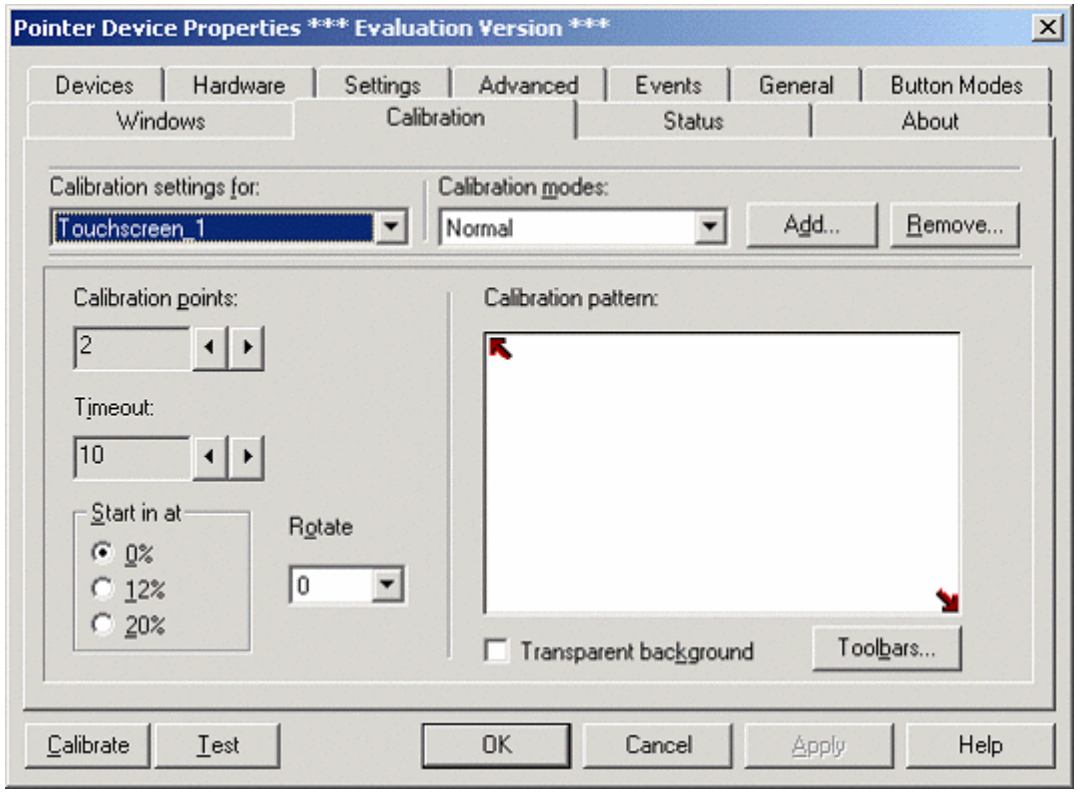


These settings are relevant. You probably don't want anything too fancy from your touchscreen; detecting when a finger (or nose) is on or off the screen is probably simplest and best. So the primary and secondary event should simply be 'Touchdown Left'.

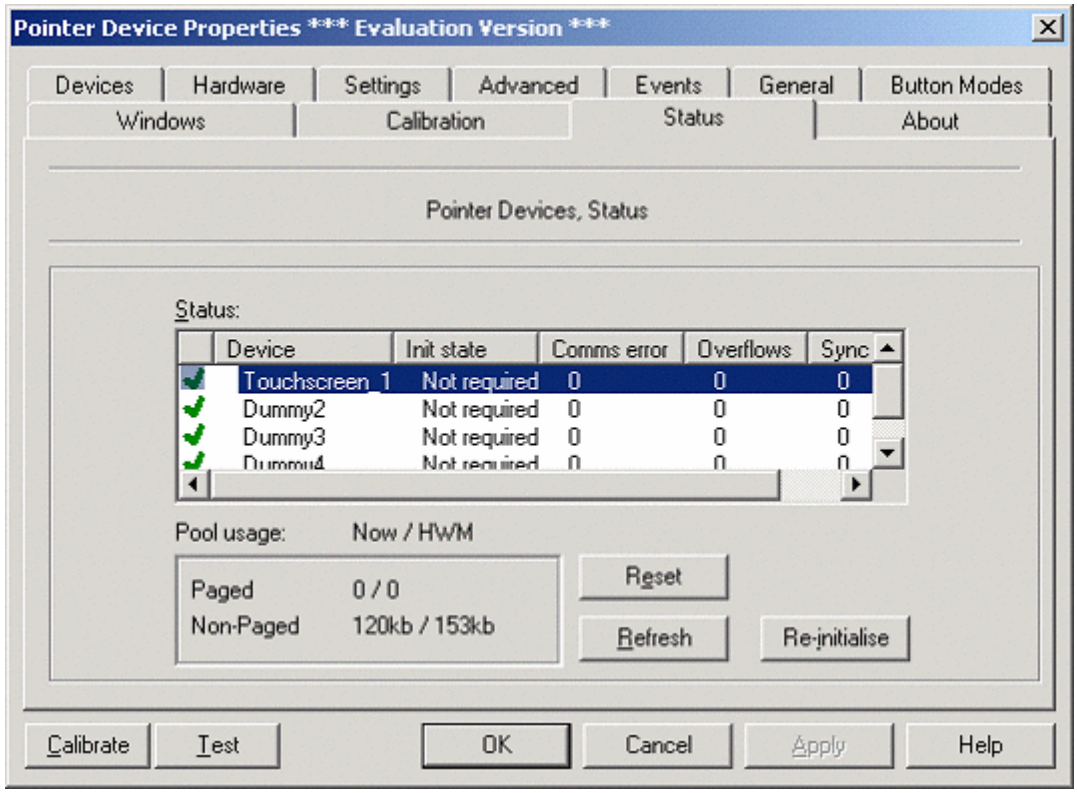




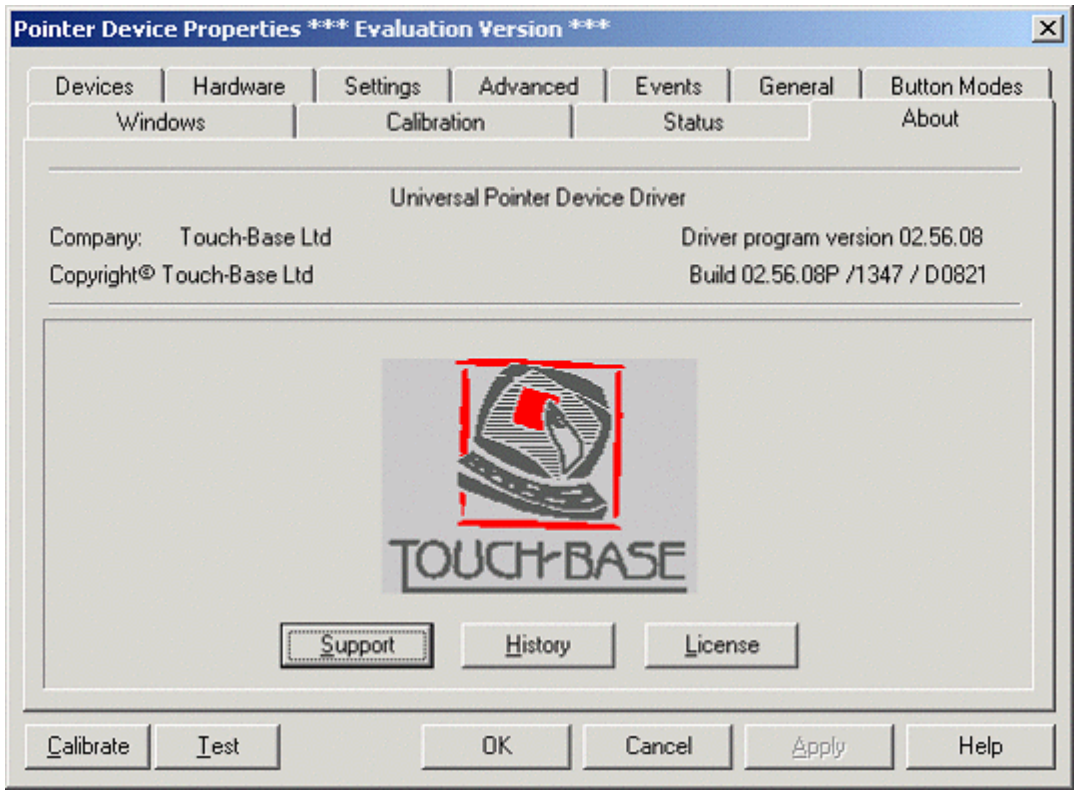




You will need to calibrate your touchscreen. Once you've configured it for the 'whole desktop', come here and click 'Calibrate'. You'll be presented (on the primary monitor) with a white screen with arrows on; touch the arrows as it asks to calibrate the touchscreen.

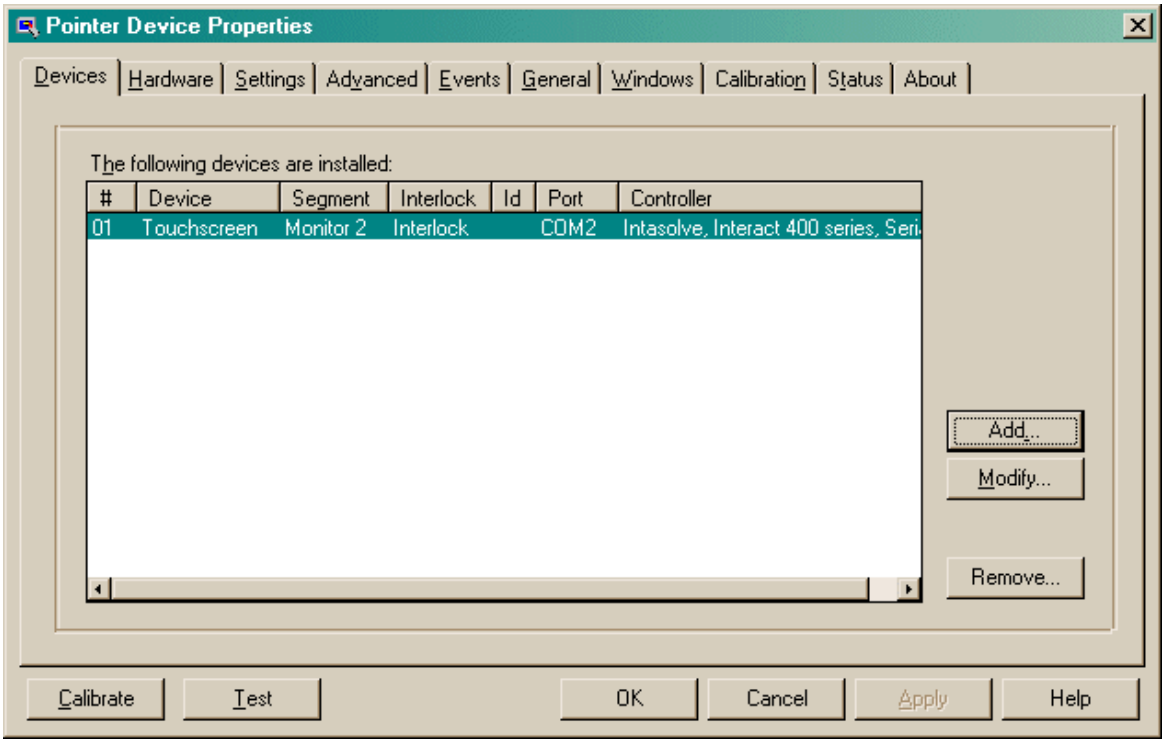


Though it would be nice if the 'reset' and 're-initialize' buttons did something useful – like getting rid of the infernal beeping that comes out of the UPDD driver every so often and has no effect on the touchscreen's function – they don't.

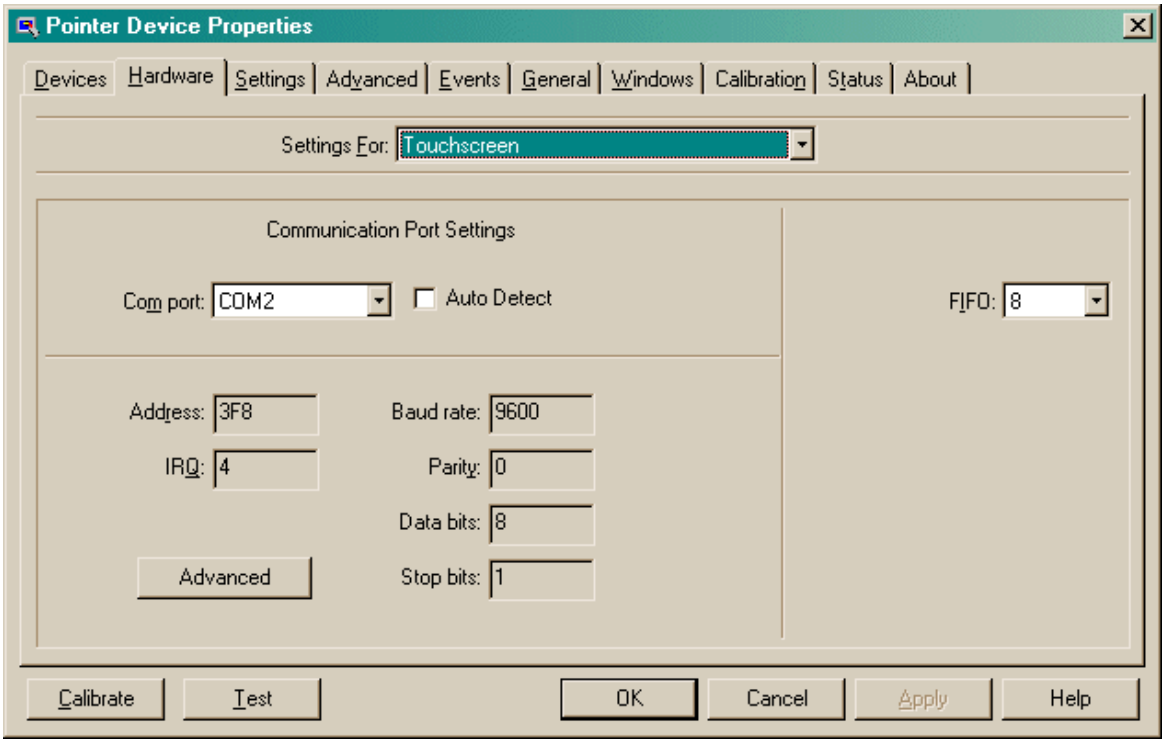


5.16.3 Configuring UPDD version 3 touchscreen drivers

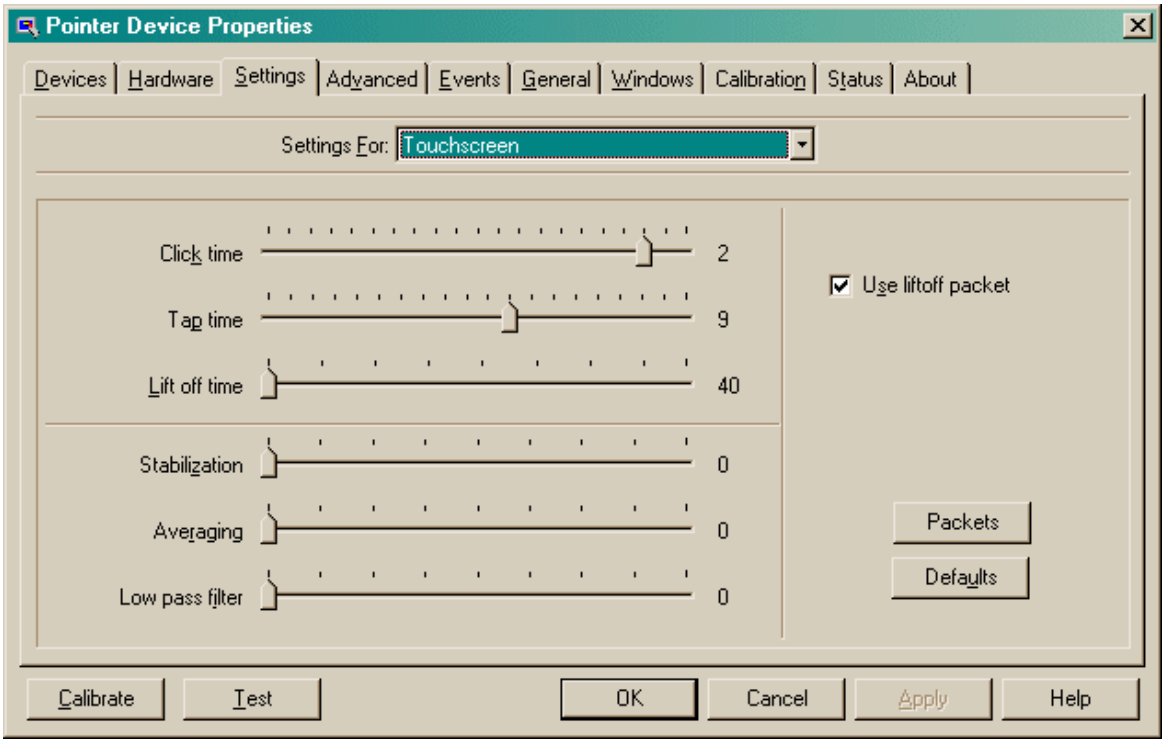
Begin by choosing *Start* → *Programs* → *UPDD* → *Settings*. All of the UPDD can be controlled from this panel.



Click *Modify* to choose which area of the desktop the touchscreen should control - or which monitor (in a multimonitor configuration). You can also choose the touchscreen's name here. In this example, it's imaginatively called "Touchscreen" and this name appears in subsequent configuration screens.

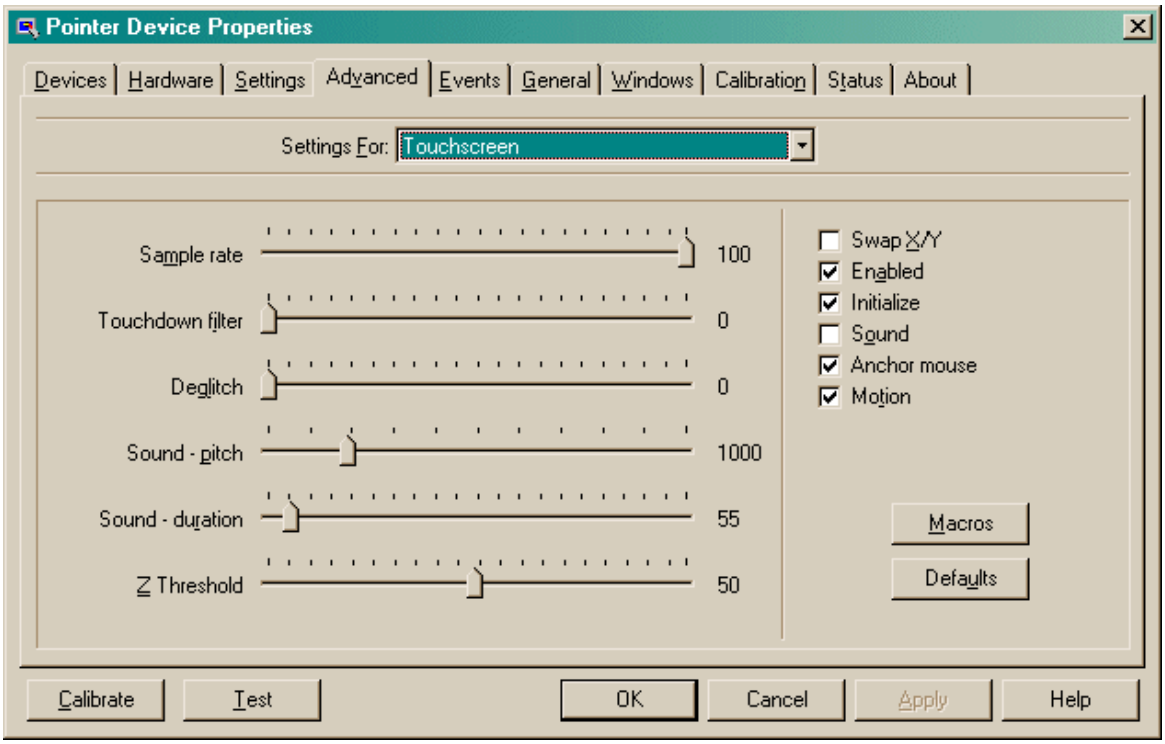


Choose the COM port settings for the touchscreen here. Typically, for serial touchscreens, this is 9600 8N1 (baud rate 9600 bps, 8 data bits, no parity, 1 stop bit) - but the correct value is that which (1) your touchscreen and (2) the Windows COM port are set to.

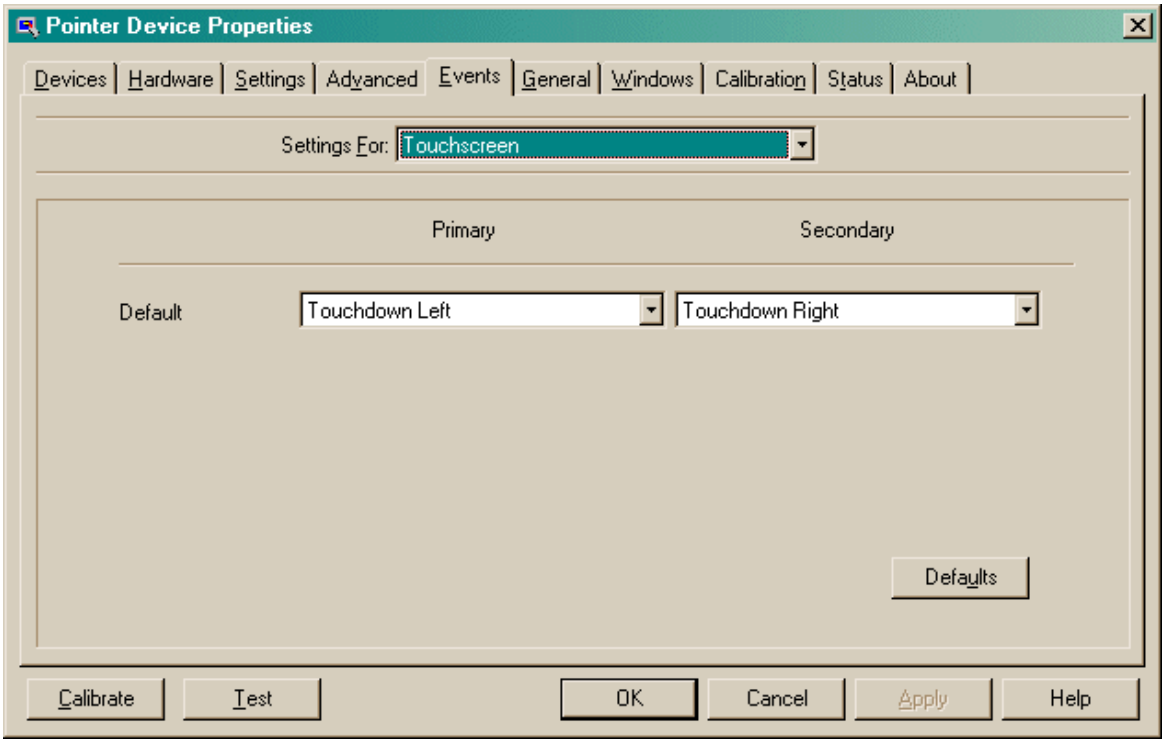


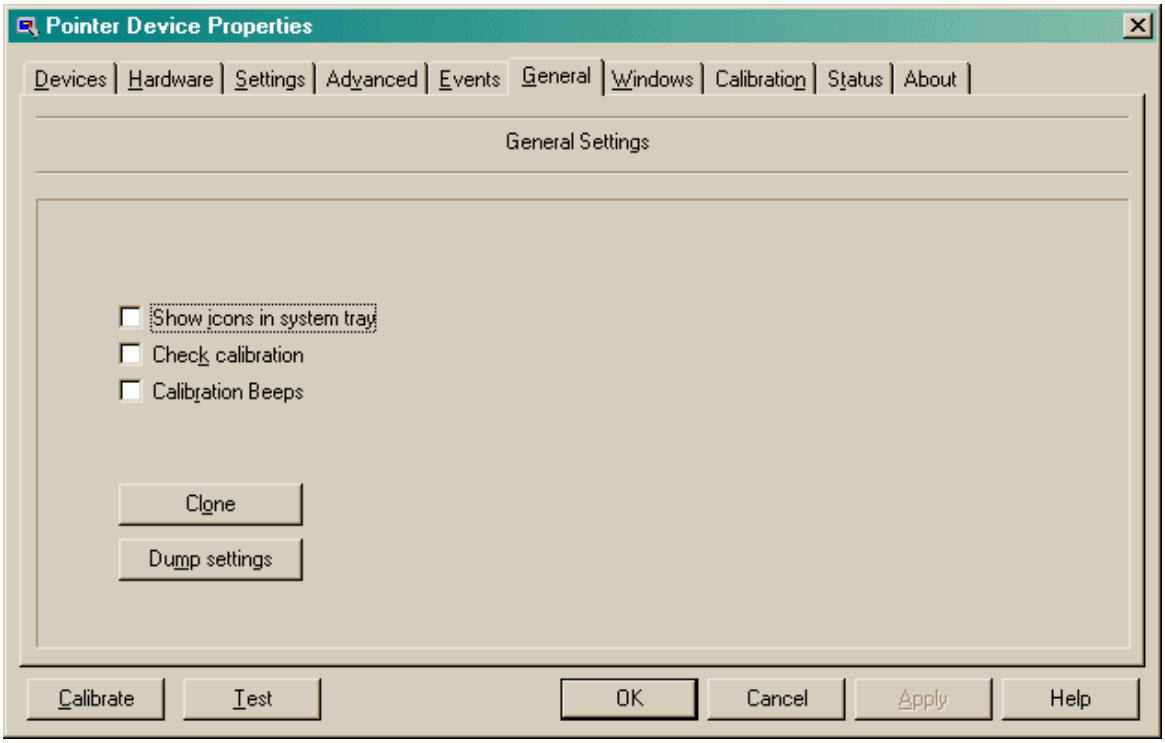
If "Use liftoff packet" is ticked, the touchscreen tells UPDD as soon as the finger is removed from the touchscreen. If it is not ticked, the "Lift off time" becomes important; this is specified in units of 20 ms. If you need to specify a lift-off time, you may wish to consider a value of 2 (i.e. 40 ms) rather than the very high value of 40 (= 800 ms) shown here; see Troubleshooting for an

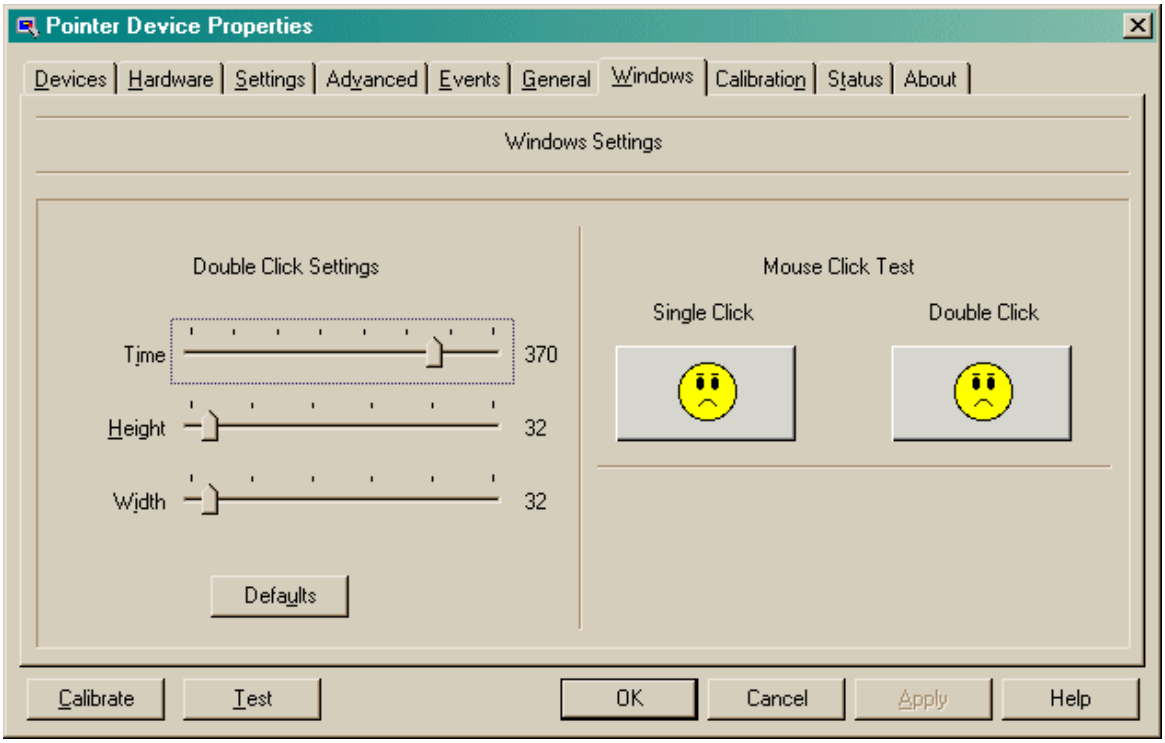
explanation of what can happen if this is misconfigured. Basically, if this value is too high, your touches appear abnormally prolonged. To repeat, the value of "lift off time" is not relevant if your touchscreen allows you to operate with "Use liftoff packet" ticked.

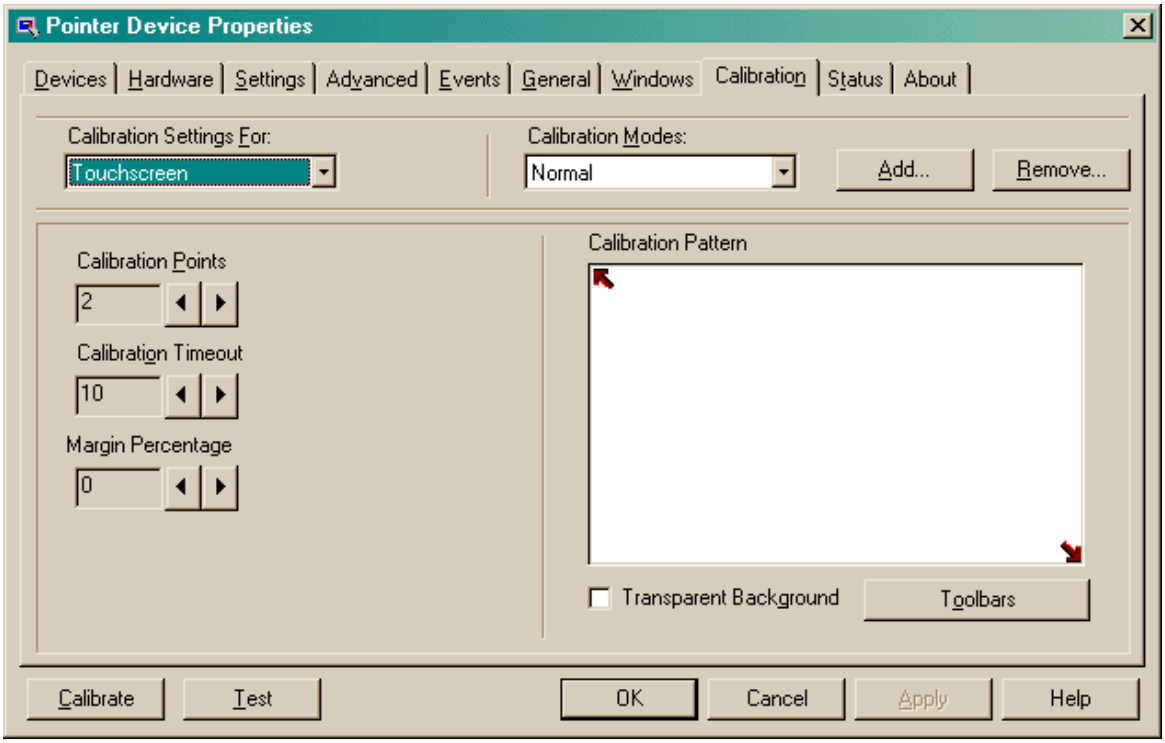


Set *Anchor mouse* on - otherwise, your subject will move the main Windows mouse pointer when it touches the touchscreen!

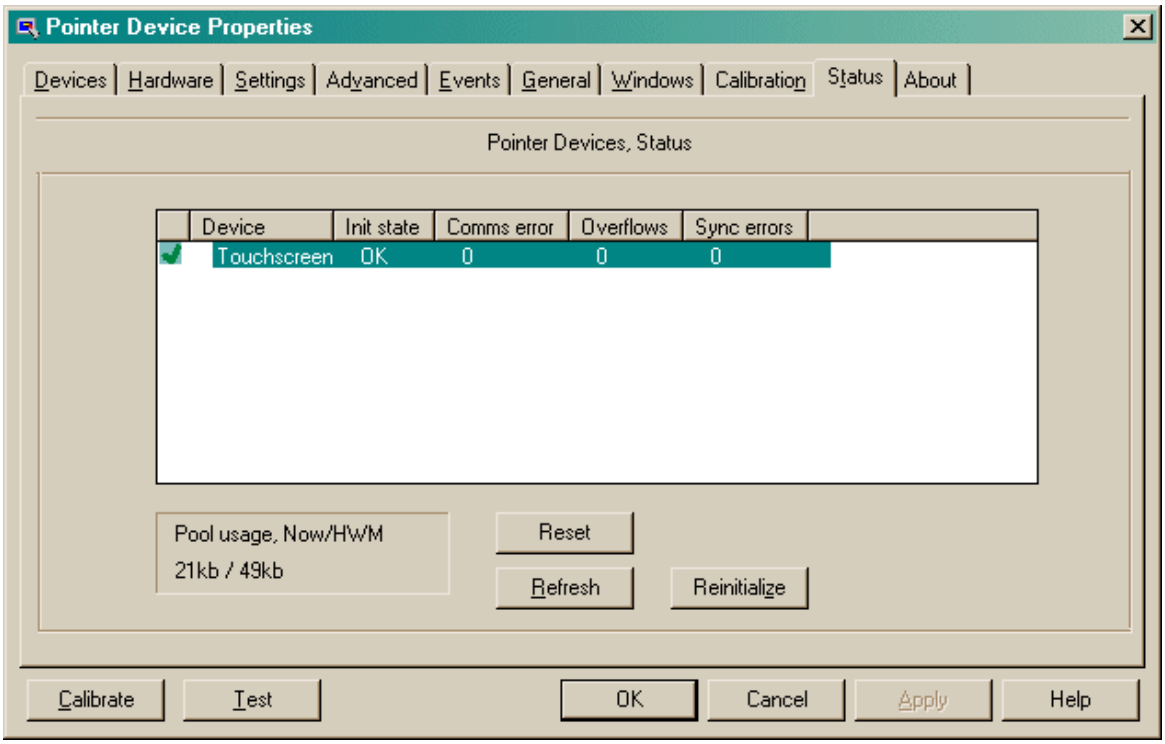




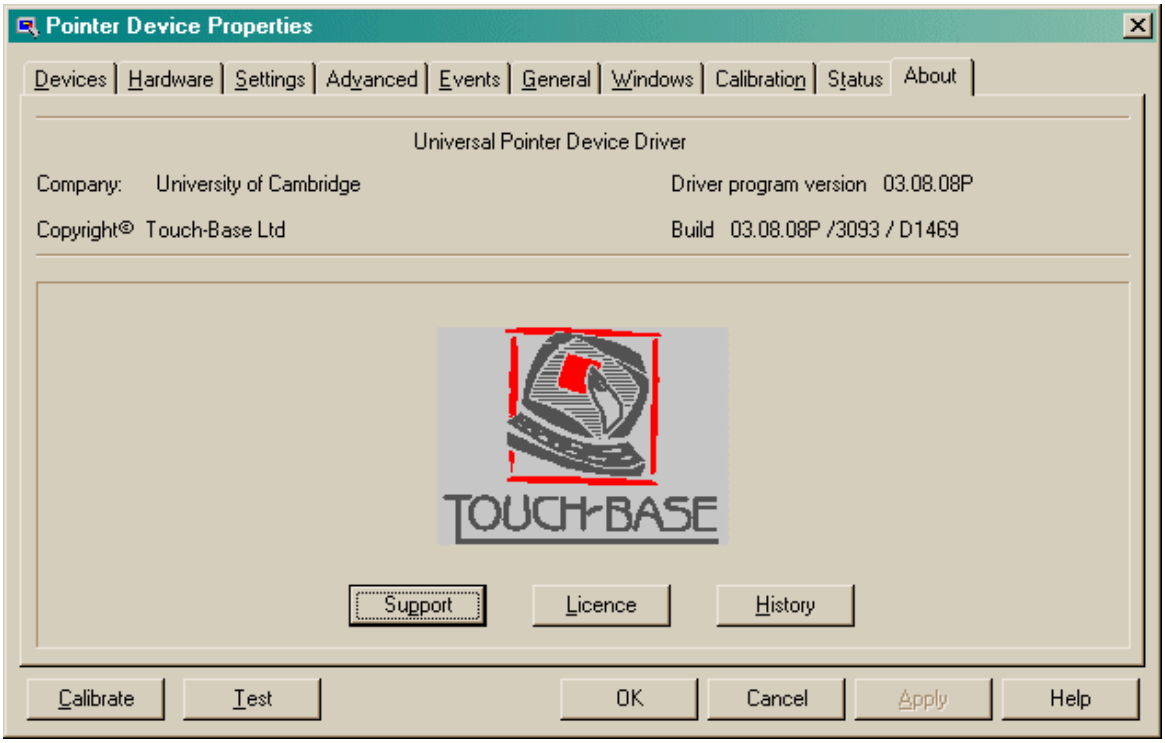




A suggestion: set the *Margin Percentage* to zero, then calibrate your touchscreen (after clicking *Calibrate*) by touching the extreme corners of the touchscreen.



In this status view, there should be none, or very few, comms / overflow / sync errors. (*Hint: if your Intasolve touchscreen has its DIP switches set to Touchbase mode, then many sync errors appear when you touch it, and it doesn't work properly.)*



Technical note: UPDD may store its parameters in the registry at

\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TBUPDD\Parameters

with other copies in

\HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\TBUPDD\Parameters
\HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\TBUPDD\Parameters
... etc.

Clicking the "Dump Settings" button writes the settings to a textfile.

5.16.4 Configuring UPDD version 4.1.10 touchscreen drivers

This section is taken from a working system running Windows 7 Professional SP1 with UPDD v4.01.10 (v4.1.10) in Jan 2015.

Overview of suggestions for Whisker:

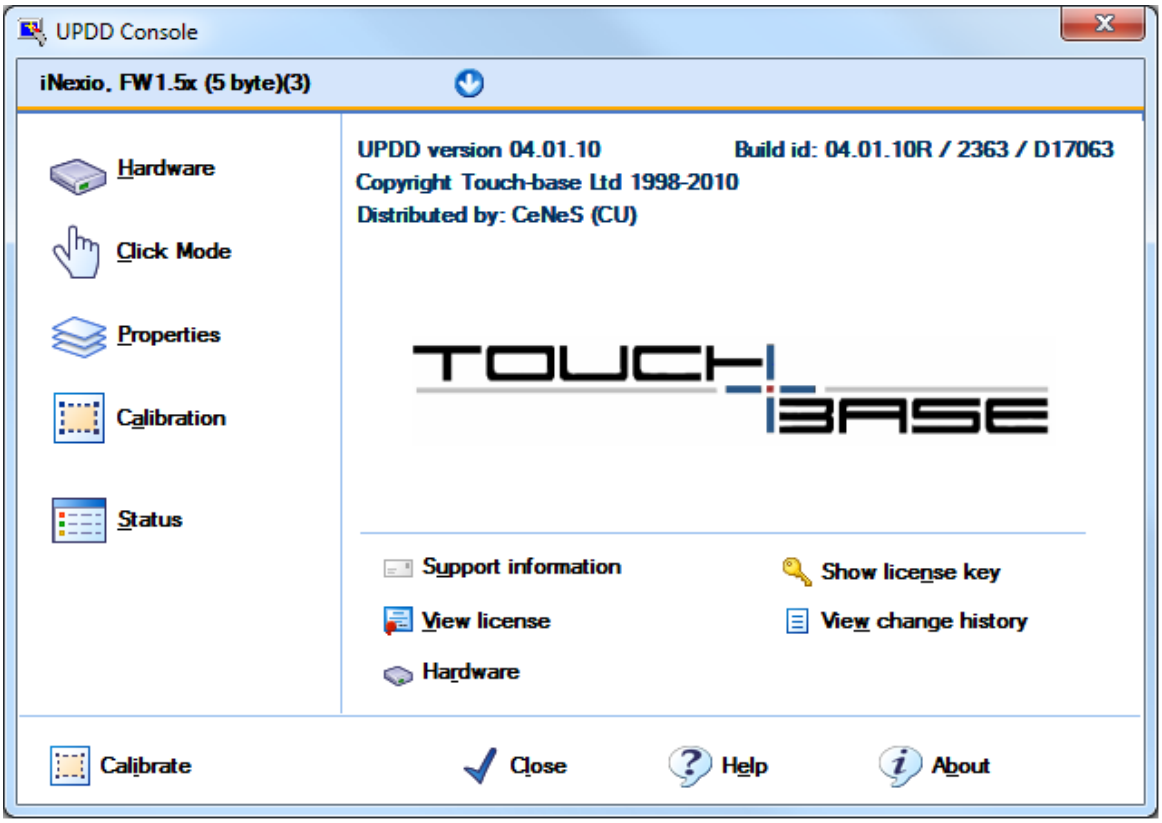
- **Until UPDD is working properly, and communicating with your touchscreen, it WON'T work properly with Whisker.**
- **You'll know that UPDD is working when you can (a) calibrate, and (b) use the UPDD "Show test screen" to draw on the correct screen with your finger.** The mouse pointer may move at the same time, but that's OK; Whisker will prevent that when it is properly configured.
- **Only then run and configure Whisker.** Ensure that Whisker's server event log indicates that it's communicating with UPDD. Ensure that your touchscreen appears in Whisker's Configure → Touchscreens list. Ensure that you point it to the correct display. Then turn on a Whisker test display and ensure that "touch down", "touch move", and "touch up" messages appear when you touch (**not** "mouse down", "mouse move", and "mouse up" messages). Don't

enable any of Whisker's "mouse pretends to be a touchscreen" options at this point, or things will get confusing.

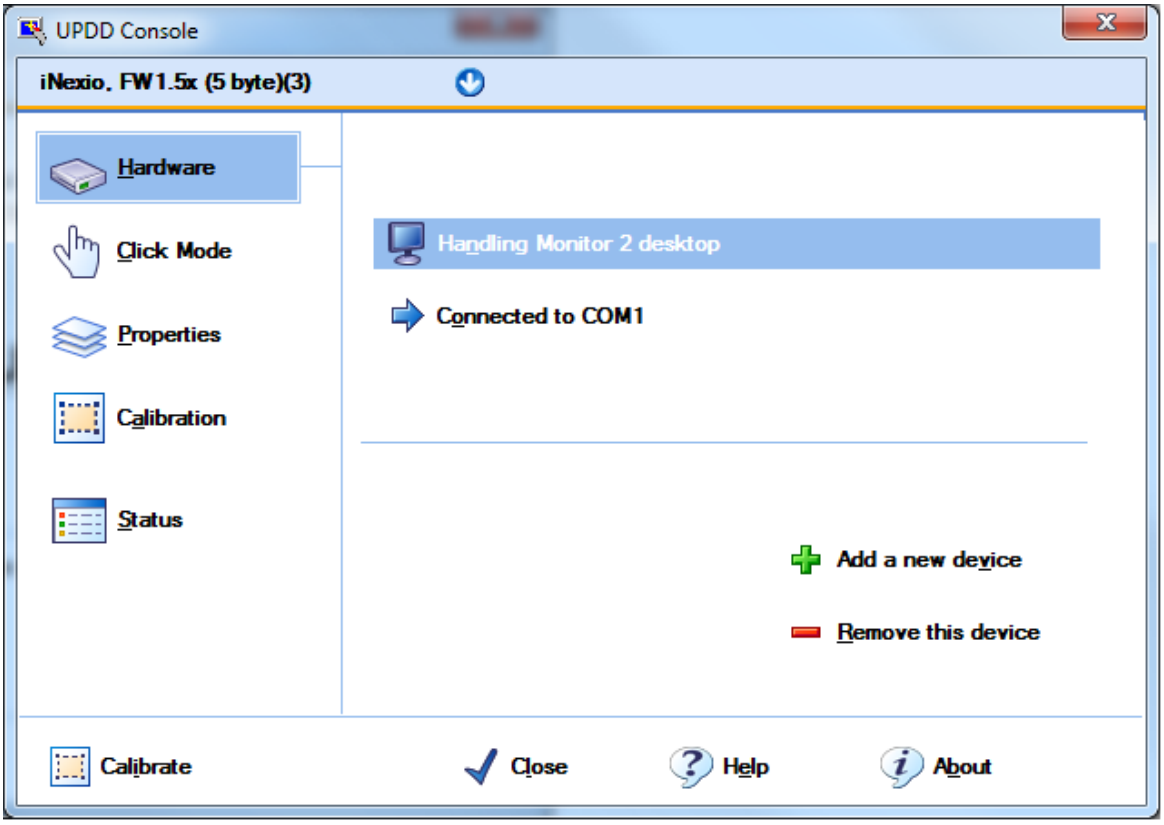
- **Also beware the halfway-house situation where your UPDD touchscreen is pretending to be the system mouse (as it readily does), and you've configured Whisker to respond to mouse input rather than properly configuring its touchscreens.** Then Whisker will appear to "half work", but the mouse pointer will keep jumping all over the place. When Whisker is talking to UPDD properly and has control of its touchscreens, the system mouse pointer is entirely unaffected by touchscreen touches.

Begin by choosing *Start* → *All Programs* → *UPDD* → *Settings*. All of UPDD can be controlled from this panel.

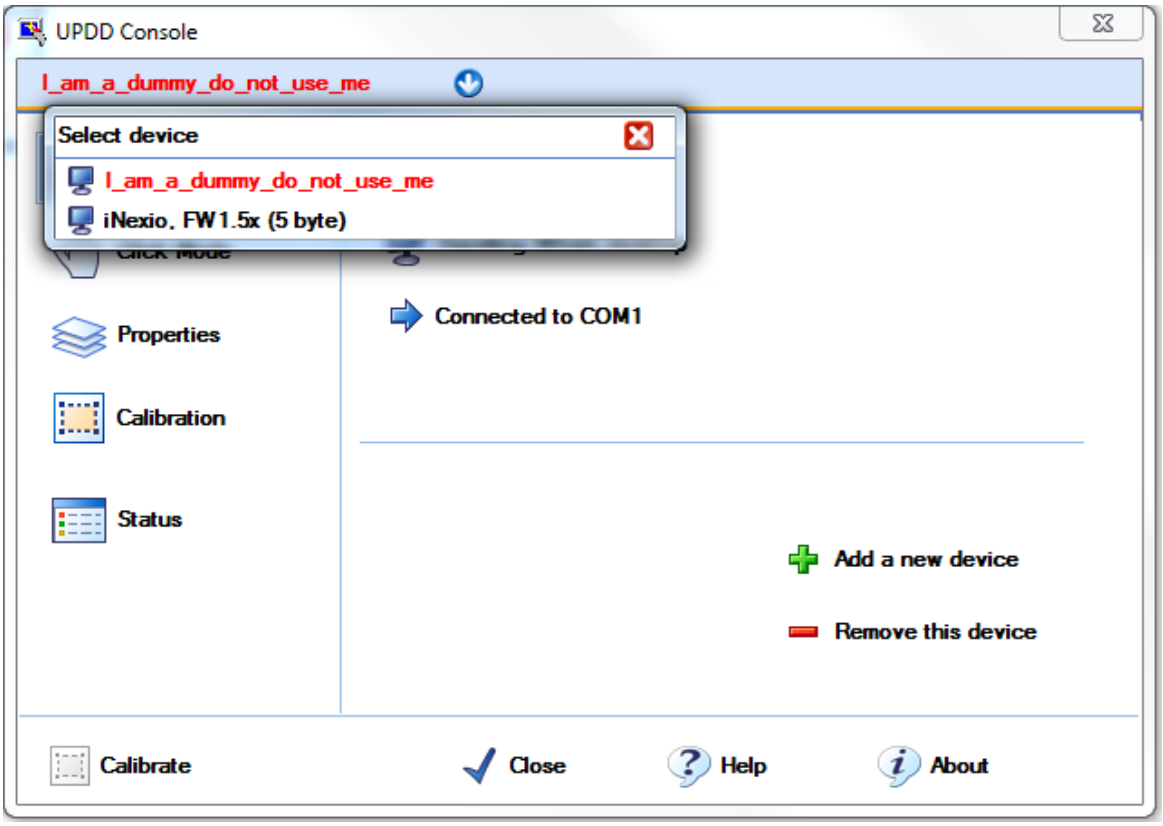
When you click **About,** you see this:



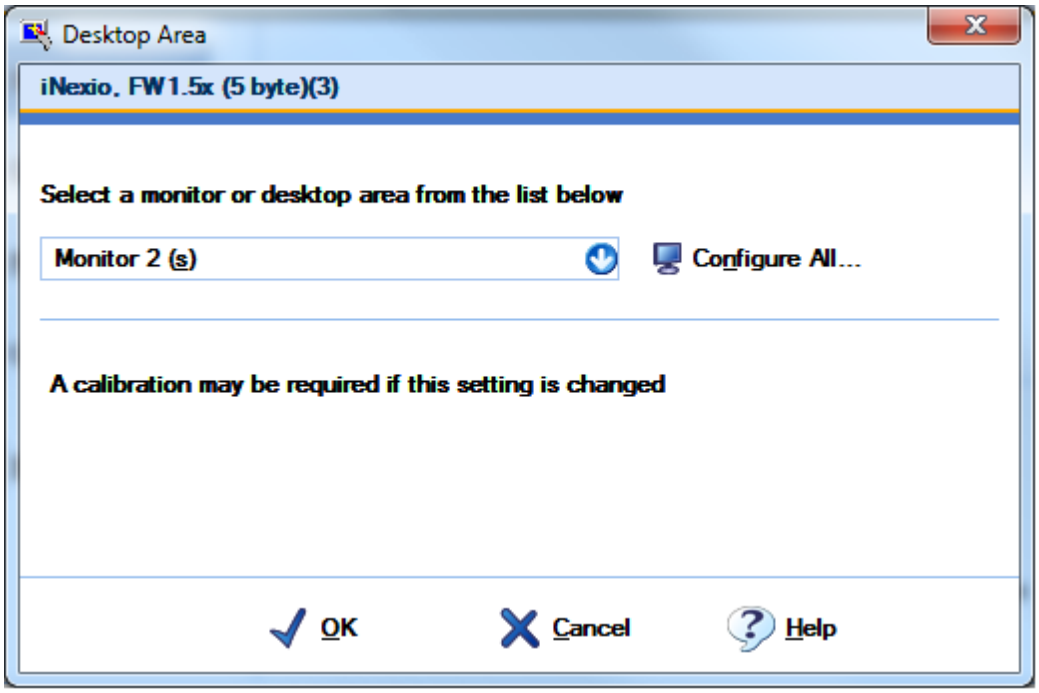
When UPDD Settings launches, it starts at the following screen. You can add and remove devices here.



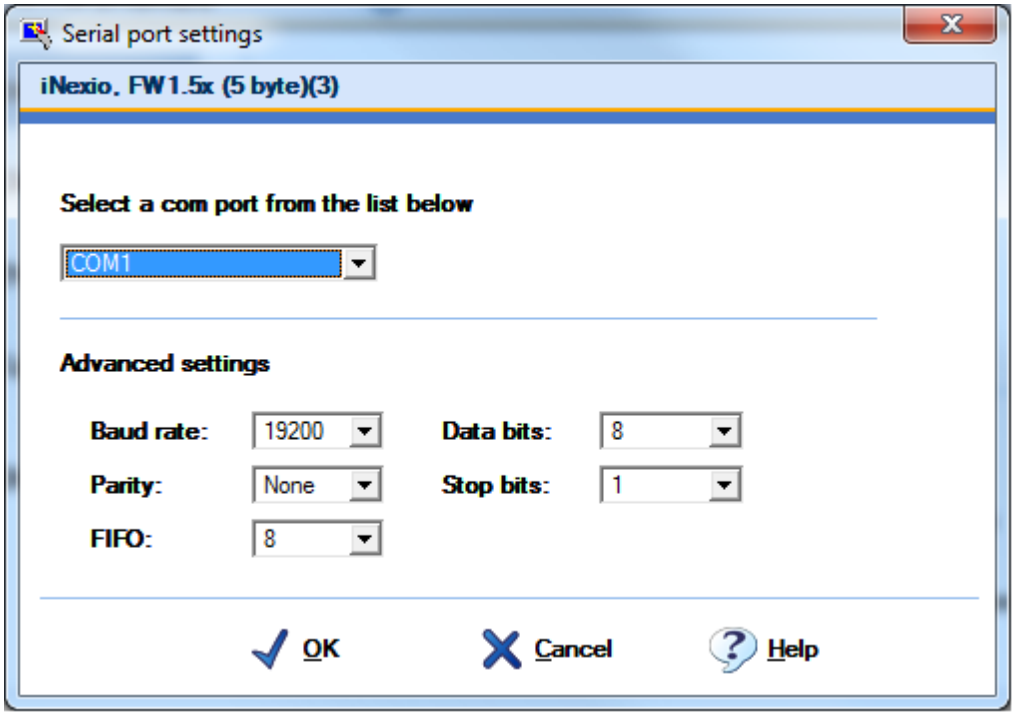
Be extremely careful if you have >1 touchscreen, or have created >1 UPDD device! At the top left, the touchscreen name is shown. Ensure you're configuring the right one!



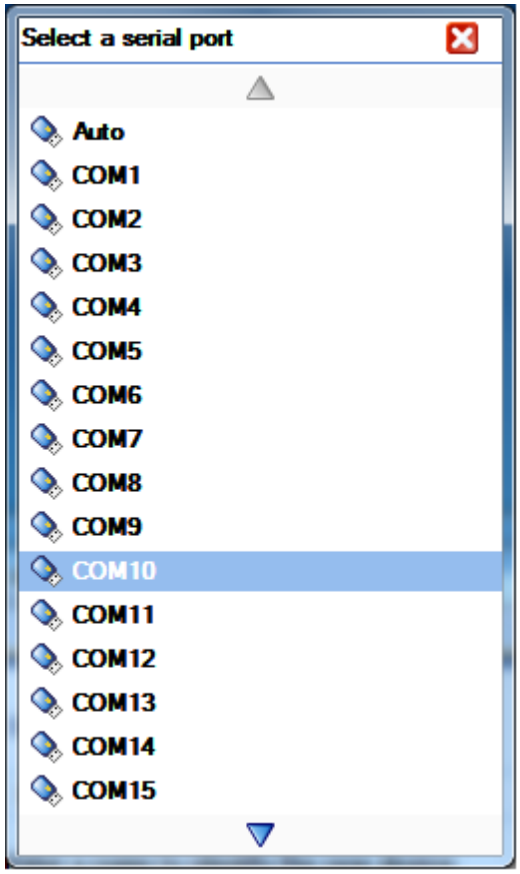
From the Hardware screen, you can configure the monitor:



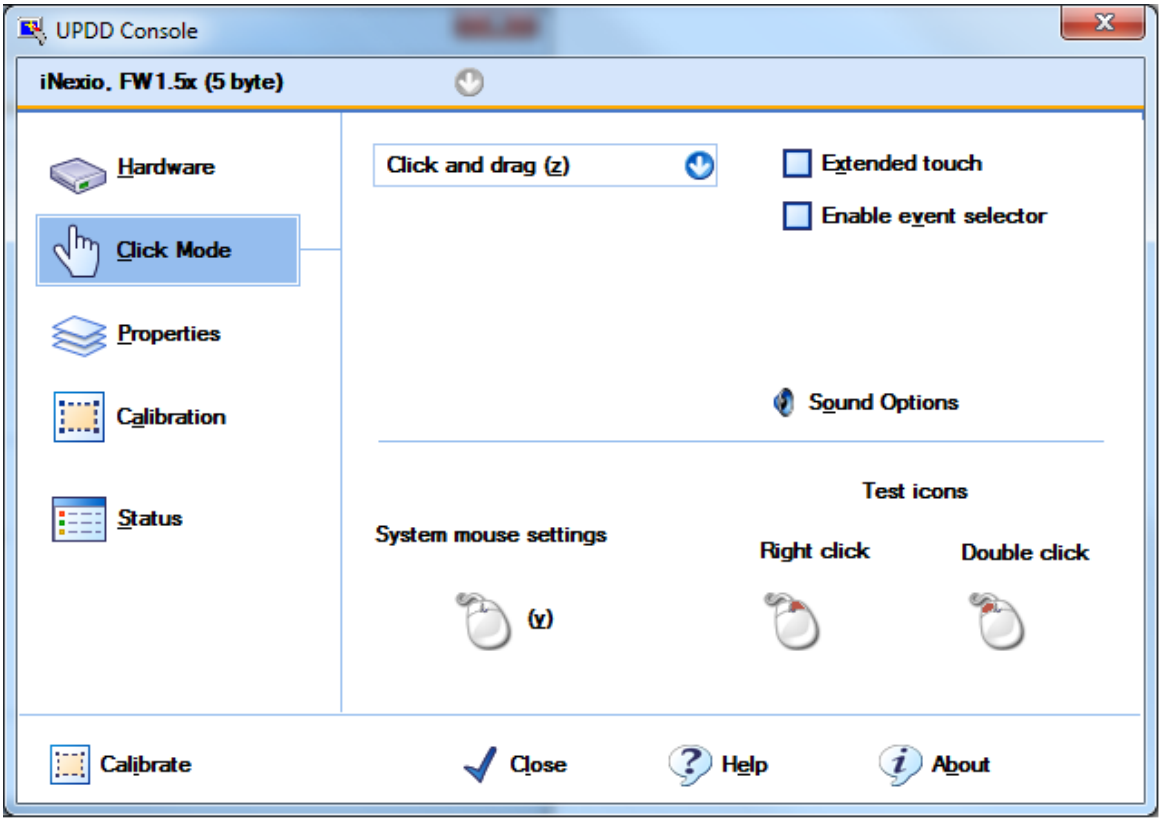
... and the COM port settings:



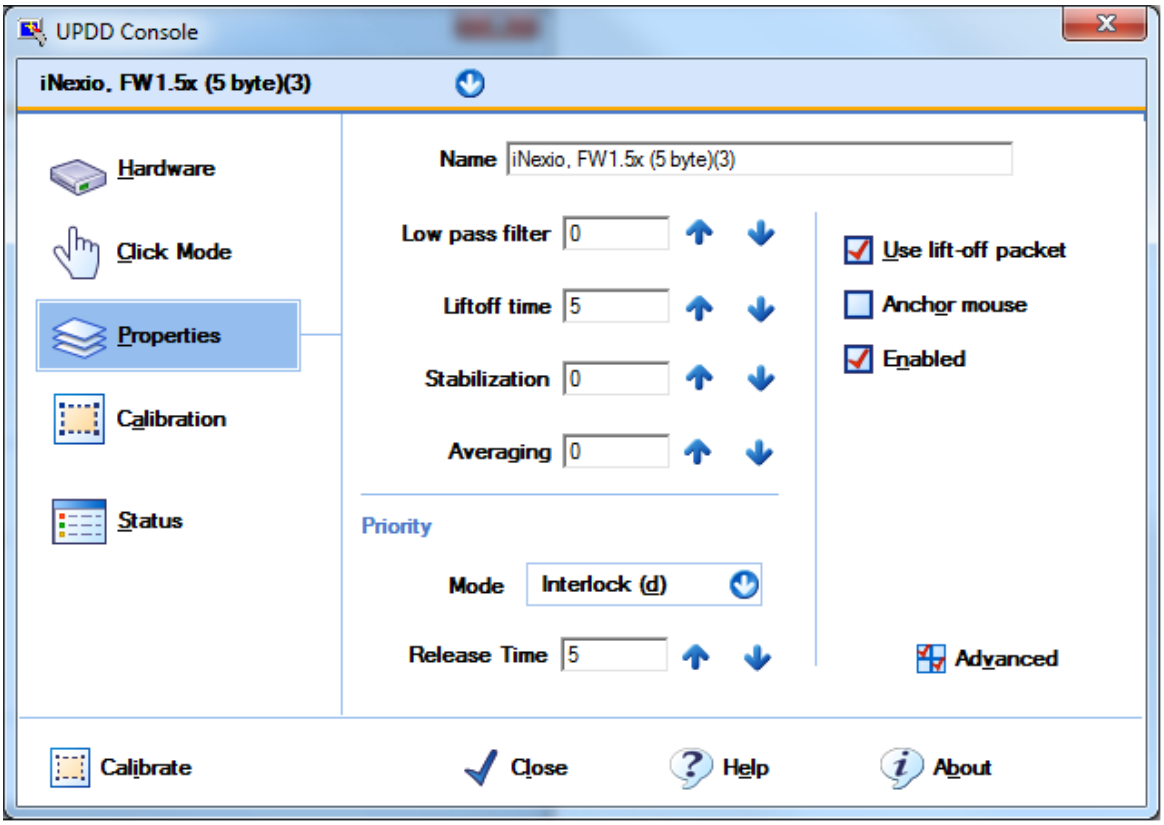
Beware: when you choose the COM port, UPDD will offer you a great many. In this case, the computer only had COM1, but UPDD still offered everything from COM1 to COM50:



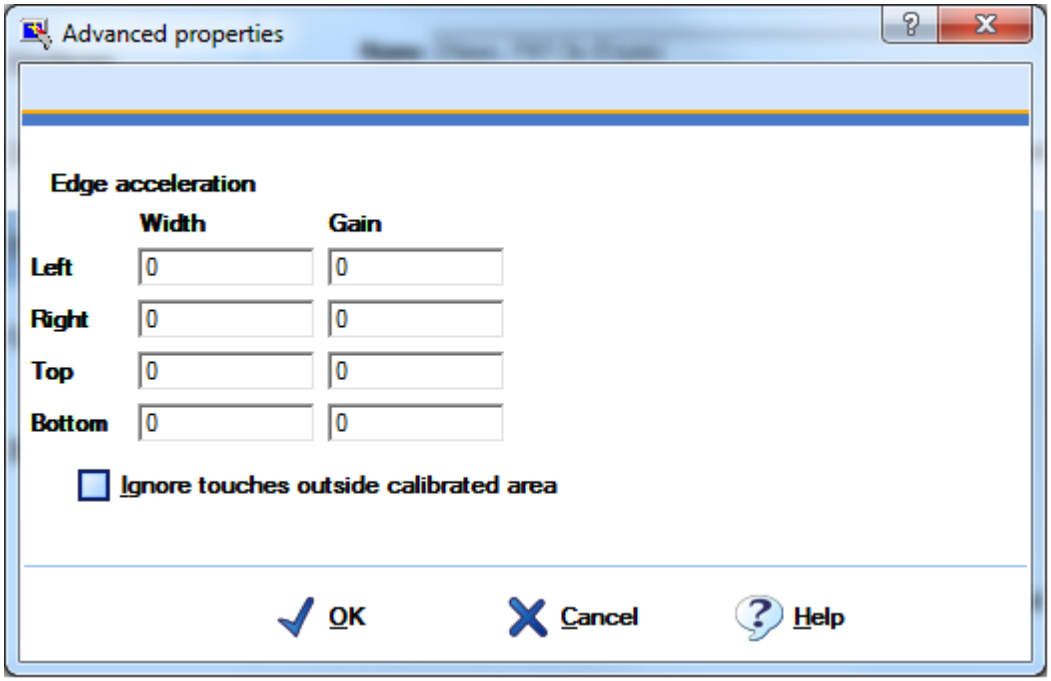
The Click Mode should have "Extended touch" disabled for non-primary monitors (according to the UPDD help), and should probably be set to "Click and drag":



The Properties page:

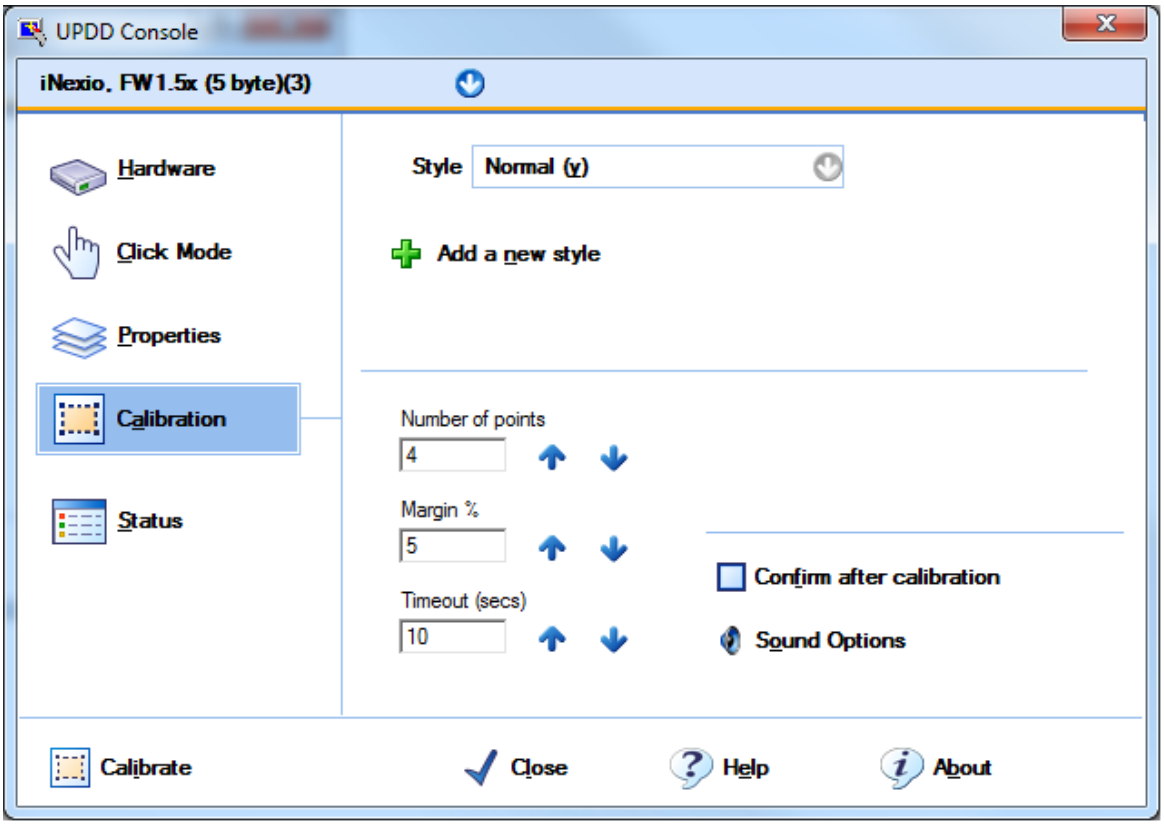


... and advanced properties (you might want to **untick** "Ignore touches outside calibrated area").

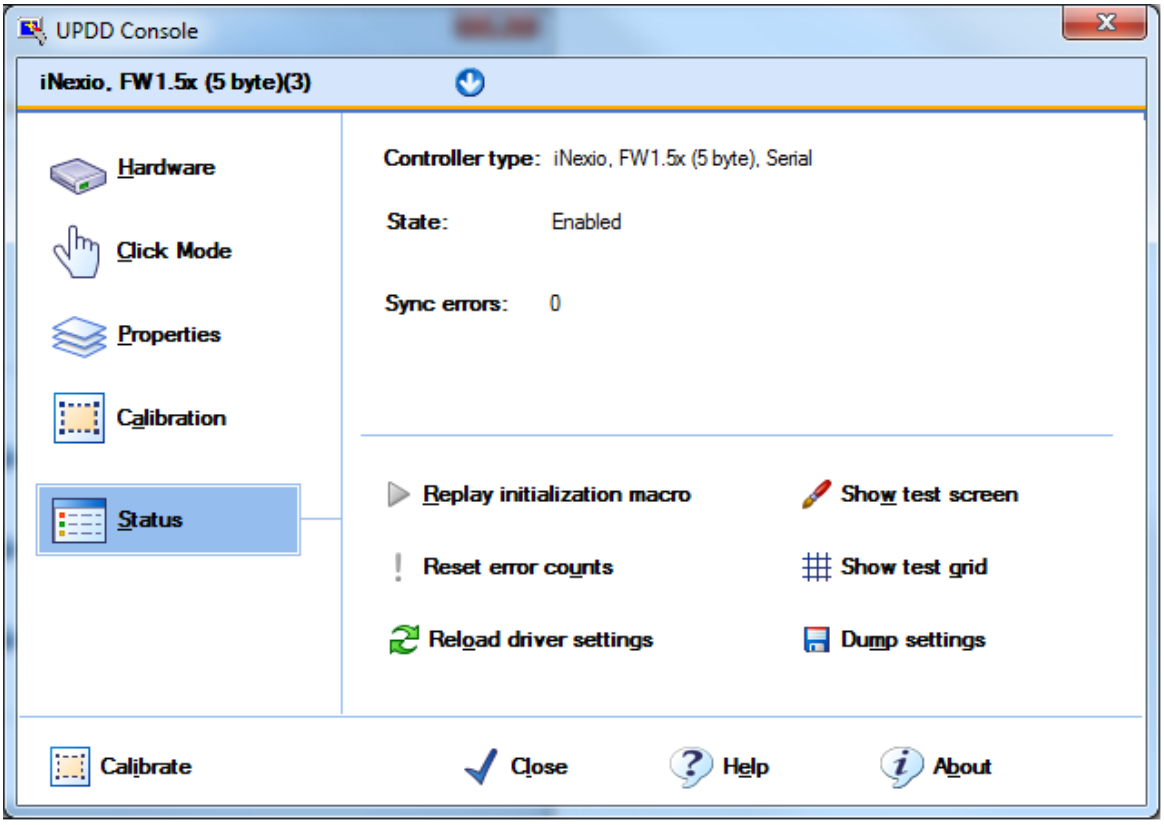


The Calibration screen configures the calibration process. **To calibrate,** just click the "Calibrate" button at the bottom left; a new window will pop up for calibration (and you may have to close

it yourself afterwards).



The final page shows the device's status, and has the helpful options to **Show test screen** (and **Show test grid**).



For more detail on the various UPDD settings, see documentation for older UPDD versions.

5.17 Med Associates operant chambers

Whisker doesn't support standard Med Associates Smart Control Interface hardware, because we don't know the specifications for low-level communication with that hardware. However, you can run Med Associates boxes using a variety of general-purpose hardware interface cards (see Hardware requirements).

Specific issues with Med Associates operant chambers:

- University of Cambridge interface system: steel boxes with 25-way connectors
- Supplying power to Med Associates operant chambers

5.17.1 University of Cambridge: Hubert's boxes (25-way connector style) (copy)

Hubert Jackson has created boxes for the University of Cambridge that enclose all the Amplicon hardware. They have the following connections:

- +28V DC in
- 0V DC in
- 78-way cable from Amplicon card in the computer
- 3 x 25-way D connections from Med Associates operant chambers

The wiring map is shown below.

Whisker line numbers are shown for three boxes, with Amplicon panels (X/Y/Z) and connection numbers within each panel (A0-C7) shown in brackets. If you have two Amplicon cards and six boxes, add 72 to each line number for the second set of three boxes.

Note that most Whisker clients will expect six boxes to be numbered 0-5, not 1-6.

Function	Box 1	Box 2	Box 3	Wire colour	25-way D socket pin	Med Associates line
Left lever operate	36 (Y-B4)	37 (Y-B5)	38 (Y-B6)	red	1	OUT 1
Liquid dipper	33 (Y-B1)	34 (Y-B2)	35 (Y-B3)	blue	2	OUT 3
Centre stimulus light	48 (Z-A0)	49 (Z-A1)	50 (Z-A2)	green	3	OUT 5
Houselight	24 (Y-A0)	25 (Y-A1)	26 (Y-A2)	yellow	4	OUT 7
Traylight	27 (Y-A3)	28 (Y-A4)	29 (Y-A5)	white	5	OUT 9
Tone	51 (Z-A3)	52 (Z-A4)	53 (Z-A5)	black	6	OUT 11
Spare 1 (output)	60 (Z-B4)	61 (Z-B5)	62 (Z-B6)	brown	7	OUT 13
Left lever response	3 (X-A3)	4 (X-A4)	5 (X-A5)	orange	9	IN 1
Right lever response	6 (X-A6)	7 (X-A7)	8 (X-B0)	pink	10	IN 2
Head entry response	0 (X-A0)	1 (X-A1)	2 (X-A2)	turquoise	11	IN 3
Left I/R response (1)	9 (X-B1)	10 (X-B2)	11 (X-B3)	grey	12	IN 4
Right I/R response (3)	15 (X-B7)	16 (X-C0)	17 (X-C1)	red/blue	13	IN 6
+28V supply	-	-	-	WHITE/RED Linked to pin 23	14	+28V
Right lever operate	39 (Y-B7)	40 (Y-C0)	41 (Y-C1)	yellow/red	15	OUT 2
Left stimulus light	42 (Y-C2)	43 (Y-C3)	44 (Y-C4)	white/red	16	OUT 4
Right stimulus light	45 (Y-C5)	46 (Y-C6)	47 (Y-C7)	red/black	17	OUT 6
Pump	30 (Y-A6)	31 (Y-A7)	32 (Y-B0)	red/brown	18	OUT 8
Clicker	57 (Z-B1)	58 (Z-B2)	59 (Z-B3)	yellow/blue	19	OUT 10
Pellet dispenser	54 (Z-A6)	55 (Z-A7)	56 (Z-B0)	white/blue	20	OUT 12
Spare 2 (output)	63 (Z-B7)	64 (Z-C0)	65 (Z-C1)	grey/blue	21	OUT 14
Shock	69 (Z-C5)	70 (Z-C6)	71 (Z-C7)	grey/black	22	OUT 16
+28V supply	- (X-SK5)	- (Y-SK5)	- (Z-SK5)	WHITE/RED	23	+28V
Ground	- (X-SK5)	- (Y-SK5)	- (Z-SK5)	WHITE/ BLACK	24	28V GROUND
Centre I/R response	12 (X-B4)	13 (X-B5)	14 (X-B6)	red/green	25	IN 5
Safety relay	66 (Z-C2) **on** during operation	67 (Z-C3) **off** during operation	not connected			

Safety relay to Z-C2 (N/O) and Z-C3 (N/C) on EX213 board. Link C2 COM to +24V and C3 COM to 0V.
Part numbers: 24V relay (RS 229-3614 or Farnell 910-806), holder (RS 802-890 or Farnell 625-619), LED (RS 802-941 or Farnell 735-073).

16 output control lines x 3 boxes = 48 lines from Y-PL3 and Z-PL4 on EX213 boards; EX213 board output control lines connected to SK7 and SK8 N/O; all COM to 0V.

6 input response line x 3 boxes = 18 lines into X-PL2 on EX230 board; EX230 board input response lines connected to SK8 0V; remove header links on EX230 board and connect J31 to J54 together to +28V (SK5,6,7,8) using header sockets (RS 406-0696 or Farnell 973-543).

(By Hubert Jackson, 4 Sep 2002.)

The corresponding device definition file should look something like this:

```
WhiskerServer v2.0 - DEVICE DEFINITION FILE - DO NOT ALTER THIS LINE
######################################################################
#
# These definitions are for 3- or 6-box systems running through Hubert's "IV"
boxes,
# which have the following connectors:
#       28V and 0V power in (plug the power supply in here)
#       a 78-way cable emerging that plugs into the computer
#       three 25-way connectors - wire these to the operant chambers, one per box
# Failsafes are 66 (on), 67 (off), 138 (on), 139 (off).
#
# Steps to follow:
# 1) Wire up the boxes according to this broad mapping scheme. *NOTE* THAT IF
YOUR MED BOXES
#    USE THE NEWER MED POWER SCHEME (SEPARATE CABLES), YOU MAY NEED TO MODIFY THE
BOXES - see separate
#    document or Whisker help system.
# 2) Make a copy of this file and modify it according to your actual mapping, if
needed.
# 3) Configure "Configure hardware -> Set device definition file" to use your
#    modified version of this file.
# 4) Configure "Configure hardware -> Configure failsafe outputs"
#    to have lines 67/138 on during operation, lines 67/139 off during operation,
and all
#    those lines off when the server exits.
# 5) Configure fake lines ("Configure hardware -> Fake (debugging) I/O lines") as
needed.
# 6) Restart the server.
#
# Hubert's "standardized" mapping is:
#
# Device                Med panel       Line numbers (for boxes 0, 1, 2, 3, 4, 5)
# ------------------------------------------------------------------------------
# L lever report        IN 1            3, 4, 5, 75, 76, 77
# R lever report        IN 2            6, 7, 8, 78, 79, 80
# Nosepoke              IN 3            0, 1, 2, 72, 73, 74
# Left locomotor beam   IN 4            9, 10, 11, 81, 82, 83
# Centre loco beam      IN 5            12, 13, 14, 84, 85, 86
# Right locomotor beam  IN 6            15, 16, 17, 87, 88, 89
# L lever operate       OUT 1           36, 37, 38, 108, 109, 110
# R lever operate       OUT 2           39, 40, 41, 111, 112, 113
# Dipper                OUT 3           33, 34, 35, 105, 106, 107
# Left light            OUT 4           42, 43, 44, 114, 115, 116
# Centre light          OUT 5           48, 49, 50, 120, 121, 122
# Right light           OUT 6           45, 46, 47, 117, 118, 119
# Houselight            OUT 7           24, 25, 26, 96, 97, 98
# Pump                  OUT 8           30, 31, 32, 102, 103, 104
# Traylight             OUT 9           27, 28, 29, 99, 100, 101
# Clicker               OUT 10          57, 58, 59, 129, 130, 131
# Tone                  OUT 11          51, 52, 53, 123, 124, 125
# Pellet                OUT 12          54, 55, 56, 126, 127, 128
```

```
# Spare 1               OUT 13          60, 61, 62, 132, 133, 134
# Spare 2               OUT 14          63, 64, 65, 135, 136, 137
# Shock                 OUT 16          69, 70, 71, 141, 142, 143
#
# In practice, you may want to modify these - e.g. adding "fake" locomotor beams
that aren't present,
# or mapping non-existent traylights to centrelights, etc.
#
# Fake lines would begin at line 144, if you define them. Suppose you define 18
fake inputs; you might use these:
# Inputs (18 of them):
#       LOCOBEAM_FRONT          lines 144, 145, 146, 147, 148, 149
#       LOCOBEAM_MIDDLE         lines 150, 151, 152, 153, 154, 155
#       LOCOBEAM_REAR           lines 156, 157, 158, 159, 160, 161
#
######################################################################


# Box 0 definition

line    0       box0    NOSEPOKE
line    3       box0    LEFTLEVER
line    6       box0    RIGHTLEVER

line    24      box0    HOUSELIGHT
line    57      box0    CLICKER
line    54      box0    PELLET
line    33      box0    DIPPER
line    36      box0    LEFTLEVERCONTROL
line    39      box0    RIGHTLEVERCONTROL
line    42      box0    LEFTLIGHT
line    45      box0    RIGHTLIGHT
line    48      box0    CENTRELIGHT
line    30      box0    PUMP

line    9       box0    LOCOBEAM_LEFT
line    12      box0    LOCOBEAM_CENTRE
line    15      box0    LOCOBEAM_RIGHT
line    27      box0    TRAYLIGHT
line    51      box0    TONE
line    60      box0    SPARE1
line    63      box0    SPARE2
line    69      box0    SHOCK

line    144     box0    LOCOBEAM_FRONT
line    150     box0    LOCOBEAM_MIDDLE
line    156     box0    LOCOBEAM_REAR

# Box 1 definition

line    1       box1    NOSEPOKE
line    4       box1    LEFTLEVER
line    7       box1    RIGHTLEVER

line    25      box1    HOUSELIGHT
line    58      box1    CLICKER
line    55      box1    PELLET
line    34      box1    DIPPER
line    37      box1    LEFTLEVERCONTROL
line    40      box1    RIGHTLEVERCONTROL
line    43      box1    LEFTLIGHT
line    46      box1    RIGHTLIGHT
line    49      box1    CENTRELIGHT
```

```
line    31      box1    PUMP

line    10      box1    LOCOBEAM_LEFT
line    13      box1    LOCOBEAM_CENTRE
line    16      box1    LOCOBEAM_RIGHT
line    28      box1    TRAYLIGHT
line    52      box1    TONE
line    61      box1    SPARE1
line    64      box1    SPARE2
line    70      box1    SHOCK

line    145     box1    LOCOBEAM_FRONT
line    151     box1    LOCOBEAM_MIDDLE
line    157     box1    LOCOBEAM_REAR

# Box 2 definition

line    2       box2    NOSEPOKE
line    5       box2    LEFTLEVER
line    8       box2    RIGHTLEVER

line    26      box2    HOUSELIGHT
line    59      box2    CLICKER
line    56      box2    PELLET
line    35      box2    DIPPER
line    38      box2    LEFTLEVERCONTROL
line    41      box2    RIGHTLEVERCONTROL
line    44      box2    LEFTLIGHT
line    47      box2    RIGHTLIGHT
line    50      box2    CENTRELIGHT
line    32      box2    PUMP

line    11      box2    LOCOBEAM_LEFT
line    14      box2    LOCOBEAM_CENTRE
line    17      box2    LOCOBEAM_RIGHT
line    29      box2    TRAYLIGHT
line    53      box2    TONE
line    62      box2    SPARE1
line    65      box2    SPARE2
line    71      box2    SHOCK

line    146     box2    LOCOBEAM_FRONT
line    152     box2    LOCOBEAM_MIDDLE
line    158     box2    LOCOBEAM_REAR

# Box 3 definition

line    72      box3    NOSEPOKE
line    75      box3    LEFTLEVER
line    78      box3    RIGHTLEVER

line    96      box3    HOUSELIGHT
line    129     box3    CLICKER
line    126     box3    PELLET
line    105     box3    DIPPER
line    108     box3    LEFTLEVERCONTROL
line    111     box3    RIGHTLEVERCONTROL
line    114     box3    LEFTLIGHT
line    117     box3    RIGHTLIGHT
line    120     box3    CENTRELIGHT
line    102     box3    PUMP

line    81      box3    LOCOBEAM_LEFT
```

```
line    84      box3    LOCOBEAM_CENTRE
line    87      box3    LOCOBEAM_RIGHT
line    99      box3    TRAYLIGHT
line    123     box3    TONE
line    132     box3    SPARE1
line    135     box3    SPARE2
line    141     box3    SHOCK

line    147     box3    LOCOBEAM_FRONT
line    153     box3    LOCOBEAM_MIDDLE
line    159     box3    LOCOBEAM_REAR

# Box 4 definition

line    73      box4    NOSEPOKE
line    76      box4    LEFTLEVER
line    79      box4    RIGHTLEVER

line    97      box4    HOUSELIGHT
line    130     box4    CLICKER
line    127     box4    PELLET
line    106     box4    DIPPER
line    109     box4    LEFTLEVERCONTROL
line    112     box4    RIGHTLEVERCONTROL
line    115     box4    LEFTLIGHT
line    118     box4    RIGHTLIGHT
line    121     box4    CENTRELIGHT
line    103     box4    PUMP

line    82      box4    LOCOBEAM_LEFT
line    85      box4    LOCOBEAM_CENTRE
line    88      box4    LOCOBEAM_RIGHT
line    100     box4    TRAYLIGHT
line    124     box4    TONE
line    133     box4    SPARE1
line    136     box4    SPARE2
line    142     box4    SHOCK

line    148     box4    LOCOBEAM_FRONT
line    154     box4    LOCOBEAM_MIDDLE
line    160     box4    LOCOBEAM_REAR

# Box 5 definition

line    74      box5    NOSEPOKE
line    77      box5    LEFTLEVER
line    80      box5    RIGHTLEVER

line    98      box5    HOUSELIGHT
line    131     box5    CLICKER
line    128     box5    PELLET
line    107     box5    DIPPER
line    110     box5    LEFTLEVERCONTROL
line    113     box5    RIGHTLEVERCONTROL
line    116     box5    LEFTLIGHT
line    119     box5    RIGHTLIGHT
line    122     box5    CENTRELIGHT
line    104     box5    PUMP

line    83      box5    LOCOBEAM_LEFT
line    86      box5    LOCOBEAM_CENTRE
line    89      box5    LOCOBEAM_RIGHT
line    101     box5    TRAYLIGHT
```

```
line    125     box5    TONE
line    134     box5    SPARE1
line    137     box5    SPARE2
line    143     box5    SHOCK

line    149     box5    LOCOBEAM_FRONT
line    155     box5    LOCOBEAM_MIDDLE
line    161     box5    LOCOBEAM_REAR
```

See also:

- Configure digital I/O cards and install them into the computer.
- Configure EX233 distributor boards.
- Configure EX213 output panels.
- Configure EX230 input panels.
- Configure EX221 mixed panels.
- Install a safety relay device.

5.17.2 Operant chamber power connections

Med Associates operant chambers use 28V devices. Output devices (e.g. pellet dispensers) are triggered by shorting the control line to ground; input devices (e.g. infrared beam detectors) short their report line to ground when they are triggered. All devices have a 3-way connector (the pin at the flat end is +28 V; the pin at the pointy end is 0 V; the centre pin is the data line).

In or about 2002, Med changed their wiring system. New boxes have a 25-way data connector and a heavy-duty 2-way power cable.

Pin on 25-way D connector on SG-716D2 panel attached to operant chamber	**Corresponding SG-716D2 panel label (where devices are attached to the panel)**	**Old Med boxes used a different connection system, in which power was transmitted down the same 25-way cable as data:**
1	OUT 1	OUT 1
2	OUT 3	OUT 3
3	OUT 5	OUT 5
4	OUT 7	OUT 7
5	OUT 9	OUT 9
6	OUT 11	OUT 11
7	OUT 13	OUT 13
8	OUT 15	OUT 15
9	IN 1	IN 1
10	IN 2	IN 2
11	IN 3	IN 3
12	IN 4	IN 4
13	IN 6	IN 6
14	IN 7	**+28 V (power)**
15	OUT 2	OUT 2
16	OUT 4	OUT 4
17	OUT 6	OUT 6
18	OUT 8	OUT 8
19	OUT 10	OUT 10
20	OUT 12	OUT 12
21	OUT 14	OUT 14

22	OUT 16	OUT 16
23	IN 8	**+28 V (power)**
24	(not connected)	**0 V (ground)**
25	IN 5	IN 5

Since we (at the University of Cambridge) already had custom boxes designed for the old Med systems with 25-way connectors, we had a choice of redesigning our boxes or rewiring the Med boxes. We chose to rewire the Med boxes for simplicity, even though this keeps the power going down the light-gauge 25-way cable. We achieved this using the following conversion:

- on the PCB of the SG-716D2 panel, solder pin 24 (0 V) to the 0 V rail;
- on the PCB of the SG-716D2 panel, solder pin 23 or pin 14 (+28 V) to the +28 V rail;
- obstruct sockets IN 7 and IN 8 on the front of the panel, since they are now connected to +28 V and are unusable as inputs;
- stick a notice on the panel saying what's been done!

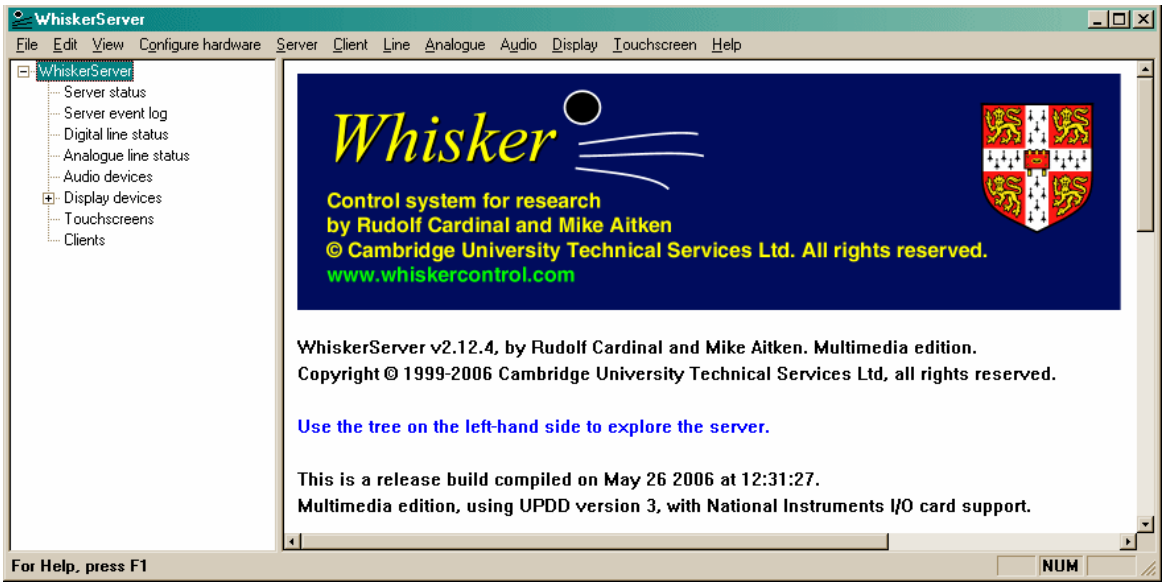
Part VI

Using the WhiskerServer console

6 Using the WhiskerServer console

6.1 Introduction

Start → Whisker → Whisker Server



The WhiskerServer program must be running at all times when you wish to use the operant chambers. It is this program that actually communicates with the hardware.

However, you almost never need to interact with the server while it is running. It can provide you with a great deal of information about what's going on within your system, and can be very helpful for wiring up the system, testing devices and debugging client programs, but in day-to-day operation you can simply minimize it and forget it. (If you try to close the server while tasks are running, thus aborting any ongoing tasks, it will complain vociferously so you don't do it by accident.)

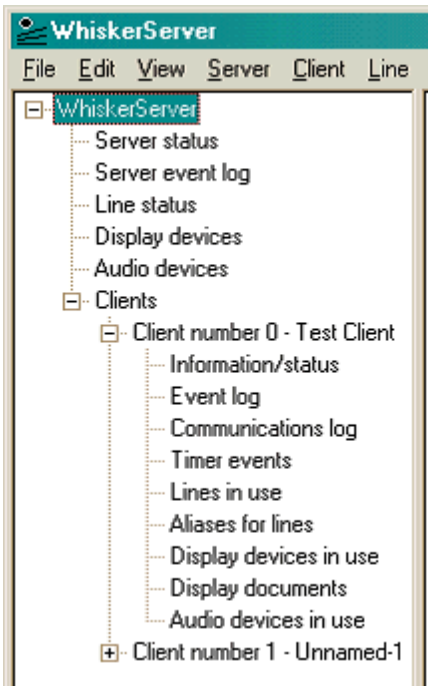
6.2 Editions of WhiskerServer

WhiskerServer is supplied in three editions:

- a **Standard Edition,** which controls digital input and output devices,
- a **Multimedia Edition,** which has all the facilities of the standard edition but also supports display devices (monitors), touchscreens, mouse input, keyboard input, and audio devices (sound cards);
- a **Programmers' Edition,** which does not support most physical I/O devices, and does not support touchscreens, but does support display devices (monitors), sound, keyboard and mouse input, "fake" digital I/O lines and serial port I/O. This edition is intended for writing and testing tasks without access to physical I/O hardware (with the tasks eventually running on a computer with the Standard or Multimedia edition installed); it also allows human testing with serial-port-based I/O devices (e.g. button boxes), keyboard/mouse input, and visual displays.

If you have the standard or programmers' version, some items on the server console's display relating to unsupported devices will be absent or greyed out.

6.3 The Left-Hand Tree



This picture shows the various views available to the server. When you click on an item on this tree, a view opens up on the right-hand side of the screen.

The root view, labelled *WhiskerServer*, displays version information about the server.

There are several views giving a summary of the server's status as a whole (from *Server status* through to *Clients*). Click on the tree on the left to open each view in the right-hand window.

Then, for each connected client, there are several views giving information about just that client (beginning with *Information/status*).

The tree structure in **this help system** mirrors the tree view in WhiskerServer, dividing the views up into those that give you information about the server as a whole, and those that pertain to individual clients.

Use the **mouse** to navigate around the display. Click '+' symbols to open up part of the tree and '–' symbols to collapse the tree.

Using the **keyboard,** you may use the up/down arrow keys to navigate around the tree; the right and left arrow keys open up and collapse parts of the tree, respectively. Use TAB to switch to the right-hand window, and Shift-TAB to switch from the right-hand to the left-hand view (except that you cannot use the keyboard to move the input focus from the right-hand view if it is showing a multimedia display, because keyboard input to a view like this may be used for a different purpose).

6.3.1 Views pertaining to the server as a whole

Please select a subheading.

6.3.1.1 Server status

Displays information about the server's network status, and how many clients are connected. Here's an example:

```
WhiskerServer v2.0, by Rudolf Cardinal. Multimedia edition.

Network status: Running
Server hostname: needle
Server IP address: 192.168.0.1
TCP ports are 1333 (main) and 1334 (immediate)

Number of clients: 2
Maximum number of clients permitted: 15

Worst inter-poll interval so far (ms): 2
This display is scheduled to be updated every 1000 ms
Worst inter-poll interval since last update (ms): 2
Since last update, have had 1008 polls, of which 100.0% were <=10ms, 0.0% were 11-20 ms, 0.0% were >20ms
Server process priority: Real-time
```

Version information. This display is from WhiskerServer v2.0, multimedia edition.

Network status. Since the server is accepting connections from clients, the network status shows as 'running'. The server's name and *IP address* is displayed. (In addition to this IP address, the address 127.0.0.1 can be used to mean 'this computer'.) Every TCP/IP based program makes a connection using a *port number* (for example, the HTTP protocol that drives the Web usually uses port 80) and the TCP port that Whisker uses by default to establish connection is 3233.

Connected clients. This example shows that two clients are currently connected, and there is an arbitrary limit of 64 clients. (If this limit causes problems, e-mail *rudolf@pobox.com*.)

Performance information. The bottom section of this display shows performance information. Whisker operates by asking the digital I/O boards very frequently whether anything has changed since it last asked, a technique known as *polling*. In this example, the longest time between two consecutive polls since the server was restarted (or the timing statistics were reset) is 2 ms; the display you are looking at will be updated approximately once every 1000 ms; the worst inter-poll interval since the display was last updated was 2 ms, and since the last display update (1000 ms ago) there have been 1008 pools, all of which have occurred with ≤10 ms between them (in fact, in this case, we know from the previous numbers that Whisker was performing rather better than this and no inter-poll time was longer than 2 ms). The meaning of this information is discussed in more

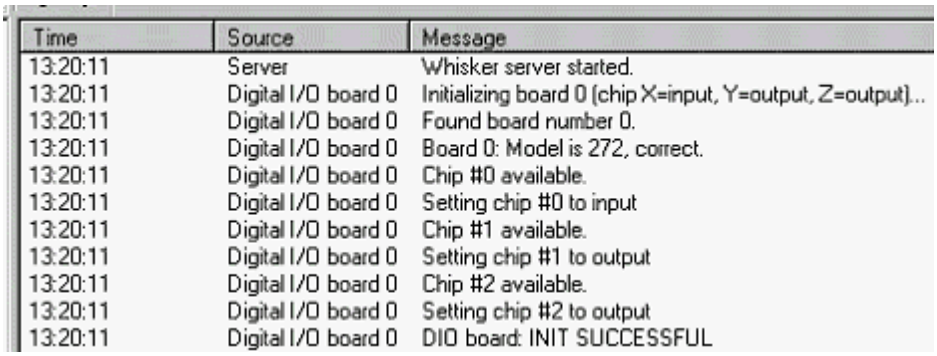
detail later (Whisker — Performance considerations).

Technical note: why use polling?

Because the Amplicon boards cannot generate interrupts when an arbitrary input line changes state.

6.3.1.2 Server event log

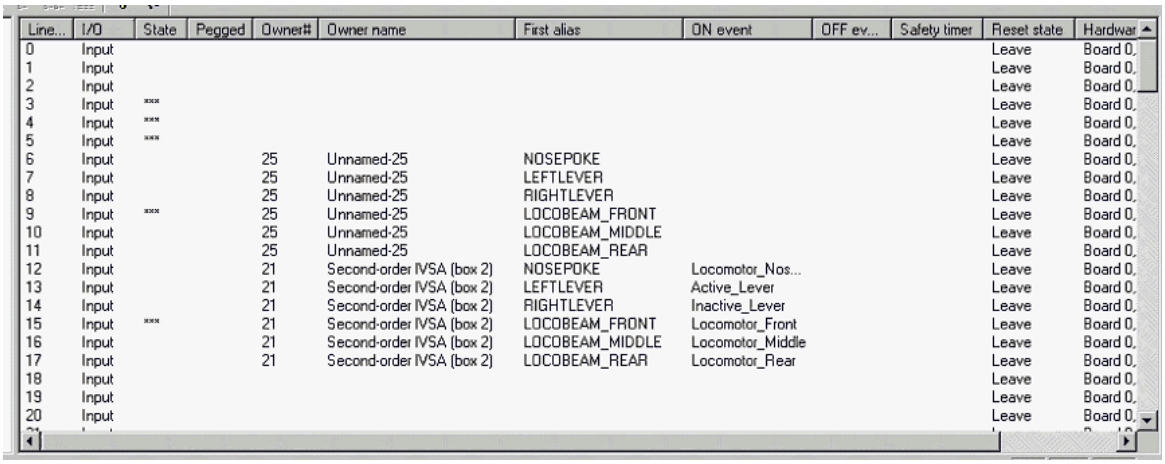
Less than thrilling, this displays significant information pertaining to the whole server. For example, when clients connect and disconnect, and when the hardware is first initialized, status and error messages are recorded here. Here's a screenshot:



Time	Source	Message
13:20:11	Server	Whisker server started.
13:20:11	Digital I/O board 0	Initializing board 0 (chip X=input, Y=output, Z=output)...
13:20:11	Digital I/O board 0	Found board number 0.
13:20:11	Digital I/O board 0	Board 0: Model is 272, correct.
13:20:11	Digital I/O board 0	Chip #0 available.
13:20:11	Digital I/O board 0	Setting chip #0 to input
13:20:11	Digital I/O board 0	Chip #1 available.
13:20:11	Digital I/O board 0	Setting chip #1 to output
13:20:11	Digital I/O board 0	Chip #2 available.
13:20:11	Digital I/O board 0	Setting chip #2 to output
13:20:11	Digital I/O board 0	DIO board: INIT SUCCESSFUL

You can save the log to disk using the **File / Save As** menu command, or by pressing **Ctrl-S**.

6.3.1.3 Digital line status



Line...	I/O	State	Pegged	Owner#	Owner name	First alias	ON event	OFF ev...	Safety timer	Reset state	Hardwar
0	Input									Leave	Board 0,
1	Input									Leave	Board 0,
2	Input									Leave	Board 0,
3	Input	***								Leave	Board 0,
4	Input	***								Leave	Board 0,
5	Input	***								Leave	Board 0,
6	Input			25	Unnamed-25	NOSEPOKE				Leave	Board 0,
7	Input			25	Unnamed-25	LEFTLEVER				Leave	Board 0,
8	Input			25	Unnamed-25	RIGHTLEVER				Leave	Board 0,
9	Input	***		25	Unnamed-25	LOCOBEAM_FRONT				Leave	Board 0,
10	Input			25	Unnamed-25	LOCOBEAM_MIDDLE				Leave	Board 0,
11	Input			25	Unnamed-25	LOCOBEAM_REAR				Leave	Board 0,
12	Input			21	Second-order IVSA (box 2)	NOSEPOKE	Locomotor_Nos...			Leave	Board 0,
13	Input			21	Second-order IVSA (box 2)	LEFTLEVER	Active_Lever			Leave	Board 0,
14	Input			21	Second-order IVSA (box 2)	RIGHTLEVER	Inactive_Lever			Leave	Board 0,
15	Input	***		21	Second-order IVSA (box 2)	LOCOBEAM_FRONT	Locomotor_Front			Leave	Board 0,
16	Input			21	Second-order IVSA (box 2)	LOCOBEAM_MIDDLE	Locomotor_Middle			Leave	Board 0,
17	Input			21	Second-order IVSA (box 2)	LOCOBEAM_REAR	Locomotor_Rear			Leave	Board 0,
18	Input									Leave	Board 0,
19	Input									Leave	Board 0,
20	Input									Leave	Board 0,

Lists every digital I/O line available to the server. **All boards and lines are numbered from zero.** The fields are as follows:

- **Line#.** The line number.
- **I/O.** Whether this is an input or an output line.
- **State.** If the line is on, '###' appears here. If it is off, nothing is displayed.
- **Pegged.** Whether the line has been forced ON or OFF by you, the person operating the server console. If nothing is displayed here, the line is free to be controlled by the client (for

output lines) or the physical device (for input lines). **Pegged lines are displayed with coloured backgrounds** to make them stand out.

- **Owner.** Name of the client that has claimed this line.
- **First alias.** The name that the client first used to refer to this line. This might be something informative, like 'Houselight'.
- **'On' event.** (For input lines.) The message, if any, that will be sent to the client whenever the line is turned on.
- **'Off' event.** (For input lines.) The message, if any, that will be sent to the client whenever the line is turned off.
- **Safety timer.** (For output lines.) For critical lines, like intravenous infusion pumps, the client can ask the server to make sure that the line is switched on or off if the client hasn't mentioned it for a while. This display might say 'OFF after 10000 ms', for example; then if the client crashes, the server will ensure the line is not left on accidentally.
- **Reset state.** (For output lines.) When the client releases the line, the server can leave it in whatever state it was in, ensure that it is off, or ensure that it is on. This display tells you what the client has chosen for this particular line.
- **Hardware description.** This is an aid to finding the physical connections associated with this line. If you have eight I/O boards, each with 72 lines, you might have difficulty finding line 524 – this display should tell you that line 524 is on board 7, channel X, line C4.
- **Server device group.** The server may be configured to give names to its devices and assign them to groups. If the server is using a device definition file and this device has been named, its *device group* appears here.
- **Server device name.** Similarly, its *device name* appears in this column.
- **State.** The state is repeated on the far right-hand side.

Tips

Some of the field may not initially be visible. Use the scroll bars to move down or to the right.

You can resize all of the fields on any list-like display by dragging the edges of the field headings. In this manner, you can bring the line number and the hardware description into view at the same time, which may be very useful for wiring up a system.)

If you click on a line, it will be highlighed and selected, like this:

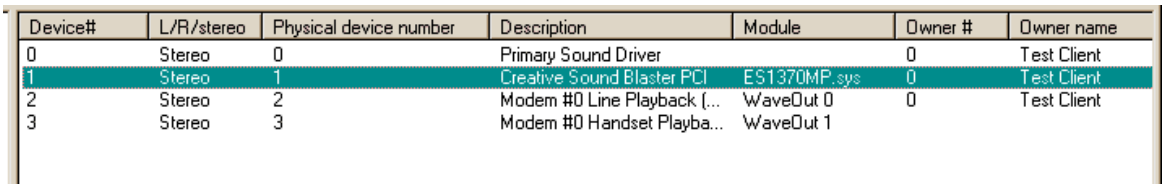
Line number	I/O	State	Pegged	Owner#	Owner name	First alias	ON event	OFF event
0	Input							
1	Input							
2	Input							
3	Input							
4	Input			1	Test Client	Lever	LeverPressed	
5	Input			1	Test Client	NosepokeDetector	AlcoveEntered	AlcoveLeft
6	Input							
7	Input							
8	Input							
9	Input							

When a line is selected, the Line menu will refer to that line.

6.3.1.4 Analogue line status

(TO BE ADDED.)

6.3.1.5 Audio devices



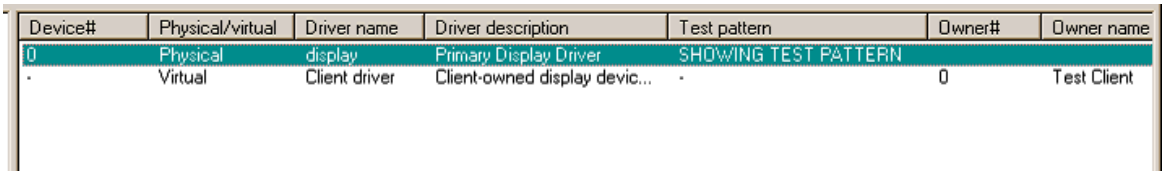
Device#	L/R/stereo	Physical device number	Description	Module	Owner #	Owner name
0	Stereo	0	Primary Sound Driver		0	Test Client
1	Stereo	1	Creative Sound Blaster PCI	ES1370MP.sys	0	Test Client
2	Stereo	2	Modem #0 Line Playback (...	WaveOut 0	0	Test Client
3	Stereo	3	Modem #0 Handset Playba...	WaveOut 1		

Lists every audio output device in use by the server. (These may be configured via *Configure hardware* → *Audio devices*).

- **Device#.** The device number. Devices are numbered from 0.
- **L/R/stereo.** If this shows 'stereo', the whole audio device is being treated as a unit. If this says 'L' or 'R', then the audio device represents a stereophonic sound card that has been *split* into two separate channels (left and right), to double the number of available (monophonic) sound devices. Note that some poor-quality sound cards exhibit 'bleed', such that sounds intended to come only from the left channel are audible (albeit quietly) on the right channel; these sound cards are probably unsuitable for splitting.
- **Physical device number.** The physical device number, as reported by Windows. (If you have split a sound card, two devices will have the same physical device number.)
- **Description.** The device description provided by Windows.
- **Module.** The sound module supporting the device, as reported by Windows.
- **Owner #.** The number of the client that owns the device at present.
- **Owner name.** The name of the client that owns the device at present.
- **Number of buffers loaded.** The number of sound buffers currently attached to this device.
- **Server device group.** The device group, as defined in the server's device definition file.
- **Server device name.** The device name, as defined in the server's device definition file.

When an audio device is selected in the list, you can test it individually (see Audio Menu).

6.3.1.6 Display devices



Device#	Physical/virtual	Driver name	Driver description	Test pattern	Owner#	Owner name
0	Physical	display	Primary Display Driver	SHOWING TEST PATTERN		
-	Virtual	Client driver	Client-owned display devic...	-	0	Test Client

Lists every physical display device (monitor) in use by the server. (These may be configured via *Configure hardware* → *Display devices*).

- **Device#.** The device's number. Devices are numbered from 0.
- **Physical/virtual.** Physical devices represent monitors connected to your computer, which clients may take control of. Logical devices are windows on the desktop that clients may create freely. (Only physical devices are shown on the server's summary; virtual devices are shown in clients' summary views.)
- **Driver name.** The name given to the device by Windows (or 'Client driver' for virtual displays).
- **Driver description.** The device driver's description, provided by Windows (or 'Client-owned display device').
- **Test pattern.** Whether the device is showing a test pattern or not. Test patterns can't be shown on devices that have been claimed by a client.

- **Owner #.** The number of the client that owns the device at present.
- **Owner name.** The name of the client that owns the device at present.
- **Device name.** The name that the client has given the device (only applicable to virtual displays).
- **Current document.** The document that is currently being displayed on the device (or would be being displayed, were a test pattern not overriding this option).
- **Server device group.** The device group, as defined in the server's device definition file.
- **Server device name.** The device name, as defined in the server's device definition file.
- **Detailed operational information.** Screen size (with and without any border), whether or not the device is scaling documents, etc.

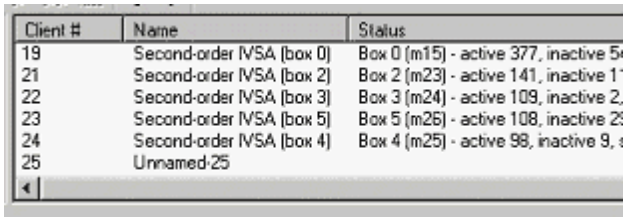
6.3.1.6.1 Physical display X - views of individual displays

Each physical display device has its own entry in the left-hand tree, under *Display devices*. Choosing one of these views lets you see a copy of whatever is happening on that display device. This is useful if the physical monitor is hidden inside an operant chamber! There are a number of further things that you can do with this view; they are discussed with the Display menu.

6.3.1.7 Touchscreens

Lists the touchscreens (by device ID) and the display device they are attached to.

6.3.1.8 Clients



Lists all connected clients, together with their name (supplied by the client), current status (supplied by the client), time since the client last communicated with the server, and whether any network errors have occurred while trying to communicate with this client.

Note that clients are assigned numbers by the server (starting with 1, and ticking up for as long as the server is running). These numbers usually bear no relationship to the box number being used, or to anything else – they are merely a way of keeping clients separate. Good clients will provide a name to the server, giving more useful information: "Second-order IVSA, box 4", for example. This replaces the "Unnamed-1", "Unnamed-2", etc, which the server assigns to clients before they have provided a proper name.

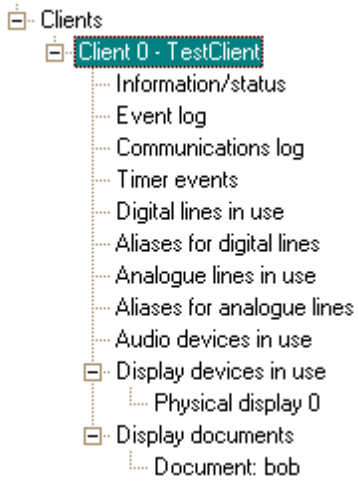
- **Client #.** The client number (see above).
- **Name.** The client's name, as reported by the client itself.
- **Status.** The client's status, as reported by the client.
- **Time since last communication.** The time elapsed since the client last sent a message to

the server.

- **Network status messages.** If a network failure is detected, an error message will appear here.

6.3.2 Views pertaining to individual clients

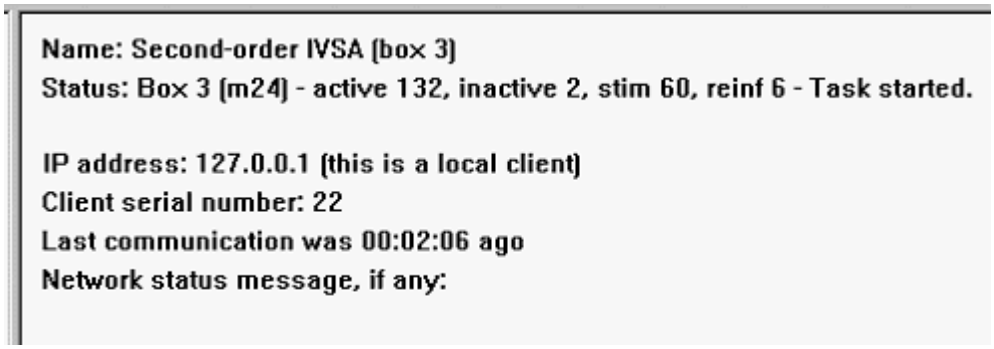
For each client, there is a further tree branch with views pertaining to just that client.



Note: when clients disconnect or are deleted

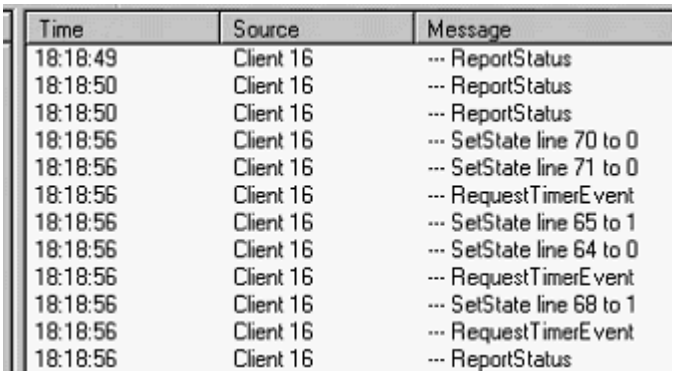
When a client is disconnected, there may be a brief (<1 s) delay before its displays vanish from the server console. This is a consequence of the way WhiskerServer's multiple threads operate internally. As a result, during this time it is possible that displays show the message 'Internal pointer error' (as they are still trying to display a client that no longer exists). This is nothing to worry about. (If the message persists for much longer, that suggests a bug! Report it to *rudolf@pobox.com* with details of which view it was and the circumstances, especially if it is repeatable, and choose *View* → *Refresh* to refresh your views.)

6.3.2.1 Information/status



Gives the client's name (supplied by the client), status (supplied by the client), the IP address of the computer the client is running on, an arbitrary serial number, the time since the client last sent anything to the server, and whether any network errors have been noted by the server while trying to communicate with the client.

6.3.2.2 Event log

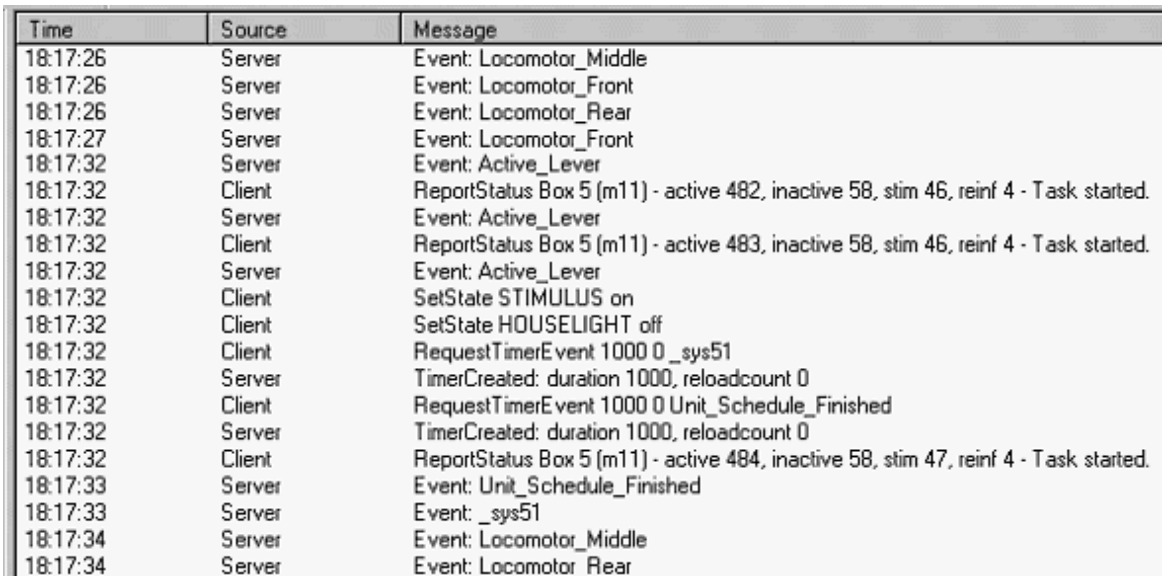


Time	Source	Message
18:18:49	Client 16	--- ReportStatus
18:18:50	Client 16	--- ReportStatus
18:18:50	Client 16	--- ReportStatus
18:18:56	Client 16	--- SetState line 70 to 0
18:18:56	Client 16	--- SetState line 71 to 0
18:18:56	Client 16	--- RequestTimerEvent
18:18:56	Client 16	--- SetState line 65 to 1
18:18:56	Client 16	--- SetState line 64 to 0
18:18:56	Client 16	--- RequestTimerEvent
18:18:56	Client 16	--- SetState line 68 to 1
18:18:56	Client 16	--- RequestTimerEvent
18:18:56	Client 16	--- ReportStatus

This view shows significant events that pertain to this client (assuming its event log is enabled). This will include all commands issued by the client to the server.

You can save the log to disk using the **File / Save As** menu command, or by pressing **Ctrl-S**.

6.3.2.3 Communications log



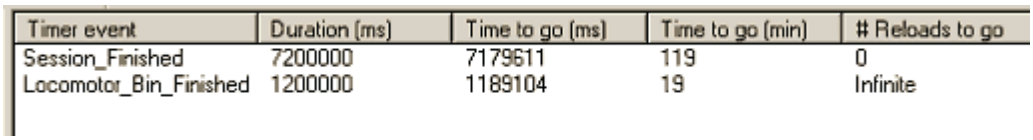
Time	Source	Message
18:17:26	Server	Event: Locomotor_Middle
18:17:26	Server	Event: Locomotor_Front
18:17:26	Server	Event: Locomotor_Rear
18:17:27	Server	Event: Locomotor_Front
18:17:32	Server	Event: Active_Lever
18:17:32	Client	ReportStatus Box 5 (m11) - active 482, inactive 58, stim 46, reinf 4 - Task started.
18:17:32	Server	Event: Active_Lever
18:17:32	Client	ReportStatus Box 5 (m11) - active 483, inactive 58, stim 46, reinf 4 - Task started.
18:17:32	Server	Event: Active_Lever
18:17:32	Client	SetState STIMULUS on
18:17:32	Client	SetState HOUSELIGHT off
18:17:32	Client	RequestTimerEvent 1000 0 _sys51
18:17:32	Server	TimerCreated: duration 1000, reloadcount 0
18:17:32	Client	RequestTimerEvent 1000 0 Unit_Schedule_Finished
18:17:32	Server	TimerCreated: duration 1000, reloadcount 0
18:17:32	Client	ReportStatus Box 5 (m11) - active 484, inactive 58, stim 47, reinf 4 - Task started.
18:17:33	Server	Event: Unit_Schedule_Finished
18:17:33	Server	Event: _sys51
18:17:34	Server	Event: Locomotor_Middle
18:17:34	Server	Event: Locomotor_Rear

This is a display primarily for debugging, as it can accumulate huge numbers of messages. It displays everything that has passed between client and server (assuming the communications log is on).

The *Source* column also shows whether a 'main' or 'immediate' socket was used (described in detail in the Whisker Programmer's Guide); the source of the message is given as Server, Client, Server-IMM or Client-IMM (where -IMM denotes an immediate socket).

You can save the log to disk using the **File / Save As** menu command, or by pressing **Ctrl-S**.

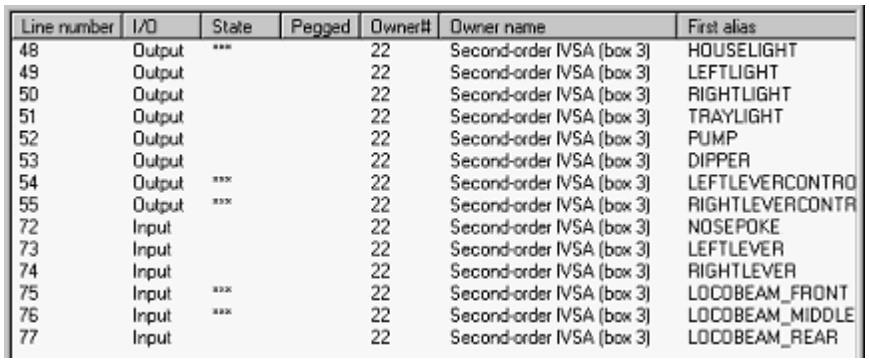
6.3.2.4 Timer events



Timer event	Duration (ms)	Time to go (ms)	Time to go (min)	# Reloads to go
Session_Finished	7200000	7179611	119	0
Locomotor_Bin_Finished	1200000	1189104	19	Infinite

Displays every timer currently in use by the client. You see the name of the timer (i.e. the event that is sent to the client when the timer expires), its overall duration in milliseconds, the time left before the timer expires (in milliseconds and in minutes), and the number of reloads left.

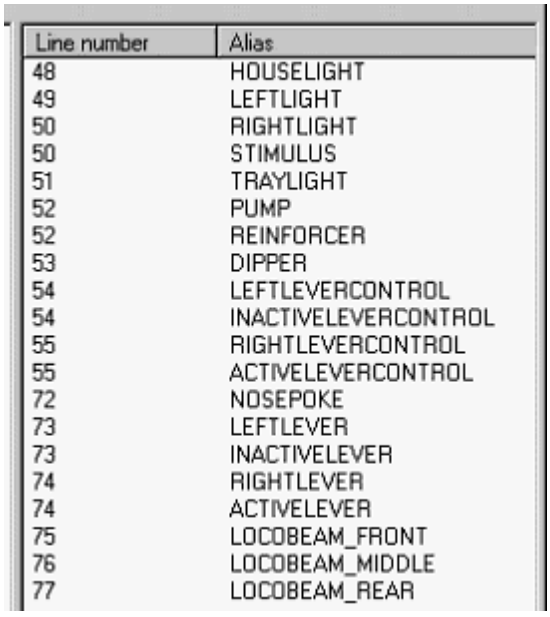
6.3.2.5 Digital lines in use



Line number	I/O	State	Pegged	Owner#	Owner name	First alias
48	Output	***		22	Second-order IVSA (box 3)	HOUSELIGHT
49	Output			22	Second-order IVSA (box 3)	LEFTLIGHT
50	Output			22	Second-order IVSA (box 3)	RIGHTLIGHT
51	Output			22	Second-order IVSA (box 3)	TRAYLIGHT
52	Output			22	Second-order IVSA (box 3)	PUMP
53	Output			22	Second-order IVSA (box 3)	DIPPER
54	Output	***		22	Second-order IVSA (box 3)	LEFTLEVERCONTRO
55	Output	***		22	Second-order IVSA (box 3)	RIGHTLEVERCONTR
72	Input			22	Second-order IVSA (box 3)	NOSEPOKE
73	Input			22	Second-order IVSA (box 3)	LEFTLEVER
74	Input			22	Second-order IVSA (box 3)	RIGHTLEVER
75	Input	***		22	Second-order IVSA (box 3)	LOCOBEAM_FRONT
76	Input	***		22	Second-order IVSA (box 3)	LOCOBEAM_MIDDLE
77	Input			22	Second-order IVSA (box 3)	LOCOBEAM_REAR

The view is identical to the server line view, but only shows lines claimed by the current client.

6.3.2.6 Aliases for digital lines



Line number	Alias
48	HOUSELIGHT
49	LEFTLIGHT
50	RIGHTLIGHT
50	STIMULUS
51	TRAYLIGHT
52	PUMP
52	REINFORCER
53	DIPPER
54	LEFTLEVERCONTROL
54	INACTIVELEVERCONTROL
55	RIGHTLEVERCONTROL
55	ACTIVELEVERCONTROL
72	NOSEPOKE
73	LEFTLEVER
73	INACTIVELEVER
74	RIGHTLEVER
74	ACTIVELEVER
75	LOCOBEAM_FRONT
76	LOCOBEAM_MIDDLE
77	LOCOBEAM_REAR

The view normally lists aliases in the order they were defined. Click on the phrase *Line number* to sort by line number instead.

6.3.2.7 Analogue lines in use

(TO BE ADDED.)

6.3.2.8 Aliases for analogue lines

(TO BE ADDED.)

6.3.2.9 Audio devices in use

Device#	L/R/stereo	Physical device number	Description	Module	Owner #	Owner name
0	Stereo	0	Primary Sound Driver		0	Test Client
1	Stereo	1	Creative Sound Blaster PCI	ES1370MP.sys	0	Test Client
2	Stereo	2	Modem #0 Line Playback (...	WaveOut 0	0	Test Client

Lists the audio devices currently controlled by the client. (See the equivalent server view for a full description of the columns.) When an audio device is selected in the list, you can test it individually (see Audio Menu).

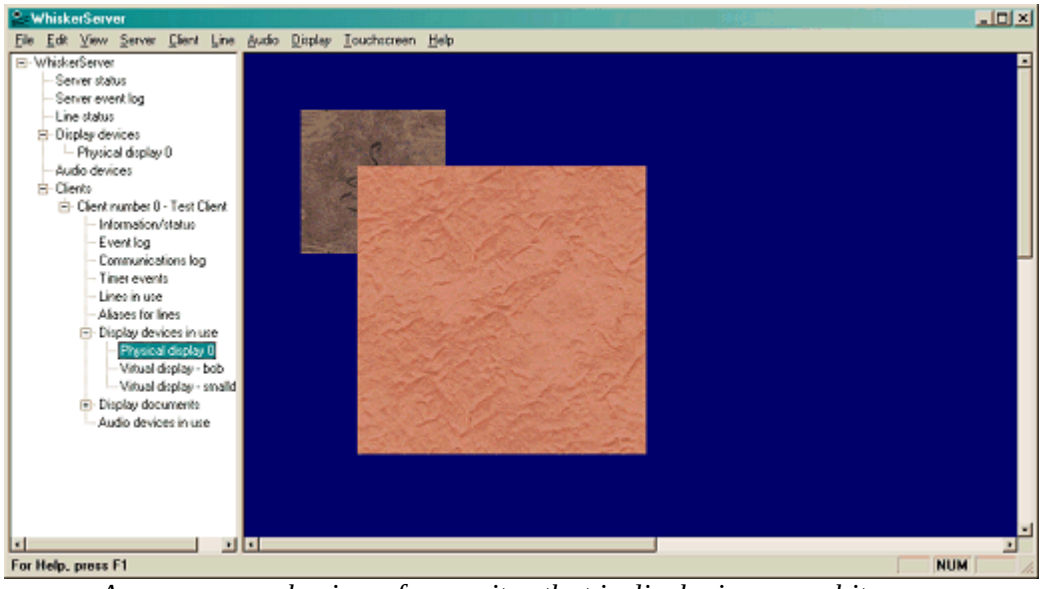
6.3.2.10 Display devices in use

Device#	Physical/virtual	Driver name	Driver description	Test pattern	Owner#	Owner name
0	Physical	display	Primary Display Driver	SHOWING TEST PATTERN		
-	Virtual	Client driver	Client-owned display devic...	-	0	Test Client

Lists all the display devices — physical or virtual — in use by the client. (The fields were described in detail earlier.) If a display is virtual, the *Device number*, *Driver name* and *Driver description* columns are uninformative but the *Display name* field contains the name given to the virtual display by the client.

When a display is selected, the Display menu operates on that line.

6.3.2.10.1 Views of individual displays

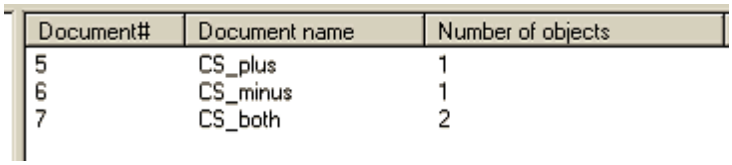


A server console view of a monitor that is displaying some bitmaps.

Each display controlled by the client has its own entry in the left-hand tree. Choose these items to see a *copy* of that display device. This is useful if the physical monitor is hidden inside an operant chamber.

The Display menu allows you to do things with these views.

6.3.2.11 Display documents

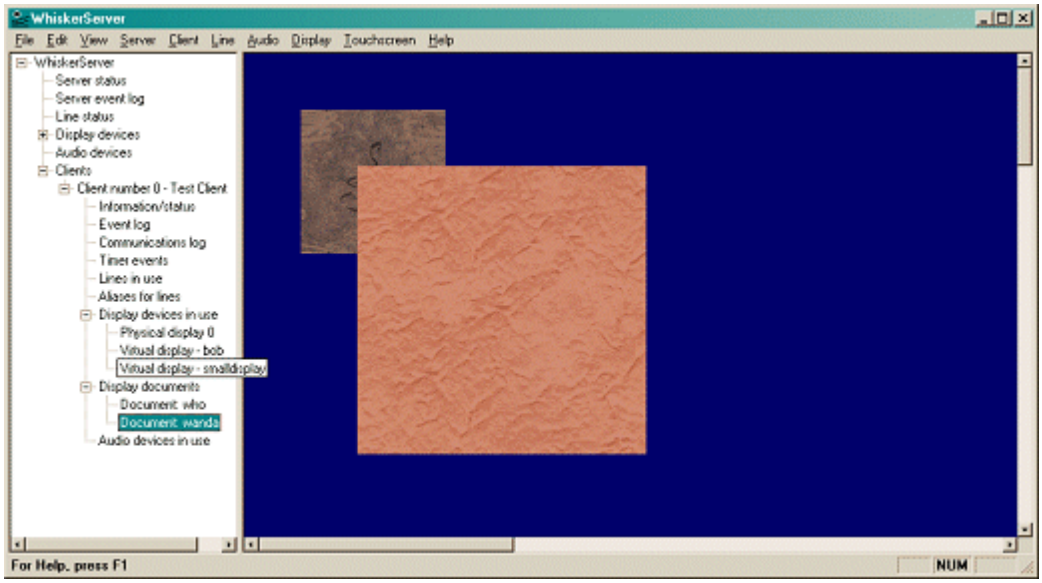


Document#	Document name	Number of objects
5	CS_plus	1
6	CS_minus	1
7	CS_both	2

Clients create *documents* which they can then show on any of their displays. This view shows a summary of those documents.

- **Document #.** The document number (only used internally).
- **Document name.** The document's name. Clients create and address the document using this name.
- **Objects shown.** The number of objects (e.g. bitmaps) present in the document.
- **Objects in cached version.** The number of objects (e.g. bitmaps) present in the cached version of the document, if there is one. Caching is a technique that allows complex documents to be built up "behind the scenes", so that the whole new document can be displayed very quickly without flicker.
- **Detailed information.** Gives the document's size, whether or not caching is in use, etc.

6.3.2.11.1 Views of individual documents

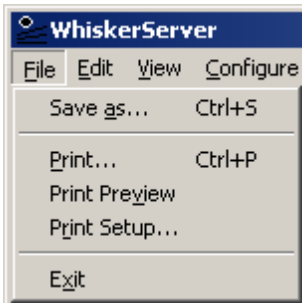


Each document is entered onto the left-hand tree so you can view them.

As documents only respond to mouse/touchscreen events when they are being displayed, the server's view of a document will never respond to the mouse; nor can it be scaled. (Views of displays, on the other hand, do support these features.)

6.4 The Menus

6.4.1 File

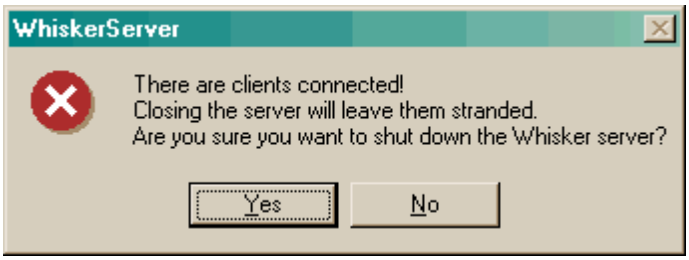


The File menu is available all the time. It does two useful things.

(1) If you are looking at one of the server's logs, such as the server event log, a client event log, or a client communication log, you can use the **Save As** command to save the log to a text file on disk.

(2) It contains the **Exit** command, used to stop WhiskerServer. WhiskerServer will request confirmation before closing:

If clients are connected, closing WhiskerServer will interrupt them and may screw up an experiment. Therefore, additional confirmation is requested:

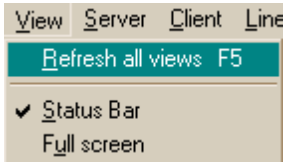


If you confirm, the server will then close.

6.4.2 Edit

The Edit menu is not very helpful.

6.4.3 View



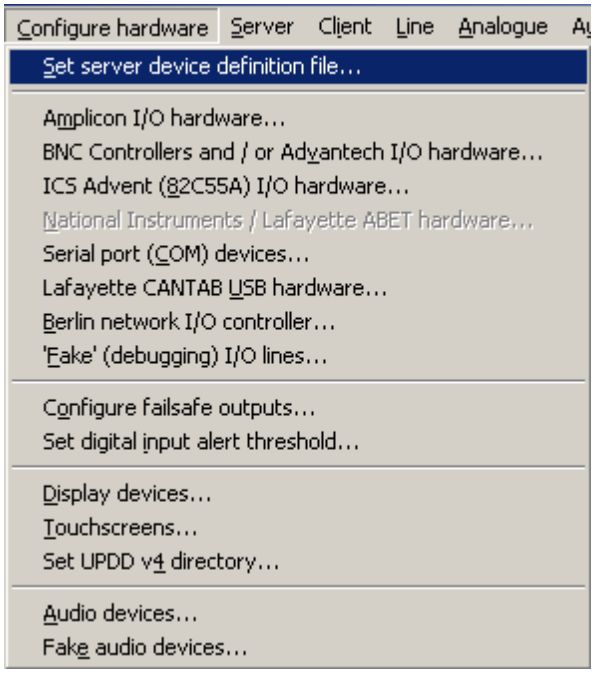
Refresh all views simply starts the views from scratch, ensuring that all the data they are displaying is up to date. You shouldn't need to use this, as all the views should be up to date all the time! It's just here in case I've made a programming error. If you think that the WhiskerServer ever fails to reflect what's going on 'under the hood', feel free to try this; it won't do any harm. If you find a bug, please report it to *rudolf@pobox.com*.

The **Status Bar** option enables you to turn on and off the status bar (at the bottom of the window). The status bar tells you the function of all the menu items (watch it as you move the mouse over the menu).

Full screen expands the server display until it fills the whole screen. Press **Escape** to return it to normal (or use the menus, which are still accessible via ALT-key shortcuts).

6.4.4 Configure hardware

Configure hardware



Set server device definition file. This chooses the text file used to give devices sensible names.

Amplicon I/O hardware. This configures Amplicon digital input/output cards, based on the 82C55 chip.

Advantech I/O hardware. This configures Advantech I/O cards.

ICS Advent (82C55A) I/O hardware. This configures ICS Advent (a.k.a. Kontron) I/O cards based on the 82C55A chip. *Deprecated hardware.*

National Instruments / Lafayette ABET hardware. This configures ABET hardware from Campden Instruments, based on the National Instruments PCI-DIO-96 card, itself based on the 82C55 chip.

Serial port (COM) devices. Configures serial ports for use as simple digital I/O lines.

Lafayette CANTAB USB hardware. Configure Lafayette USB digital I/O devices.

'Fake' (debugging) I/O lines. This configures 'fake' input and output lines; this may be useful for testing programs on a computer that does not have digital I/O hardware fitted.

Configure failsafe outputs. This dedicates special digital output lines to control failsafe relays.

Set digital input alert threshold. This configures a self-monitoring facility, whereby if large numbers of input lines change state simultaneously (which may indicate a cable fault) an alert is generated.

Display devices. Configures display devices (monitors).

Touchscreens. Configures touchscreens.

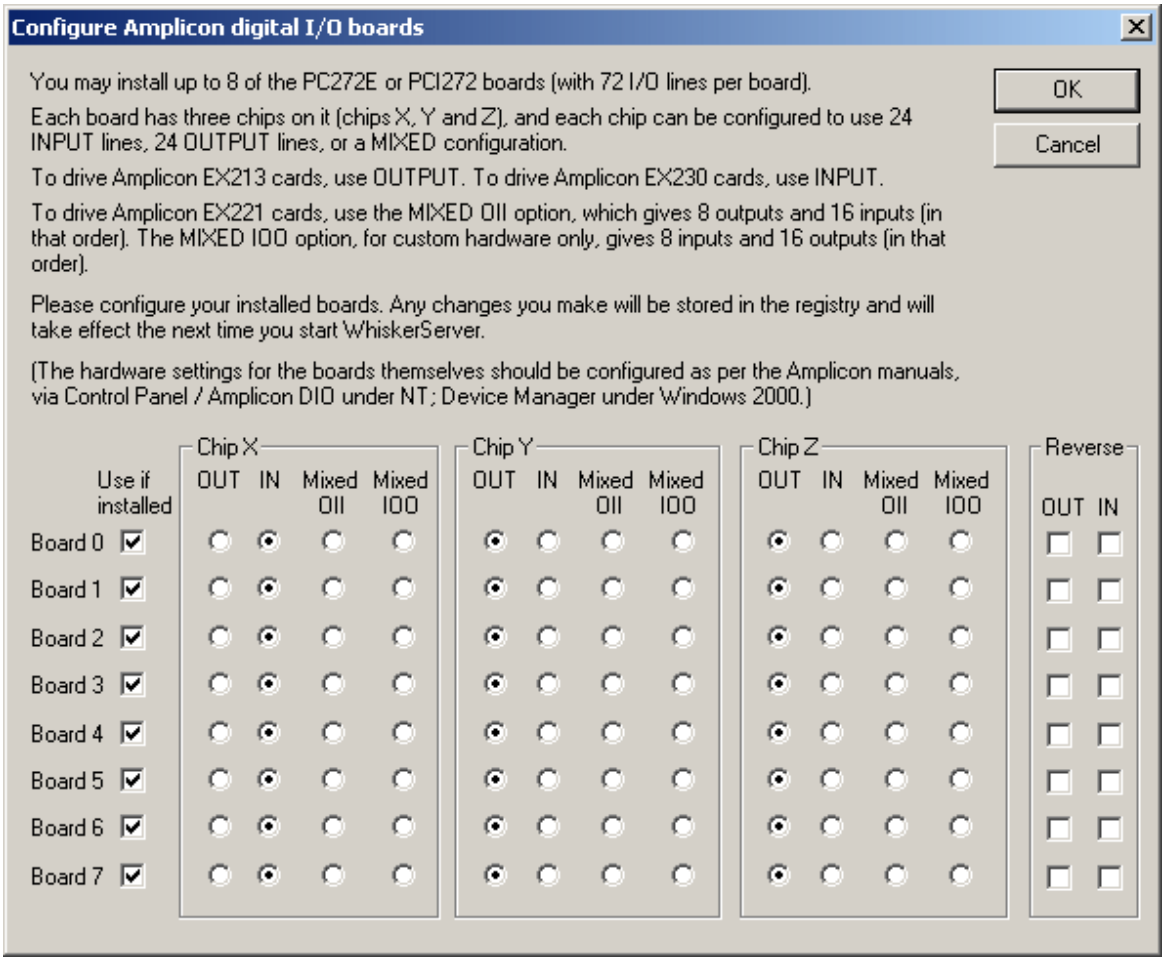
Set UPDD v4 directory. Configure the location where UPDD version 4 lives.

Audio devices. Configures sound output devices.

Fake audio devices. Configures 'fake' audio devices, useful for running tasks that require sound, silently, on a computer with insufficient sound cards.

6.4.4.1 Amplicon I/O hardware

Configure hardware → Amplicon I/O hardware



There may be up to 8 Amplicon digital I/O cards (a limit set by the Amplicon drivers). You can choose whether to use each one, if it is installed. (Choosing to use a nonexistent card does no harm! So by default, all the "use" boxes are ticked.) Each card has three 82C55 chips, termed X, Y and Z. Each chip controls 24 lines, and can be configured to have a 24-input card plugged into it, a 24-output card, or a 'mixed' card with 16 inputs and 8 outputs (the EX221; the outputs are the first eight lines, so it's denoted OII here). From 22 August 2006, WhiskerServer also supports custom hardware with a different mix (8 inputs, then 16 outputs, denoted IOO here).

You should set the map up so it matches your hardware.

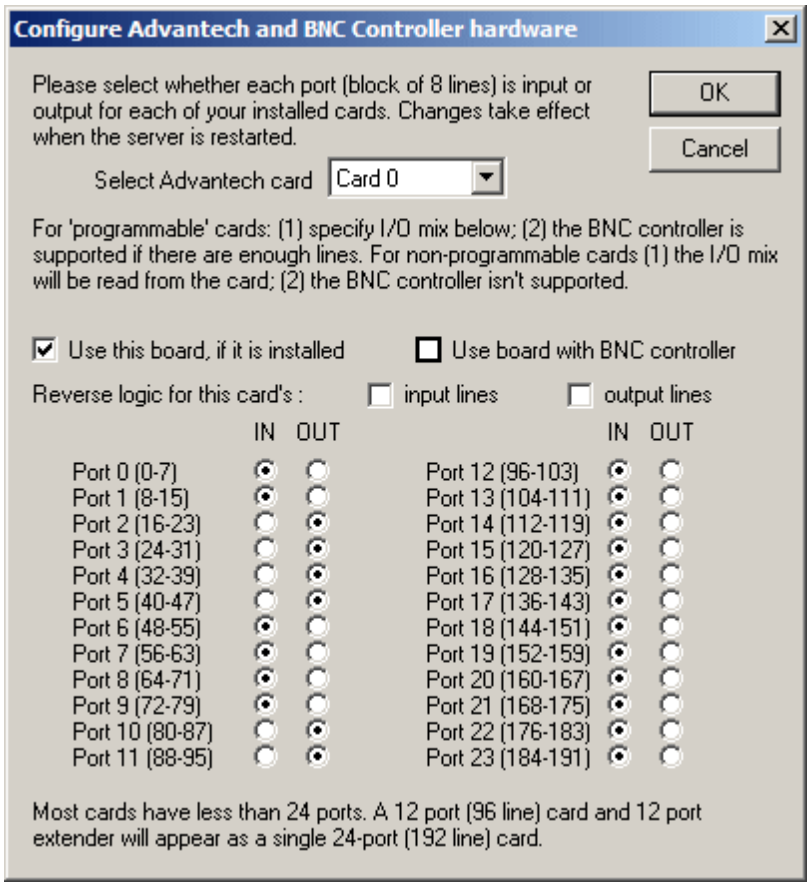
You can also choose whether to reverse the logic of the card's inputs and/or outputs.

Remember, to configure the interrupts and port addresses for each card, use *Start* → *Settings* → *Control Panel* → *Amplicon DIO* under Windows NT, or *Start* → *Settings* → *Control Panel* → *System* → *Hardware* → *Device Manager* → *Amplicon DIO* under Windows 2000.

Changes will not take effect until you restart the WhiskerServer program.

6.4.4.2 Advantech / BNC controller I/O hardware

Configure hardware → *Advantech I/O hardware*



You may have up to 8 Advantech cards installed on a machine. You can choose whether to use each one, if it is installed. (Choosing to use a nonexistent card does no harm, so by default, the "use" box is ticked.)

You may also choose whether to reverse the logic of the inputs and/or outputs. This setting will effect all lines controlled by the card, whether they are attached via a BNC system or directly attached to the Advantech hardware.

Changes will not take effect until you restart the WhiskerServer program.

Advantech cards may be **programmable** (you can tell it which ports are to be used as inputs and which as outputs) or **non-programmable**. The port map shown only applies to programmable cards - for non-programmable cards, the input/output mix is read from the card itself when

Whisker (re)starts (and the port I/O map in this dialogue box is ignored). The BNC controller is only supported via programmable cards.

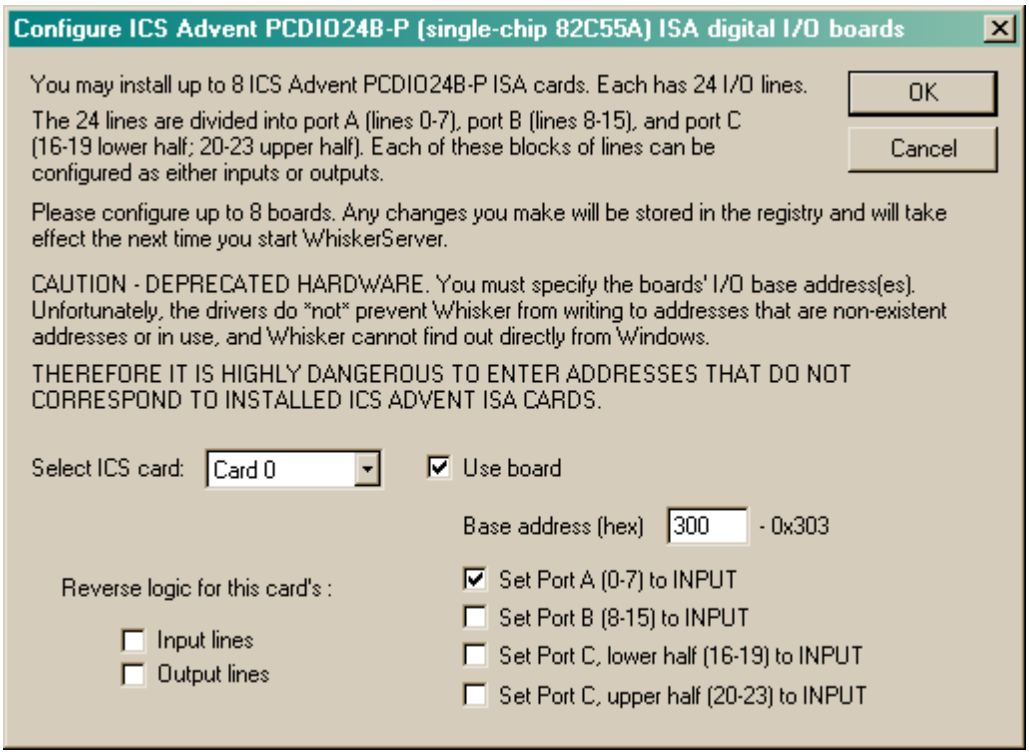
If "Use Board with BNC controller' is ticked" Whisker will attempt to find a Campden BNC system attached to this card.

Please Note:

- If a BNC system is attached and powered up when WhiskerServer starts, the number and configuration of all attached IO cards will be detected automatically, and all the cards will be available.
- If no system is attached, or the BNC controller system is not powered up when the WhiskerServer starts, none of the IO cards will be available (i.e. neither lines from the Advantech card nor BNC controller system) will be available.
- This option can only be selected with one of the installed Advantech cards at any one time (multiple controllers can, however, be 'piggy backed' from a single card).

6.4.4.3 ICS Advent I/O hardware

Configure hardware → ICS Advent (82C55A) I/O hardware



You may have up to 8 ICS Advent cards (82C55A chips). For each card, set the card's I/O base address, and choose which ports are to be used as inputs and which as outputs. You can also choose whether to reverse the logic of the card's inputs and/or outputs. Changes will not take effect until you restart the WhiskerServer program.

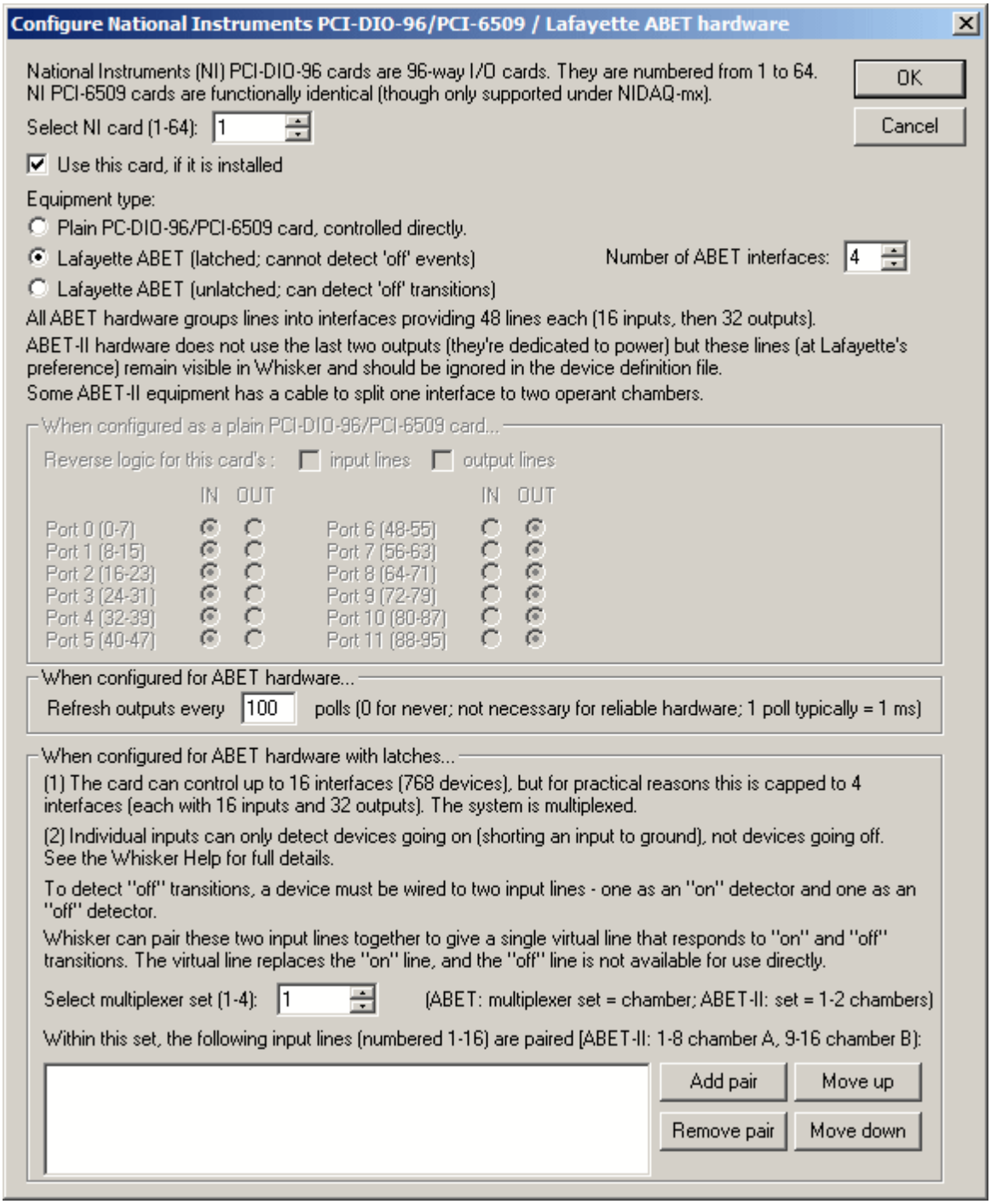
This hardware is supported as it was/is used by the original Monkey CANTAB system (supplied by Cambridge Cognition Ltd). The original Monkey CANTAB was a DOS-based program that expects a single 82C55A chip at I/O address 0x300.

For details of I/O address assignment and why this is a dangerous card, see

- Installing ICS Advent hardware
- The ICS Advent PCDIO24B-P card

6.4.4.4 National Instruments / Lafayette ABET hardware

Configure hardware → National Instruments / Lafayette ABET hardware

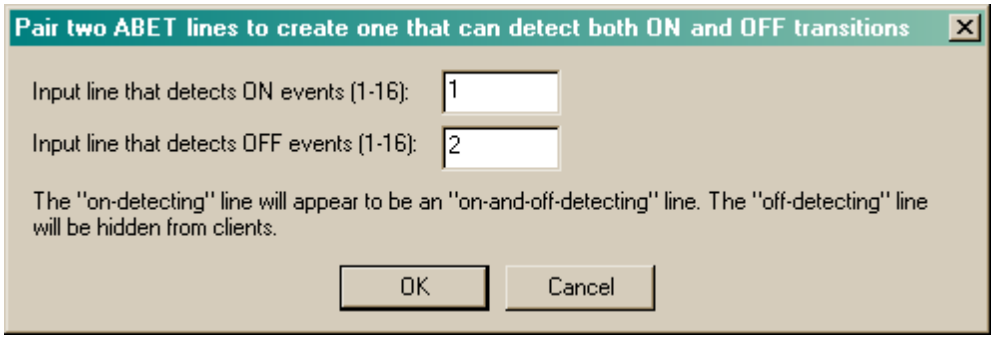


This configures ABET hardware from Lafayette (USA) or Campden Instruments (UK), based on the National Instruments (NI) PCI-DIO-96 card, itself based on the 82C55 chip.

- By default, the newer **NI-DAQmx** library is preferred to the older **NI-DAQ Traditional** library for communicating with NI cards. This option chooses the order in which the libraries are tried (tick for mx then Trad; untick for Trad then mx).
- Select an NI **card to configure.** National Instruments devices are numbered from 1-64 by the driver software (e.g. **National Instruments *Measurement & Automation Explorer***) supplied with the PCI-DIO-96 card.
- Choose **whether to use** the selected device. It does no harm to attempt to use non-existent cards, so by default all cards are selected.
- Choose whether the card is to be used on its own, as a plain 96-way digital I/O card, or **whether it has Lafayette ABET hardware plugged into it, and of which sort.**
- If you have ABET equipment, **specify the number of interfaces** attached to it (from 1-4). Each interface communicates with 48 digital lines, of which 16 are inputs and 32 are outputs (though in some configurations only 30 outputs are actually used; see below). (The I/O card itself could deal with 16x48 lines using this multiplexing system, but there is a cap of 4x48 with the ABET multiplexer for practical and timing reasons.)

  PLEASE NOTE: The ABET hardware with latches cannot detect OFF transitions on inputs (input devices going off) using a single wire -- merely devices going on. To detect devices (e. g. levers) going off as well as on, you need to wire them to **two** inputs -- see below.

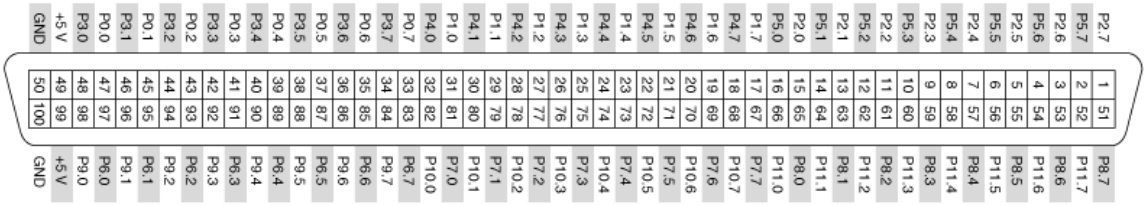
- If you are using the NI PCI-DIO-96 card as a **plain 96-way digital I/O card,** choose whether each of the twelve *ports* (numbered from 0 to 11) should be configured for input or output. Each port controls 8 digital lines. By default, the logic used by the card is the TTL system: +5 V DC means "on", and 0 V means "off". You can choose whether to **reverse the logic for the input or output lines.**
- If you are using the ABET system from Lafayette, the configuration is very simple - all the details are handled for you. All you need to do, *if you wish and if you have the ABET hardware with latches that cannot detect OFF transitions*, is to **configure pairs of input lines to be merged into a single virtual line that can detect devices going OFF as well as going ON.** Full details are given below. You can **add, remove,** and re-order the pairs (re-ordering is merely cosmetic!).



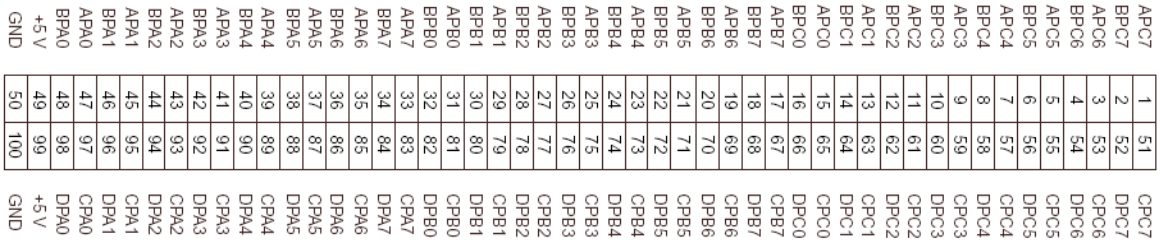
- Oh, one more thing (Oct 2009)... some ABET hardware is quite sensitive to static electricity (report from Lafayette, Oct 2009). To deal with this, Whisker can refresh the outputs (to make sure they're doing what they're meant to be doing). For ABET hardware, you can specify how often to do this ("**Refresh outputs every X polls**"). Typically, one poll is 1 millisecond (see Server > Set internal timer resolution). The default is every 100 polls (every 100 ms). Setting this to zero disables this additional hardware output refresh.

PCI-DIO-96 card and cable connections to Lafayette ABET hardware

The **NI PCI-DIO-96** card (provided by Lafayette as item **81404**) is installed in the computer. It has a 100-way socket on the back. The card pinouts (for a NI 6508/6508/PCI-DIO-96) look like this (using the pin names as referred to by the NI-DAQmx software):



or, with different pin names (as referred to by the older "Traditional NI-DAQ" software):



Internally, the card has four 82C55 chips, each with 24 digital I/O lines. The card is addressed in terms of 12 ports, each with 8 digital I/O lines; each port is configured as a whole for input or output.

Each 82C55 chip can generate an interrupt upon a low-to-high transition of PC3 and PC0 (see the PCI-DIO-96 manual: *Theory of Operation / Interrupt Control Circuitry*). That means that wire APC0, APC3, BPC0, BPC3, CPC0, CPC3, DPC0, DPC3 can generate interrupts.

If you are using the ABET hardware, then into the 100-way socket is plugged a cable (an **R1005050** cable) that ends in two 50-way plugs. Their pinouts are as follows:

APC7	1	2	BPC7	CPC7	51	52	DPC7
APC6	3	4	BPC6	CPC6	53	54	DPC6
APC5	5	6	BPC5	CPC5	55	56	DPC5
APC4	7	8	BPC4	CPC4	57	58	DPC4
APC3	9	10	BPC3	CPC3	59	60	DPC3
APC2	11	12	BPC2	CPC2	61	62	DPC2
APC1	13	14	BPC1	CPC1	63	64	DPC1
APC0	15	16	BPC0	CPC0	65	66	DPC0
APB7	17	18	BPB7	CPB7	67	68	DPB7
APB6	19	20	BPB6	CPB6	69	70	DPB6
APB5	21	22	BPB5	CPB5	71	72	DPB5
APB4	23	24	BPB4	CPB4	73	74	DPB4
APB3	25	26	BPB3	CPB3	75	76	BPB3
APB2	27	28	BPB2	CPB2	77	78	DPB2
APB1	29	30	BPB1	CPB1	79	80	DPB1
APB0	31	32	BPB0	CPB0	81	82	DPB0
APA7	33	34	BPA7	CPA7	83	84	DPA7
APA6	35	36	BPA6	CPA6	85	86	DPA6
APA5	37	38	BPA5	CPA5	87	88	DPA5
APA4	39	40	BPA4	CPA4	89	90	DPA4
APA3	41	42	BPA3	CPA3	91	92	DPA3
APA2	43	44	BPA2	CPA2	93	94	DPA2
APA1	45	46	BPA1	CPA1	95	96	DPA1
APA0	47	48	BPA0	CPA0	97	98	DPA0
+5 V	49	50	GND	+5 V	99	100	GND

- The two 50-way plugs then plug into a **Lafayette Instruments 81403 converter box.**
- From that emerge two 36-way Centronics-style cables (part **81405,** with suffixes -6 or -25 for the length in feet), which plug into a **Lafayette Instruments 81401 ABET starter interface,** which also receives a 28 V DC power supply, via an **81410 power supply cable.** This 81401 interface controls the first chamber (chamber number 1). *It's impossible to plug the pairs of Centronics cables in backwards, since the gender orientation of each cable in the pair is different.*
- You can stack further interfaces (in the form of **Lafayette Instruments 81402 expander interfaces**) on top of the 81401 (they slot together). Each controls one operant chamber, numbered consecutively. Up to this point, the system is using 5 V TTL logic.
- From each of the interfaces (81401 and 81402) emerge two 25-way cables (part **81406,** with suffixes -10, -25, or -50 for the length in feet). From this point on, the system uses 28 V. The two 25-way cables from one 81401/81402 interface plug into a **Lafayette Instruments 81408 full I/O module connection block,** which has screw terminals that devices can be connected to (16 inputs and 32 outputs) plus 10 modular six-pin (RJ-25?) connectors to control intensity modules (intensity is controlled in "7 discrete steps", so this six-pin connector presumably has a three-bit intensity signal plus power and ground and something else each). *It's impossible to plug the pairs of 25-way cables in backwards, since the gender orientation of each cable in the pair is different.*
- An alternative to the 81408 is an **81409 mini I/O module,** which has screw terminals for 8 inputs and 15 outputs and 5 modular connectors for intensity modules.

The electronic specifications for the ABET hardware are:

- Maximum number of interfaces: 16
- Per interface, maximum of 32 outputs, 16 inputs, 10 intensity modules.

  - *RNC: Since only 50 pins go into the screw terminal block, and 32+16=48, I'd imagine that you can't use 32 outputs and 10 intensity modules independently. Yes, confirmed by the manual: Intensity block 1 uses output data lines 1-3; intensity block 2 uses outputs 4-6; intensity block 3 uses outputs 7-9; intensity block 4 uses outputs 10-12; intensity block 5 uses outputs 13-15; intensity block 6 uses outputs 16-18; intensity block 7 uses outputs 19-21; intensity block 8 uses outputs 22-24; intensity block 9 uses outputs 25-27; intensity block 10 uses outputs 28-30.*
- Supply voltage: minimum 22 V DC, maximum 35 V DC.
- Quiescent current per interface: 40 mA.
- Maximum output current per interface: 2 A.
- Maximum current per output: 0.5 A.
- Maximum output line voltage: 50 V DC.
- Maximum input line voltage: 50 V DC.
- Maximum length between computer and interface stack 25', and between interface stack and environments (operant chambers) 100'.
- Power connector: 2.1 mm centre-positive DC jack
- Timing resolution: 1 ms.

The signal connections to actual devices work like this:

- An output device is connected to (1) a +28 V DC rail on the screw terminal, labelled "+28 VDC", and (2) the controlling terminal, labelled "OUTPUTS 1...32". **Each ABET output line provides a connection to common or ground when it is activated** (and is, presumably, not connected to anything when it is not activated). Output devices are described as being wired either with a two-wire connection (e.g. red = +28 V, black = output) or a three-wire connection (e.g. red = +28 V, white = output, black = common/ground).
- An input device is connected to (1) a 0 V DC common/ground terminal, labelled "COMMON", and (2) the sensing terminal, labelled "INPUTS 1...16". The input lines are pulled up to a logic level high (actually about +4.5 V); **the input devices must provide a switch closure to common (ground) to activate the input.** Input devices are two-wire (e.g. red = input, black = common).

So it's a "short-to-ground = active" system.

Multiplexing system for Lafayette/Campden ABET hardware

The 96-way digital I/O card is used to control up to 16 operant chambers, each with up to 16 inputs and 32 outputs. That's 48 lines per chamber, and therefore up to 768 devices. Clearly, some trickery is needed to get a 96-way card to control 768 devices, and that trickery is *multiplexing*. This is the system in use:

```
On the PCI-DIO-96:                                      RNC:
       Address outputs (4 bits):   APA<3:0>             Port 0.3 to port 0.0
- output
       Output enable (1 bit):      APA<4>               Port 0.4
- output
       Service request inputs:     DPA<0:7>(chambers 1-8)     Port 9
- input
                                   DPB<0:7>(chambers 9-16)    Port 10
- input
       Data inputs:                CPA<0:7>(inputs 1-8)       Port 6
- input
```

```
                                  CPB<0:7>(inputs 9-16)      Port 7
- input
      Data outputs:               APB<0:7>(outputs 1-8)      Port 1
- output
                                  APC<0:7>(outputs 9-16)     Port 2
- output
                                  BPA<0:7>(outputs 17-24)    Port 3
- output
                                  BPB<0:7>(outputs 25-32)    Port 4
- output

                                                             BPC (port 5) not
used, or used for IRQs, it seems
                                                             CPC (port 8) not
used, or used for IRQs
                                                             DPC (port 11) not
used, or used for IRQs

Alternatively:

#define ABET_ADDRESS_OUTPUT_PORT         0      // port  0 = output (APA) -
address outputs (bits 0-3) and output enable (bit 4)
#define ABET_DATA_OUTPUT_PORT_A          1      // port  1 = output (APB) -
controlling selected chamber outputs 1-8
#define ABET_DATA_OUTPUT_PORT_B          2      // port  2 = output (APC) -
controlling selected chamber outputs 9-16
#define ABET_DATA_OUTPUT_PORT_C          3      // port  3 = output (BPA) -
controlling selected chamber outputs 17-24
#define ABET_DATA_OUTPUT_PORT_D          4      // port  4 = output (BPB) -
controlling selected chamber outputs 25-32
#define ABET_UNUSED_PORT_A               5      // port  5 = UNUSED (BPC)
#define ABET_DATA_INPUT_PORT_A           6      // port  6 = input  (CPA) -
monitoring selected chamber inputs 1-8
#define ABET_DATA_INPUT_PORT_B           7      // port  7 = input  (CPB) -
monitoring selected chamber inputs 9-16
#define ABET_UNUSED_PORT_B               8      // port  8 = UNUSED (CPC)
#define ABET_SERVICEREQUESTPORT_A        9      // port  9 = input  (DPA) -
service request inputs for chambers 1-8
#define ABET_SERVICEREQUESTPORT_B        10     // port 10 = input  (DPB) -
service request inputs for chambers 9-16
#define ABET_UNUSED_PORT_C               11     // port 11 = UNUSED (DPC)

Additionally, Vern Davidson (vern@lafayetteinstrument.com) commented that "some of
the I/O is used as an address for the ABET interfaces (81401 and 81402), while the
rest of the I/O on the card are latched inputs from ABET interfaces, latched
outputs... and interrupt requests (IRQs)". Now APC0, APC3, BPC0, BPC3, CPC0, CPC3,
DPC0, DPC3 are the only pins that can generate interrupts - so maybe BPC, CPC,
and/or DPC are used for interrupts (6 potential interrupt-generating lines).

The lines are active high (high voltage = on, low voltage = off).

The multiplexing system works like this:

  "In the ABET software from Lafayette, the 16 service request lines are constantly
  monitored.
  When an input to a chamber is activated, the service request line for that
  chamber goes high.
  The time that occurs is recorded as the event time; the chamber is then addressed
```

```
  to determine which input occurred.
  The address outputs are tied to a multiplexer:
       0000 = chamber 1
       0001 = chamber 2
       0010 = chamber 3
       etc.
  The output enable line is used to latch the outputs on the addressed chamber,
  like this:
       Set the address.
       Set the outputs.
       Pulse the output enable line.
       The outputs are now latched.
  Please note that the inputs are also latched. Pulsing the output enable line will
  reset the input latches, so it is necessary to read the inputs before setting the
  outputs."

Much fiddling with code, lightbulbs, and switches...
WE CAN'T DETECT OFF TRANSITIONS.
This must be what the multiplexer hardware (84101/84102 interface) does:
```

- Scan the physical hardware or respond automatically to ON transitions. (Perhaps the multiplexer itself has 82C55s in, and relies on their change-detection system? Otherwise it could perhaps respond to OFF transitions too...)
- If an input line has gone ON, and the OUTPUT-ENABLE bit is clear, (1) set the [line corresponding to the] chamber's bit in the SERVICE-REQUEST port, (2) trigger an interrupt (somehow) to request service from software that relies on interrupts (note: Whisker doesn't rely on interrupts, and uses 1 kHz polling instead).
- When the computer writes the ADDRESS-OUTPUT port, (1) latch the appropriate chamber's INPUT states - meaning the lines that have just gone on, i.e. the stored/pending transitions, not the lines that *are* on - into the DATA-INPUT lines. THINK OF THE INPUT PORT AS THE "PENDING ON-TRANSITION NOTIFICATION" PORT.
- When the computer writes the OUTPUT-ENABLE bit, (1) copy the DATA-OUTPUT state into the chamber's output lines; (2) clear all on-transition notifications.
- Note that the computer must clear the OUTPUT-ENABLE bit to allow the multiplexer to respond to inputs again (see above).

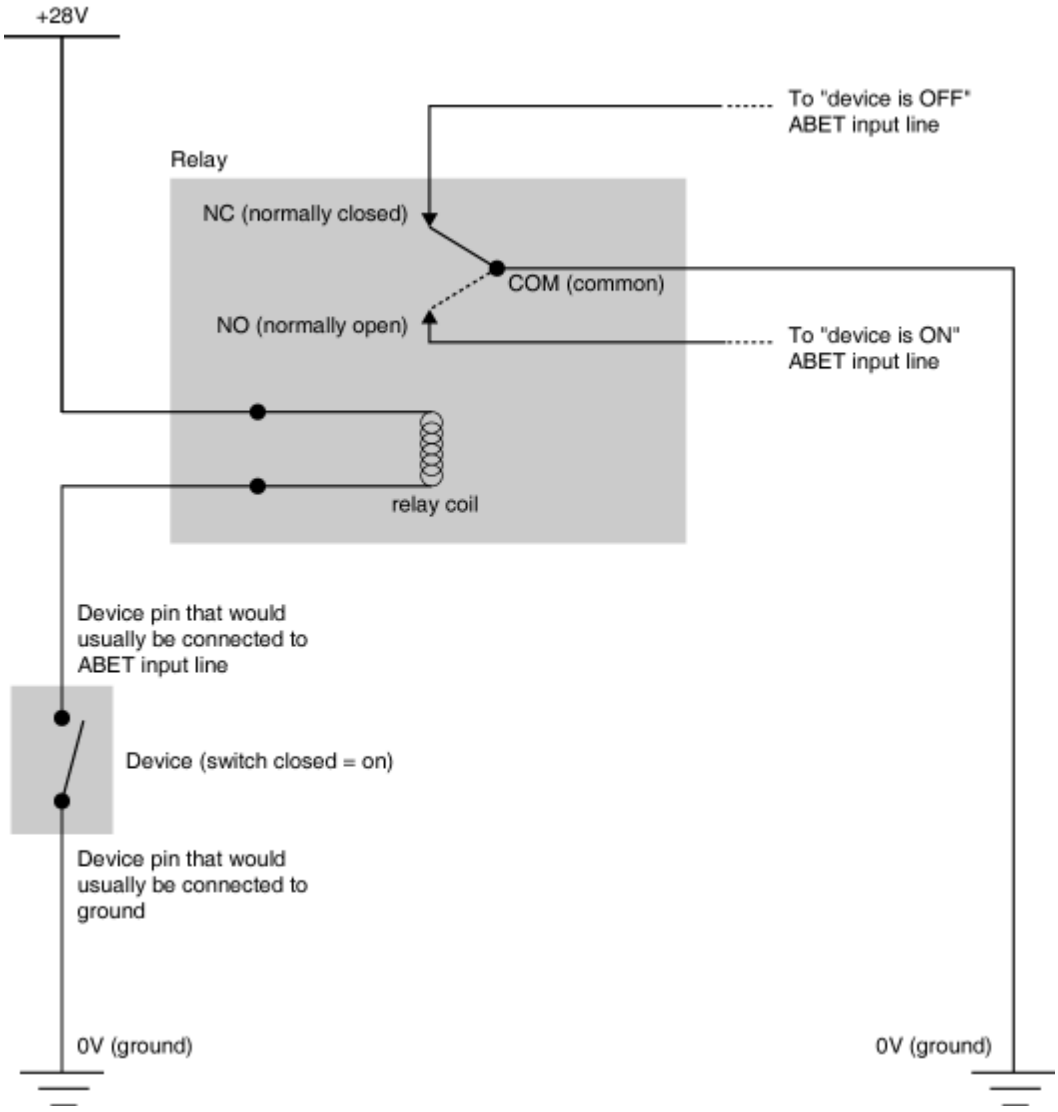
Detecting OFF transitions using ABET hardware and Whisker

Whisker copes with the lack of OFF-detecting hardware like this:

- Lines that cannot detect OFF events respond to ON transitions by going on for a single poll (typically 1 ms).
- **You cannot set OFF events on an input line that does not detect them.** (This is to prevent you from relying on OFF events whose timings -- one poll -- are fictional, and do not reflect the real-world device.)
- **You cannot read the state of an input line that cannot detect OFF events.** In the status display, the number of on *transitions* is shown rather than the current state of the line.
- You can dedicate a second line to detecting OFF events. You would wire these physically so that the device going on triggers an ON event on one line, and the device going off triggers an ON event on a second line. (Lafayette do this for their **80113 retractable lever,** the only device they list that appears to notify the hardware of OFF events.)
  - **Line can thus be paired: two physical lines that only detect ON events are combined to give a virtual line that responds to ON and OFF events.**
  - The virtual line starts in the "off" state.

- When the "on" line goes on, the virtual line goes on. When the "off" line goes on, the virtual line goes off. (If both go on simultaneously, the virtual line stays or goes off -- *and neither an ON nor an OFF event is generated by the server if the line was previously off*.)
- The combined virtual line can be treated normally (i.e. you can set OFF events on it, and you can read its state).
- The virtual line has the line number of the on-detecting line. The "off-detecting line" appears in a different colour on the server's display. It cannot be claimed, and it cannot be pegged (you can peg the corresponding virtual line instead). If you read its state on the console, it will read as OFF.

The easiest way to wire a device physically in order to make an "off-detecting" as well as an "on-detecting" line, using a device that shorts its signal line to ground when it goes on, would be like this, with a 28V relay (assuming the relay and device can cope with the resulting applied voltages!):



The ABET-II system

ABET-II changes the wiring system from ABET. In ABET, each operant chamber had 16 inputs and 32 outputs (the first 16 and the last 32 lines, respectively, within each group of 48 lines corresponding to each chamber, and numbered consecutively as you'd expect).

ABET-II dispenses with two outputs in each set of 48. Then, each set can either driver one operant chamber ("not split") or two ("split"), the splitting being done with a cable.

Within each of the (16) sets of 48 lines, then, the following wiring diagram is used:

ABET-II (not split)
- lines 0-15 = chamber inputs 0-15
- lines 16-47 = chamber outputs 0-31 (except that lines 31 and 47 are unused)

ABET-II (split)
- lines 0-7 = chamber A inputs 0-7
- lines 8-15 = chamber B inputs 0-7
- lines 16-31 = chamber A outputs 0-14 plus one unused line (line 31)
- lines 32-47 = chamber B outputs 0-14 plus one unused line (line 47)

The "no-latch" system (probably to be called ABET-III or ABET-Touch)

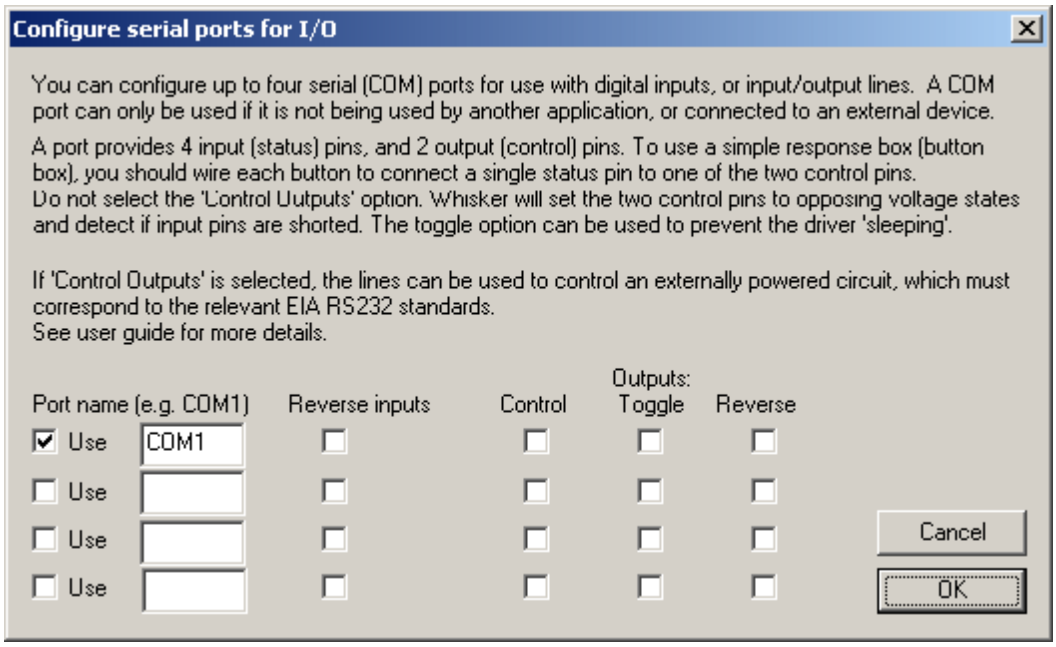
The "no-latch" system is a modification to the ABET-II multiplexer (though it could in principle be applied to the ABET system as well). It removes the significant problem of not being able to detect "off" transitions.

See above for the description of how the multiplexer works normally - that is, when a device is activated, the input line and service request were set, and were cleared subsequently following a read of the inputs. The service request and input would not be set again until the device was deactivated and then reactivated.

In this modification, with the latches removed, the service request and input stay activated as long as the device is active. (Reading the service request and input will clear them but they will be set again immediately after reading.) When the device is deactivated released the corresponding input and resulting service request will clear upon reading and stay cleared.

6.4.4.5 Serial port (COM) devices

Configure hardware → Serial port (COM) devices.



You may configure up to 4 serial ports for use with digital I/O.

Whisker will attempt to use any port named in a row which has the **Use** option checked. The **Port name** entered in the box must be the port name as used by Windows. (These names are generally COM1, COM2 etc., and can be seen in the Device Manager).

The **Reverse Inputs** boxes change which of the states (SPACE or MARK; see below) correspond to On and Off in Whisker.

The three columns of boxes labelled **Outputs** are used to configure how Whisker uses the *terminal status lines* (outputs)

- The **Control** option should only be selected if you wish to use the modem status line pins of a serial port to control external hardware. If you are connecting a simple response box, as described below, this option must **not** be selected.
- The **Toggle** option, if selected, will toggle the state of the terminal status lines every time the card is polled. This item should not be selected if the lines are being used as outputs for externally powered circuits.

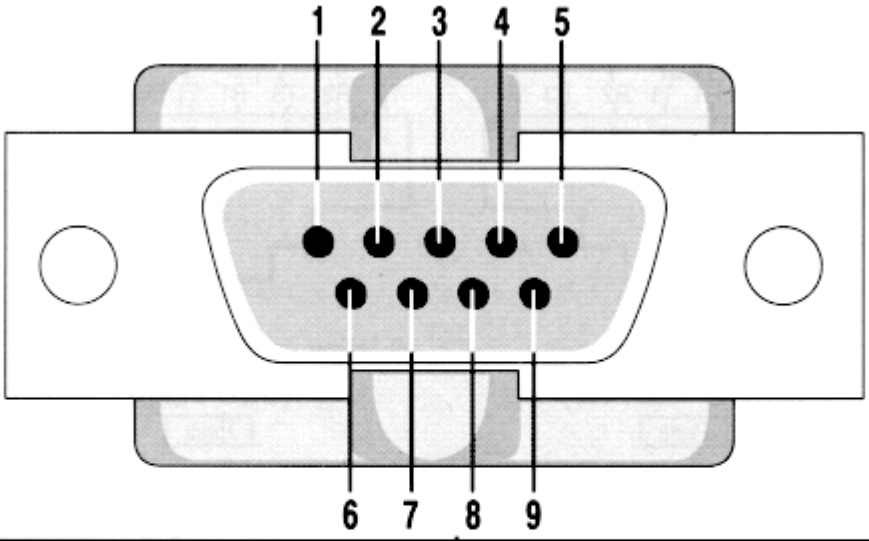
Some laptops have been reported to power down the PCM-CIA serial cards if no lines change state for a certain period, which means that a response box could cease functioning after a period of disuse.

This option is provided as a work around solution to this problem.

- The **Reverse** option reverses which state (SPACE or MARK; see below) correspond to On and Off in Whisker. This has an effect even if the control pins are not being used for output (i. e. when **Control** is not checked). Thus either RTS or DTR can be used as the voltage source for a simple response box.

Using Serial port for IO with a simple response box

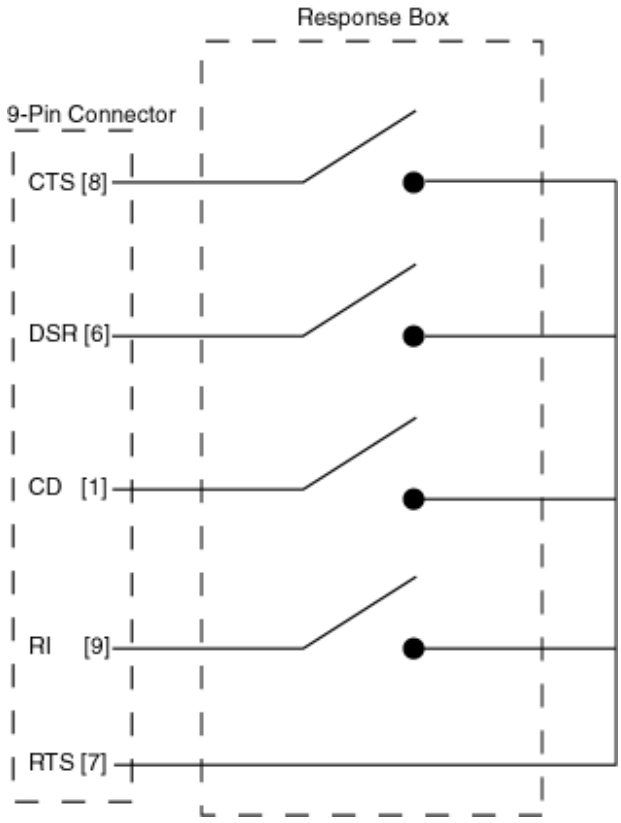
Serial (COM) ports are designed for connecting a computer with a data modem. The standard defines 6 status lines, 4 of which are used to signal modem status, and 2 of which are used to signal terminal (PC) status. The line assignment on a standard 9-pin Serial port is shown below:



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Serial ports can be configured to use the 4 **modem status lines** of a standard serial port (CTS, DSR, CD, and RI) as digital inputs.
By default, the 2 **terminal status lines** (RTS, DTR) will be set such that RTS is On and DTR is Off.

A simple response box can therefore be wired by switching the four modem status independently to lines to RTS, as shown below.



Using Serial port for IO with externally powered devices.

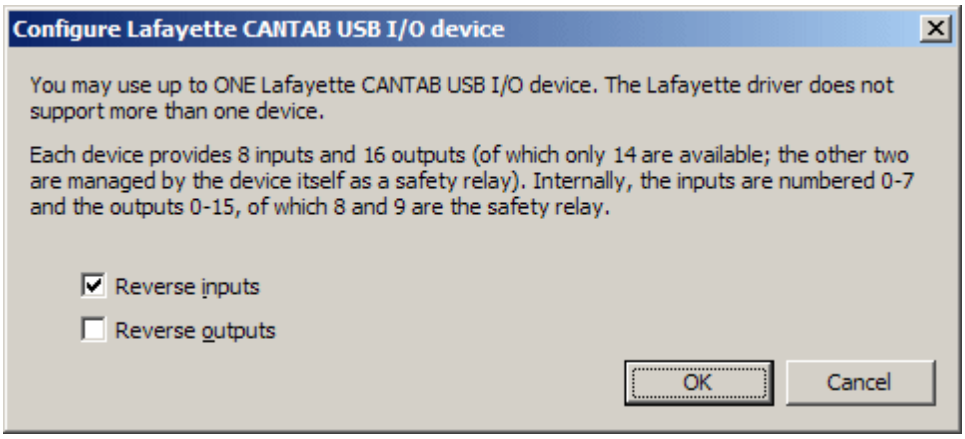
It is imagined that most users will wish to connect simple, passive input devices (e.g. response boxes), wired similarly to that shown above.

However other devices may be used provided the voltages the require and supply are suitable. We don't plan to use Whisker in this way: the digital IO hardware types Whisker supports already give a number of inexpensive options for digital output control. If you wish to use such devices, the two **terminal status lines** (RTS and DTR) can be used by Whisker as ordinary digital outputs. If you wish to use these lines as such, you must select the **Control** and not select the **Toggle** options for the Serial Port.

Technical Note: Serial ports do not use TTL voltages, as used by other devices (e.g. parallel ports), but rather support two different voltage ranges relative to signal ground: SPACE [= +3 to +25V] and MARK [- 3 to -25V], some machines, including many laptops, may use different voltages (the ones here correspond to EIA RS232C standard; see the documentation with the serial card to check the voltage limits).

6.4.4.6 Configure Lafayette CANTAB USB hardware

Configure hardware → Lafayette CANTAB USB hardware.



This is a USB device providing 8 inputs and 14 outputs (plus 2 outputs used by the card as a safety relay and not accessible to Whisker).

The Lafayette driver only allows one such device to be used (as of 9 Aug 2009). Whisker autodetects whether one of these devices is present or not.

The only options available are whether to reverse the logic (on/off) states for inputs and/or outputs.

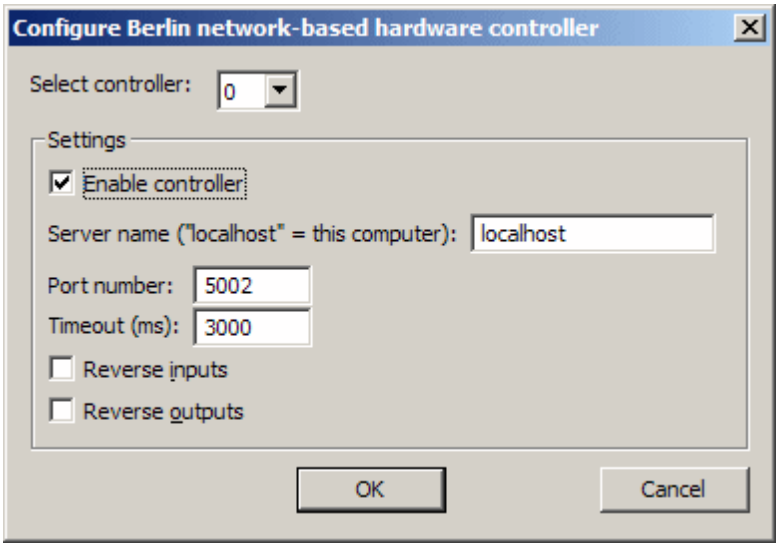
Restart Whisker to apply any changes.

6.4.4.7 Configure Berlin network I/O controller

Configure hardware → Berlin network I/O controller.

This option entered Whisker in May 2010. The biology group in Charité Universitätsmedizin Berlin led by York Winter has network-based I/O controllers; that is, they have software that talks to digital I/O devices. Whisker communicates with this software (thereby supporting their I/O devices) through TCP/IP.

You can configure up to 16 controllers (numbered 0-15). Each has the following options:



Note that Whisker looks for these servers at *startup time*. If it finds them, it adds them to the set of Whisker's digital I/O lines (and if not, then Whisker won't look for them again). If communication with one of the Berlin servers is lost, Whisker flags an errors in its log, and flags all the lines as "FAILED" in its line list - but it does not attempt to reconnect (on the principle that if the connection is unreliable, then the lines shouldn't be used). Clearly, this system is designed to work on internal networks (within a single computer) or local networks of guaranteed connectivity.

The communications specification used by the Berlin servers is as follows:

- TCP/IP
- Messages separated by CR LF (\n\r).
- Nagle algorithm disabled on both sides (necessary for real-time performance).

The command syntax used by the Berlin servers is as follows:
FROM WHISKER TO BERLIN SERVER:

- `GetNumberOfInputChannels` (leads to NumberOfInputChannels message)
- `GetNumberOfOutputChannels` (leads to NumberOfOutputChannels message)
- `SetChannelOn channel` (no response)
- `SetChannelOnPulse channel durationms` (no response)
- `SetChannelOff channel` (no response)
- `GetSensorState channel` (leads to SensorState message)

FROM BERLIN SERVER TO WHISKER:

- `NumberOfInputChannels max` (input channels are numbered from 0 to max-1)
- `NumberOfOutputChannels max` (output channels are numbered from 0 to max-1)
- `SensorState channel state` (no response; "state" is "0" or "1"; may arise spontaneously as the inputs change state, or in response to a GetSensorState message)

6.4.4.8 Set server device definition file

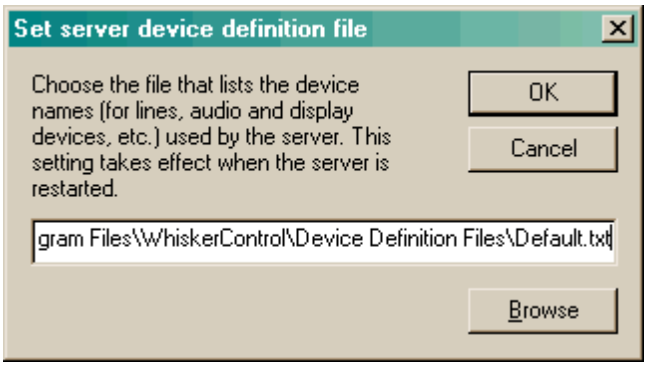
Configure hardware → Set server device definition file

This topic is discussed in detail in the *Programmer's Guide* (see the section entitled The server's device definition file). Essentially, the device definition file is a list telling the server what each device represents in the physical world — for example,

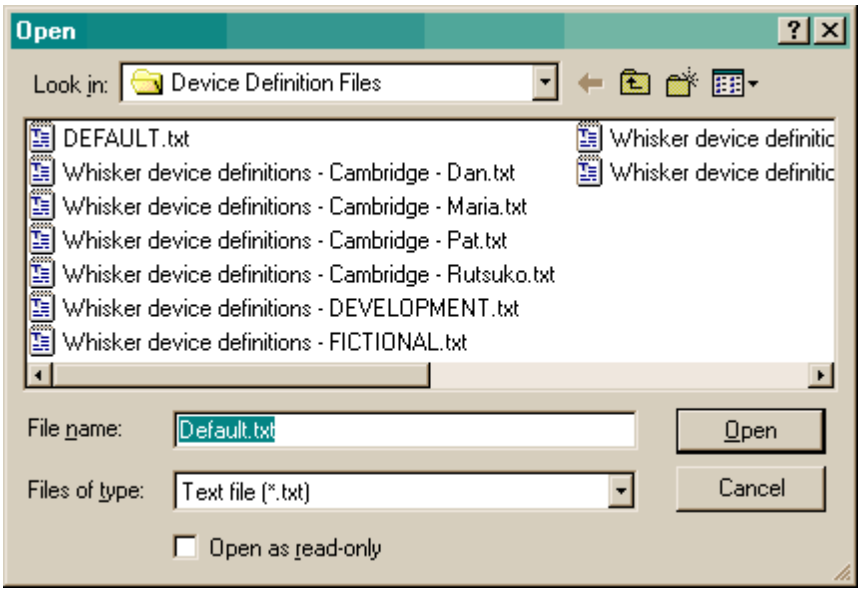
```
line        0       box1        LEFTLEVER
line        1       box1        RIGHTLEVER
line        2       box2        LEFTLEVER
...
display     0       box1        display
...
audio       0       box1        leftsound
audio       1       box1        rightsound
audio       2       box2        leftsound
...
```

This configuration setting is a per-user setting in Windows (i.e. each Windows user can set Whisker up to use a different configuration file).

Each computer is expected to have at least one device definition file reflecting the way that that particular computer is wired up. Create the definition file; tell the server where it is in the dialogue box:



… clicking *Browse* to find and specify the file:

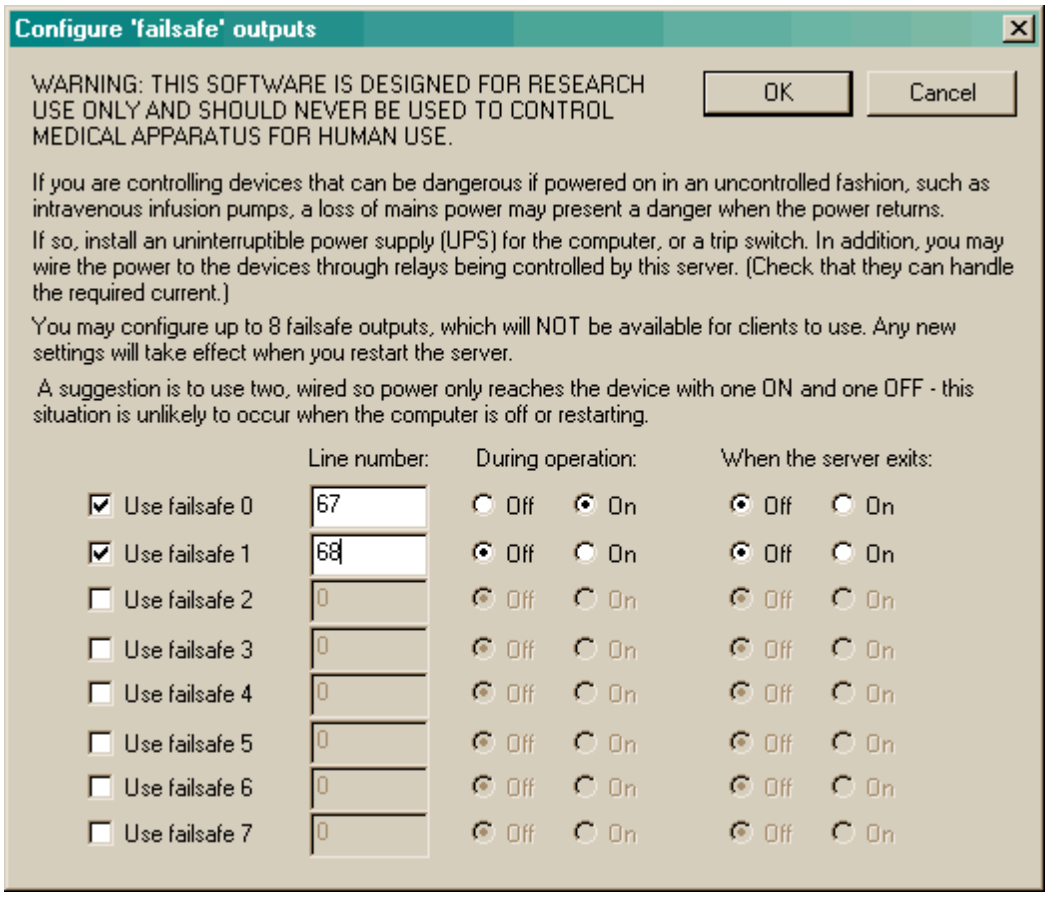


… and then clients can refer to the devices by name rather than by number (as described in the *Programmer's Guide;* see The server's device definition file). If you change the wiring of your computer, simply edit the device definition file; you will not need to change your client tasks.

The device definition file is only read when the server starts.

6.4.4.9 Configure failsafe outputs

Configure hardware → Configure failsafe outputs



In the *Installation and Hardware Guide* (section entitled Danger — Critical Devices), we discussed the issue of wiring the power supply to critical devices (such as intravenous infusion pumps) in a fail-safe manner. One way of doing this is to connect two relays to two output lines, requiring one relay to be on and one to be off before any devices get power.

A circuit for this is given in the *Installation and Hardware Guide*, together with instructions for configuring failsafe outputs in this dialogue box.

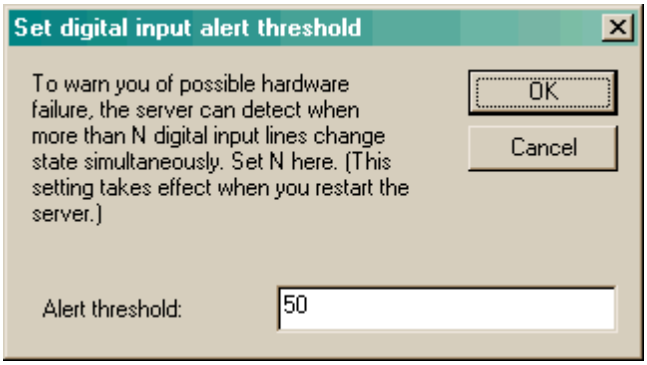
Essentially, you may dedicate lines as failsafes by ticking a 'Use failsafe…' box and filling in the line number you wish to use. You may then specify the state (on/off) that you want the failsafe line to be in when the server is running (i.e. when you want power to your devices) and what state it should be in when the server quits (i.e. when you want power cut off). This line will then be *unavailable* for clients to use.

See the Hardware Guide for an example of setting up failsafe devices.

6.4.4.10 Set digital input alert threshold

Configure hardware → Set digital input alert threshold

This configures a self-monitoring facility, whereby if large numbers of input lines change state simultaneously (which may indicate a cable fault) an alert is generated.



The dialogue box allows you to enter a number. If this number of *input* lines change state *simultaneously* (meaning within one poll interval, typically 1 ms), a message will be written to the server's event log, saying

```
*** ALERT: 50 input lines changed state simultaneously
```

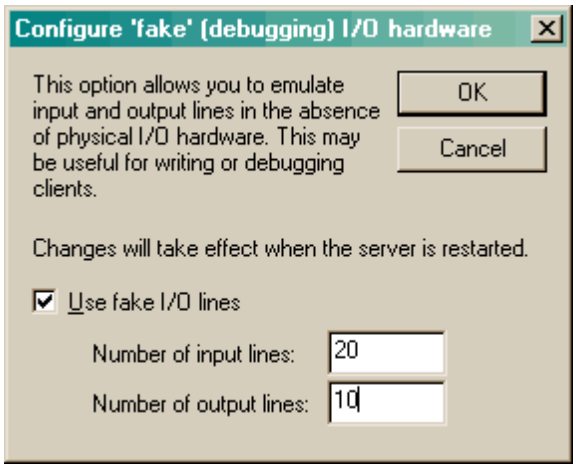
This may indicate a loose cable. (Typically, if a cable to an Amplicon card works loose, the inputs are disconnected, and a disconnected input is perceived as being on. If the cable is just making contact, large numbers of inputs may appear to flash on and off together.)

If you set this number very low (e.g. 2), you are likely to get the alert message during normal operation.

6.4.4.11 Fake (debugging) I/O lines

Configure hardware → Fake (debugging) I/O lines

This configures 'fake' input and output lines; this may be useful for testing programs on a computer that does not have digital I/O hardware fitted.



Fake lines have no connection to physical devices, but behave in all other respects like genuine

lines. You may turn fake lines on and off using the line-control facilities available for genuine lines (see the Line menu). This enables you to test client software that expects a response via input lines.

The other reason for using fake I/O lines is to be able to run a client that expects a certain device to be present, with that device nonexistent or disabled. (Simply create spare I/O lines and set your server's device definition file to refer to them.)

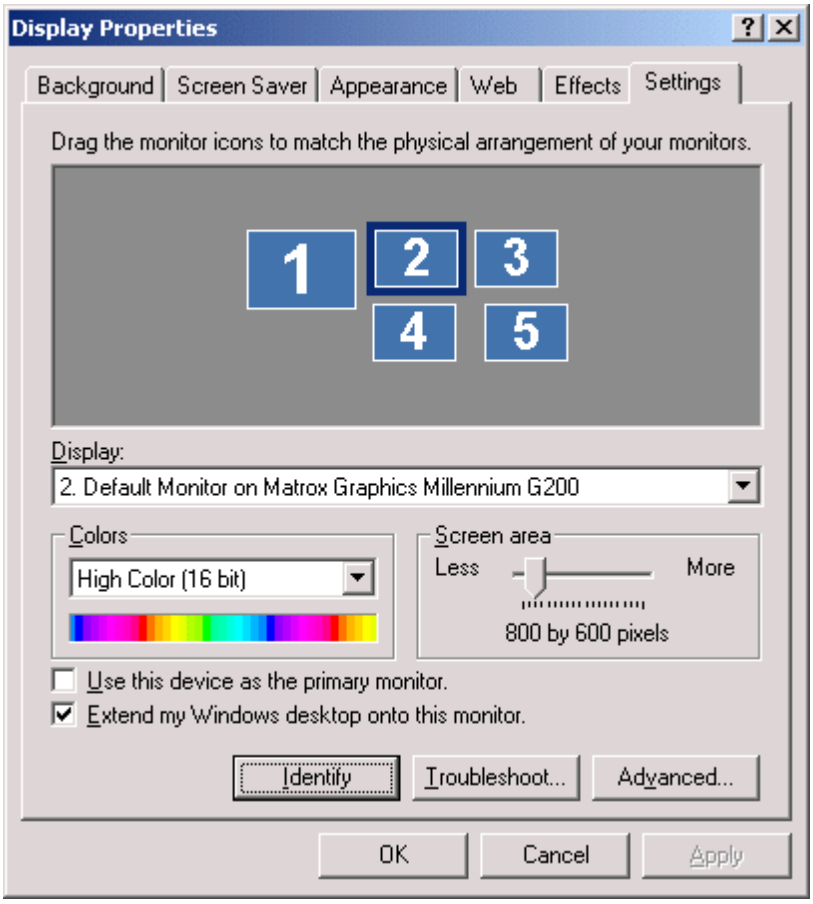
Fake inputs are numbered before fake outputs, but after all genuine digital I/O cards.

6.4.4.12 Display devices

Configure hardware → Display devices

Configures display devices (monitors).

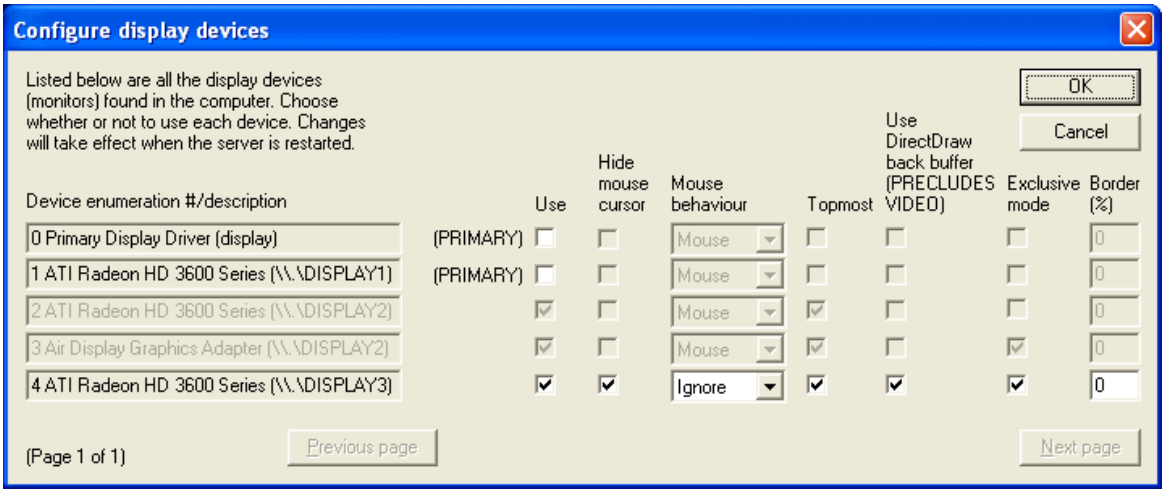
Before configuring displays for use with Whisker, configure the monitors in the Windows Control Panel. On a multimonitor computer running Windows 2000, choose *Start → Settings → Control Panel → Display → Settings* and you will see something like the following:



The Control Panel allows you to set the relative position of your monitors (by dragging the images around), set the resolution of each, etc. Whisker is only able to use monitors that are part of the Windows desktop (this is not deliberate, but appears to be a bug in the Windows DirectDraw enumeration function). Therefore, any display that you wish to use with Whisker must have *Extend my Windows desktop onto this monitor* ticked.

One and only monitor will have *Use this device as the primary monitor* ticked. I will refer to this as the *primary monitor*. It is the one that the Windows desktop starts from; programs that are not multimonitor-aware default to opening on the primary monitor.

Within Whisker, when you choose *Configure hardware* → *Display devices* you will see the following dialogue box:



N.B. This screenshot is from a computer that had three monitors attached, of which one (DISPLAY2) is currently disabled. Note that individual displays may appear more than once, under different names and/or via different driver systems.

Use device. You may enable or disable each display device. If the device is enabled, the following options become available:

Hide mouse cursor. Makes the mouse cursor invisible when it enters this display area.

Mouse behaviour. Choose from:

- Mouse: the mouse generates mouse-down, mouse-move, and mouse-up events.
- Ignore: the mouse does nothing.
- Touchscreen: the mouse generates touch-down, touch-move, and touch-up events.

Best mouse options for animal-oriented multiple-monitor rigs. Typically, while it is impossible to prevent the mouse pointer from crossing over a window (reliably) that is not using DirectDraw, you may not want the mouse to appear on monitors that are inside your operant chambers. In that case, tick *Hide mouse cursor* and set the mouse behaviour to *Ignore* — the mouse will not be prevented from entering the portion of the desktop corresponding to this display, but will be *invisible* and *ineffective*.

Best mouse options for iPads with Air Display software. The Air Display software makes an iPad behave like an additional monitor; touching the iPad generates mouse events. For this setting, choose "mouse behaviour = touchscreen" so that the iPad emulates a touchscreen for Whisker tasks (a touchscreen emulating a mouse emulating a touchscreen!). You probably also want "hide mouse cursor" ticked.

Best mouse options for human testing with a mouse. Don't hide the mouse cursor. Choose mouse behaviour to be "mouse" or "touchscreen" depending on what your Whisker client task expects.

Losing the mouse pointer. Hiding the mouse cursor can cause one problem, namely that you may 'lose' your mouse pointer because it is invisible; in that case, you need to move it in the general direction of a monitor that doesn't have this kind of Whisker display on it! (If your

primary monitor is at the top left of the desktop map in Control Panel, for example, and isn't being used for a Whisker display, simply move your mouse left and up and the mouse pointer should come back into view on the primary monitor.)

OLDER VERSIONS (before WhiskerServer v4.2.1): had an "Exclude mouse" option. When "Exclude mouse" was ticked, this was equivalent to "Hide mouse cursor" and "Ignore" mouse behaviour. When it was unticked, this was equivalent to showing the mouse pointer and having it behave like a mouse.

Keep topmost. (Only available if DirectDraw is not being used.) This ensures that the Whisker display is forced on top (in front) of any other windows. It should normally be ticked. However, you are recommended to *avoid* ticking this option for the primary monitor — as it will then be very hard to use any other programs!

Use DirectDraw back buffer (PRECLUDES VIDEO). Select between using the standard Windows graphics routines (this option off), or high performance DirectX routines (this option on). This should generally be enabled, unless: (a) you wish to use video (see Video objects); (b) you experience problems with a piece of hardware: not using DirectDraw may solve some problems. One particular instance of (b) is that DirectDraw mode doesn't work with an iPad running Air Display (May 2012).

Exclusive mode. Puts DirectDraw into exclusive mode, in which Whisker tries to take exclusive control of a whole monitor for that display.

Border. Suppose your touchscreen is smaller than your monitor - the touchscreen is 1024 arbitrary units wide, while the monitor is 1280 arbitrary units wide. You may then want to scale the Whisker display so that the display is only 1024 units wide, so all of it is visible through the touchscreen. To achieve this, you want a display that's 1024/1280 = 0.8 (80%) of the size of the monitor. So you would want a 10% border on either side of the screen. To achieve this, enter 10% in the "border" box. (The default is no border - 0% - in which the whole monitor is used.) The border applies to the left/right and top/bottom borders simultaneously.

Note: the primary monitor may appear twice — once as 'Primary display driver', and once as the video card that is driving that monitor. There is no point enabling both.

When you first run WhiskerServer, the defaults for the primary monitor are *don't use —don't go topmost — don't exclude mouse*. You can then change these settings if you wish. All other monitors default to *use — don't use DirectDraw — go topmost — don't exclude mouse*. This is a safety feature: it is suggested that you use the mouse to check that the display is where you expect and responds correctly, and then exclude the mouse when you have finished testing.

Display settings take effect when you restart the server.

Tips

MULTIPLE MONITOR SYSTEMS CAN BE CONFUSING.

As the owner of a multimonitor system, you may notice all sorts of strange behaviour — for example, when most programs go into a 'full screen' mode (e.g. Microsoft Word), their borders extend slightly beyond the desktop you are familiar with and may be visible on the next monitor! For reasons such as these, Whisker displays that aren't using DirectDraw should have *Keep topmost* enabled.

Even a 'topmost' window cannot entirely prevent anything from being visible in front of it. If you configure Windows so that window outlines are shown when windows are dragged (in Windows 2000, *Start* → *Settings* →

Control Panel → *Display* → *Effects* → *Show window contents while dragging = OFF*), the outlines will be visible over the Whisker display. (Once you release the mouse button, the dragged window will vanish behind the Whisker display.)

Thus, it is possible to move another window 'underneath' a Whisker display. If the Whisker display is enforcing *Keep topmost* behaviour, it may be impossible to retrieve the other window!

It is impossible for Whisker to defend non-DirectDraw windows entirely against 'virtual desktop management' programs that allow you to move windows around. For example, *TriPlus WinSpace 3 Light* is capable of moving Whisker full-screen non-DirectDraw displays around on the desktop, giving ridiculous results.

THE SIMPLEST AND BEST RESULTS SHOULD BE OBTAINED USING DIRECTDRAW EXCLUSIVE MODE.

Note also that Whisker display windows appear in the Windows task bar to confirm that they are being used. This is deliberate.

Tips

IF YOU LOSE CONTROL OF YOUR COMPUTER WHILE TESTING…

It is possible to lose the ability to see where you are working if you mis-configure the display devices. If you enable the primary monitor, you may have to press **Alt-TAB** to get back to your desktop, because the taskbar will not be visible. But **if you enable the primary monitor and set the *Keep Topmost* attribute,** the display will force itself in front of everything on your desktop and keep itself there. It is even more confusing if you also set the *Exclude mouse* attribute, because then you can't see the mouse.

To rescue yourself,

1. Shut down the server.

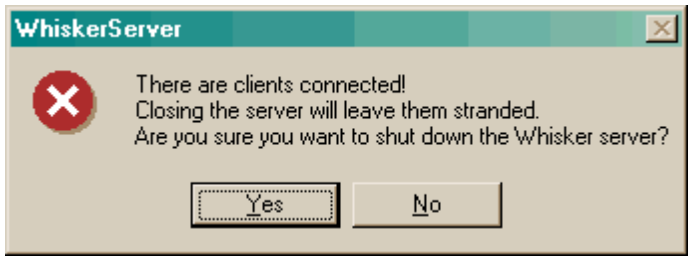
Hold Alt down and **press TAB** until the small, moving box is over the WhiskerServer icon (*not* the full-screen display device, which will have an 'F' in its icon – the WhiskerServer icon has no letters, just the eye-and-whiskers picture). **Release Alt.** This will transfer the focus to the WhiskerServer console, *even though you may not be able to see it.* Your screen will probably look like this until WhiskerServer is shut down:

which is the problem!

Press Escape at least three times to ensure that WhiskerServer is not displaying any menus or dialogue boxes.

Press Alt-F4 to close WhiskerServer. Although you will not be able to see it, WhiskerServer will now pop up this dialogue box:

so press **Enter.** If clients were connected, a second dialogue box will appear (again, you will be unable to see this):



so press **Enter** again. The server will now close, and your desktop will become visible again.

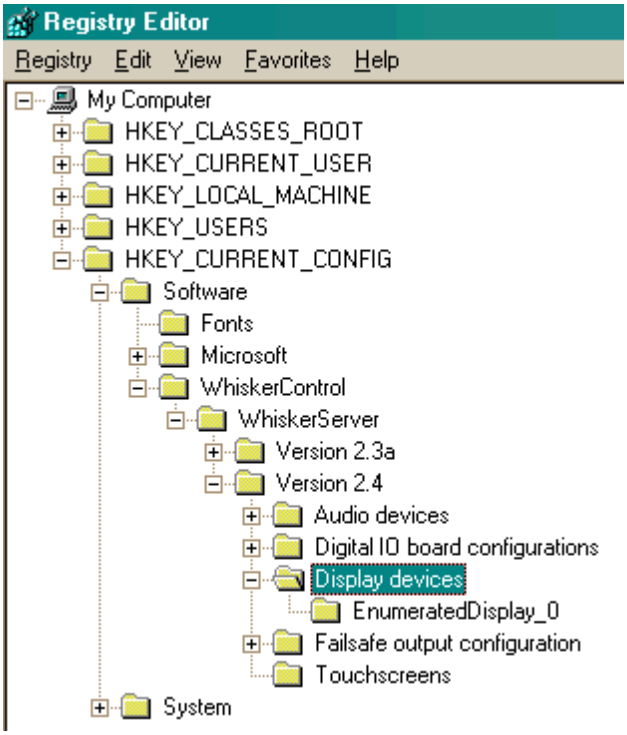
2. Disable use of the primary monitor

To avoid this situation recurring, choose *Start* → *Run* and type in **regedit** ↵. This will run the Registry Editor.

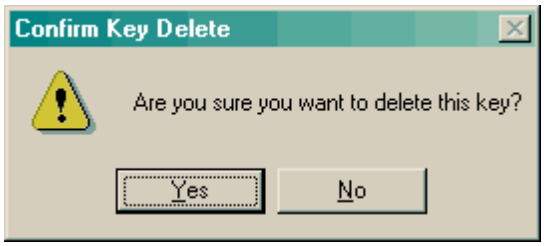
In the left-hand tree view of the Registry Editor, find the following entry:

HKEY_CURRENT_CONFIG\Software\WhiskerControl\WhiskerServer\Version [the most recent]\Display devices

It will look something like this:



If you wish, you can explore the tree further and directly edit the entries for the primary monitor (EnumeratedDisplay_0) to disable it, but the easies thing is to **click on 'Display devices'** in the left-hand tree and **press DEL.** The Registry Editor will ask for confirmation:

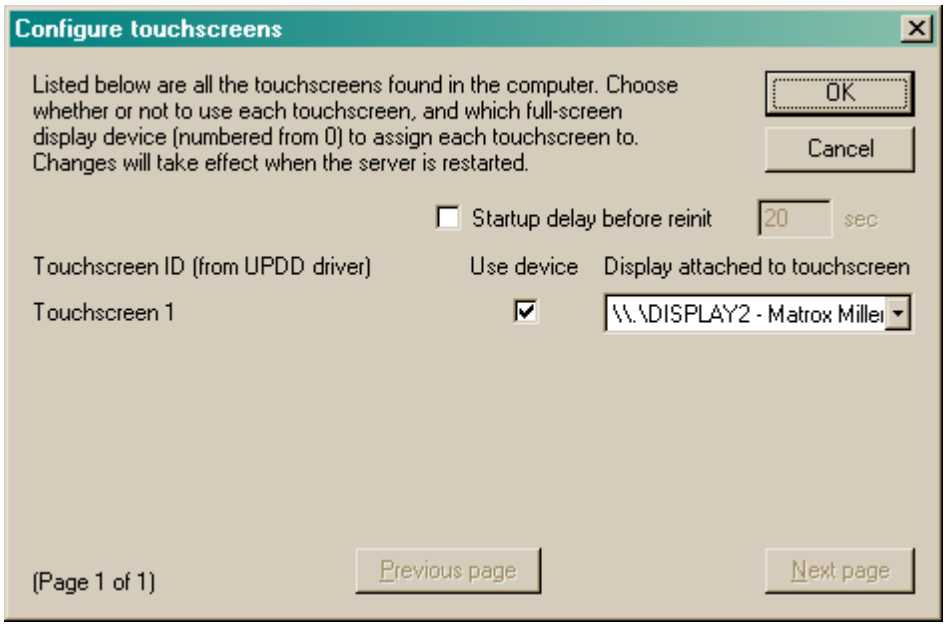


When you next run WhiskerServer, it will return to its default behaviour, in which the primary monitor is disabled. You may have to reconfigure your other monitors.

Why is this situation permitted? Because I think it may be useful to have full-screen control of your primary monitor under some circumstances, e.g. if you are testing an overly-inquisitive subject on a laptop. If nobody ever uses this facility and a consensus of opinion develops that it's a pain to have the primary monitor enabled in 'topmost' mode, I might remove the option. Comments to rudolf@pobox.com.

6.4.4.13 Touchscreens

Configure hardware → Touchscreens



Lists every touchscreen connected to the server. Simply tick **Use device** for each one you want to use, and choose the physical display (monitor) that the touchscreen is attached to.

If touchscreens are powered via the Whisker failsafe relays (*we do not recommend this!*), *and* they have difficulty communicating with the UPDD touchscreen driver upon power-up (as has been reported for some ELO Caroll Touch touchscreens), touchscreens may not communicate with UPDD successfully upon power-up (i.e. upon starting Whisker). From v2.12.5, Whisker asks UPDD to re-initialize communication with its touchscreens when Whisker starts. However, should this prove insufficient because of UPDD-touchscreen communication problems, it is possible to delay the Whisker startup process so that Whisker initializes its failsafe relays (powering up touchscreens connected inappropriately via the failsafe relays), wait a while, and then request re-initialization of the UPDD-touchscreen connection. This option is offered in the dialogue box (**Startup delay before reinit**).

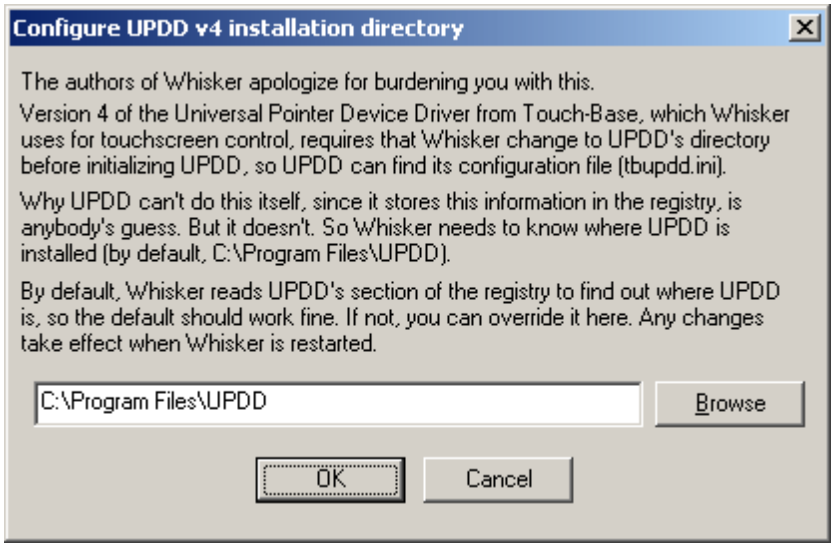
If you are using **UPDD version 4,** see also Set UPDD v4 directory.

6.4.4.14 Set UPDD v4 directory

Configure hardware → Set UPDD v4 directory

The Universal Pointer Device Driver (UPDD) from Touch-Base, version 4, requires that any software using it (such as Whisker) changes to the UPDD installation directory (where tbupdd.ini) is located prior to initializing the UPDD interface. This is curious, since UPDD could and should do this for itself. As it doesn't, Whisker needs to know where UPDD is installed (by default, this is C:\Program Files\UPDD).

By default, Whisker reads UPDD's location from UPDD's section of the Windows registry (HKEY_LOCAL_MACHINE\SOFTWARE\Touch-Base\InstallDir). You can override this behaviour here, though you shouldn't need to.

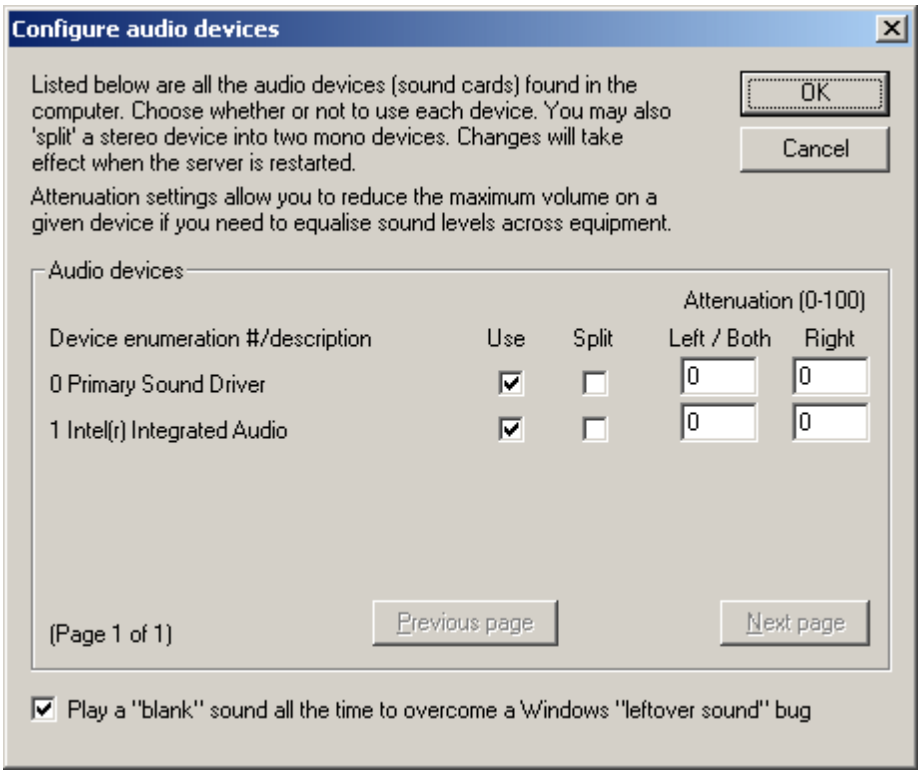


This option is irrelevant for earlier versions of UPDD.

6.4.4.15 Audio devices

Configure hardware → Audio devices

Configures sound output devices. All the sound devices available in the system are listed in the dialogue box (shown below). Some sound cards may appear twice — for example, the computer that provided the screenshot above has one sound card in it, which appears twice (as 'Primary Sound Driver' and as 'Creative Sound Blaster PCI').



Use device. You may enable or disable each sound device. (It is not sensible to enable two devices that are actually the same physical sound card, even though the example above does so!)

Split device. Every device that you are using, you may treat as a single stereo device, or split it into two separate devices — the left and right channels. This doubles the number of available (monophonic) sound devices. **Note** that some poor-quality sound cards exhibit 'bleed', such that sounds intended to come only from the left channel are audible (albeit quietly) on the right channel; these sound cards are probably unsuitable for splitting. **Consider this a "budget" option** that may help if you are very short of physical devices, but which is not guaranteed for high-quality sound; see also the note below regarding distortion.

Logical devices — the kind that Whisker clients and/or the server's device definition file need to deal with — are numbered sequentially from 0 when WhiskerServer starts. So if the first and third physical sound cards are enabled and the first sound card is split, the logical devices will be 0 (first sound card, left channel), 1 (first sound card, right channel), and 2 (third sound card, stereo).

Play a "blank" sound all the time... There is a bug in at least several versions of Windows, in which (1) sound A is played and stopped; (2) sound B is started. The "tail end" of sound A can then be played as sound B starts, which often sounds like a click. This is nothing to do with Whisker, but is also apparent with demo applications from the DirectX 9 SDK (when Whisker isn't even running). A workaround that Whisker offers is to play a "blank" (zero) sound all the time. This fixes that problem. **However,** some sound drivers/cards produce nonzero power when asked to play zero -- and for some applications, that matters. If you have a buggy version of Windows and you care about the clicks, turn this feature on. If you have suboptimal sound drivers/hardware and care about the background low-level hum, turn it off.

These settings take effect when the server is restarted.

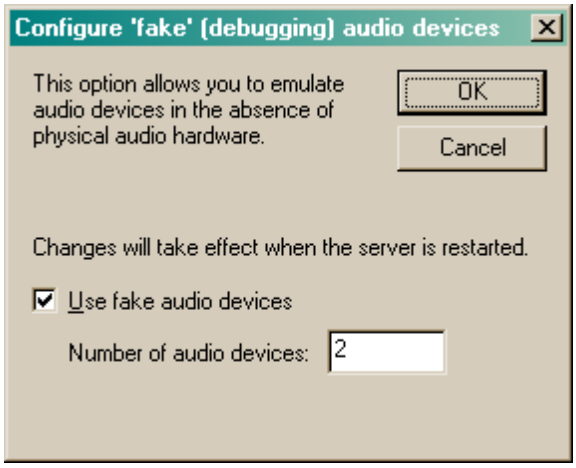
Additional note 1: other sound problems. *Some users have experienced problems when "splitting" sound cards in which frequencies are distorted: for example, a 16 kHz sine wave (generated by Whisker) is emitted with sidebands at 4 kHz, 8 kHz, 12 kHz, 20 kHz, and 24 kHz when played through a "split" sound card, but correctly without them when played on the audio device directly (e.g. Matt Boehm to Rudolf Cardinal, 16 Dec 2016). This may relate to rounding errors in the sound driver's calculation of 3D spatial location (MRFA to RNC, 17 Dec 2016).*

Additional note 2: bugs in audio drivers. *We have also experienced severe and hard-to-track software problems relating to poor sound drivers (see also MonkeyCantab version tracker around 17 Nov 2010). We found that particular SoundBlaster kernel-mode drivers failed to restore the floating-point unit (FPU) state correctly, having used the FPU in kernel mode. This is a dreadful driver bug, which leads to unpredictable corruption of the FPU state, and therefore can affect any software using the FPU (such as MonkeyCantab positioning its stimuli, or Excel doing sums). For example, Excel would give different answers depending on whether Windows Sound Recorder was playing sounds (with no other non-OS software running, and the wrong answers being when the sound card was active). The specific rule that this driver flouted is documented at http://msdn.microsoft.com/en-us/library/ff565388%28VS.85%29.aspx.*

6.4.4.16 Fake audio devices

Configure hardware → Fake audio devices

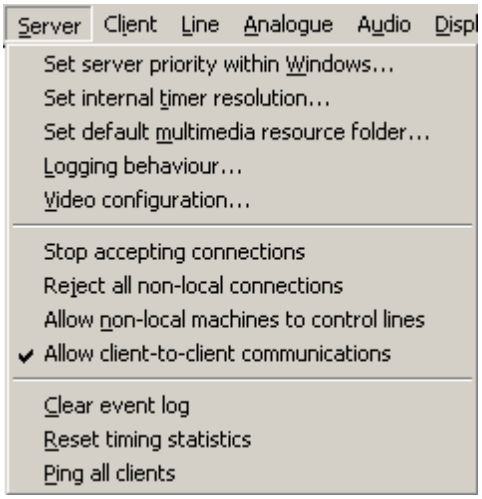
Configures 'fake' audio devices, useful for running tasks that require sound, silently, on a computer with insufficient sound cards.



Fake audio devices make no sound and are not connected to any sound cards in the computer, but behave in other respects like genuine lines.

Fake audio devices are after all genuine audio devices.

6.4.5 Server



Set server priority within Windows. This controls the priority given to WhiskerServer by Windows, and influences the server's performance. This is an advanced option.

Set internal timer resolution. This configures the speed of the timer with which WhiskerServer polls digital I/O hardware. It is an advanced option.

Set default multimedia resource folder. When clients use multimedia files (bitmaps, WAV files, etc.) and do not specify exactly where to find them, the server will try looking in the directory (folder) specified here.

Logging behaviour. Set server logging options.

Video configuration. Configure video options.

Stop accepting connections prevents any more clients from connecting. This option does not affect clients that are already connected. Once selected, the menu item changes to **Start**

communications…, which allows you to restart comms. (When restarting comms, you may specify the TCP port numbers in the process, but I recommend that you do not change the port numbers unless you understand TCP/IP well – client software will not expect the change.)

Reject all non-local connections is an option that prevents any other computer from connecting to the Whisker server. This is the highest security level, but is off by default, because it prevents you from finding out how your subjects are doing by using WhiskerStatus on your office computer. If you are concerned about the prospect of people elsewhere in the world finding out about Whisker, finding out how it works, discovering the IP address of your lab computer and reading the status messages you intended for yourself, you can either enable this facility, or install a firewall on your network to prevent access from outside.

Allow non-local machines to control lines also sets the behaviour of the server with respect to other computers. Assuming you haven't ticked "Reject all non-local connections", the server allows other computers to ask it for status information (e.g. using the WhiskerStatus program), but does not normally allow other computers to control hardware devices; it responds to any such attempt with an error message. If you want to let other computers have access to the hardware, tick this option. Bear in mind that if your computer is connected to the Internet in the absence of any firewalls, any computer in the world could try to influence your devices!

(Note: changing these last two settings does not affect clients *currently* connected, only clients that connect to the server subsequently.)

Allow client-to-client communications. If this setting is ticked *and* individual clients specifically permit it, clients are allowed to send each other messages. This facility may be used, for example, to control (e.g. configure and start) behavioural tasks from a different computer, or for more advanced facilities (e.g. one client making a judgement based on certain input devices and informing another). See the *Programmer's Guide* for full details. Disabling this option prevents any client–client communication via the server.

These last three options are remembered for next time — they are stored in the registry.

Clear event log removes all entries from the server's event log.

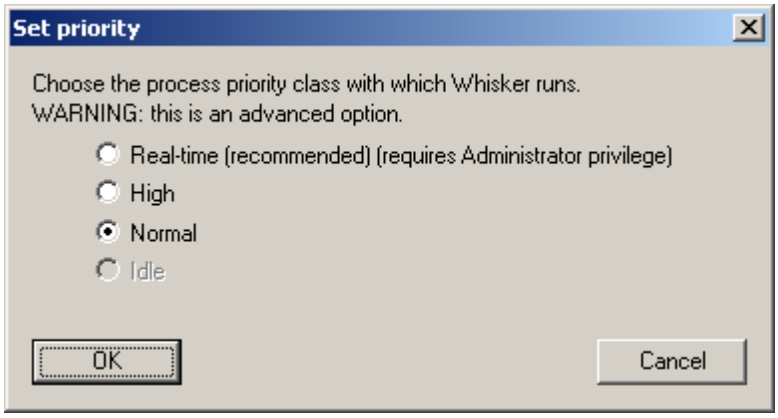
Reset timing statistics resets all the timing measurements displayed in the Server Status view to zero. This is discussed further in Whisker — Performance considerations.

Ping all clients sends a 'Ping' message to all connected clients (see Client Menu).

6.4.5.1 Set server priority within Windows

Server → Set server priority within Windows

Caution: advanced option



With this dialogue box, you can configure the process priority with which WhiskerServer runs. This sets the 'importance' of the server relative to all the other programs running. The default is real-time. These options are as follows:

- **Real-time.** The highest possible priority. This process pre-empts all other processes, including operating system processes performing tasks. 'An application that runs at this level for more than a very brief interval can, for example, cause disk caches not to flush or cause the mouse to be unresponsive.' [Source: Windows Platform SDK documentation.] **This mode requires WhiskerServer to be run with Windows Administrator privileges.**
- **High.** This process pre-empts those running at Normal or Idle levels. Used by applications that must retain rapid responsiveness, re-gardless of the system load. Example: the Windows NT task list.
- **Normal.** The normal priority. Most processes use this.
- **Idle [not allowed].** The process runs only when the system is idle. Example: screen savers.

You may think that it sounds dangerous to run at the real-time priority. In practice, Whisker does only execute for a very brief interval every time it is called by Windows (every 1 ms or so); if Whisker were to occupy the CPU for longer than the interval, it would in fact 'lock up' the computer completely. To avoid this every happening, Whisker 'yields' control for a brief period every time it is called, to ensure that vital system tasks can always be carried out.

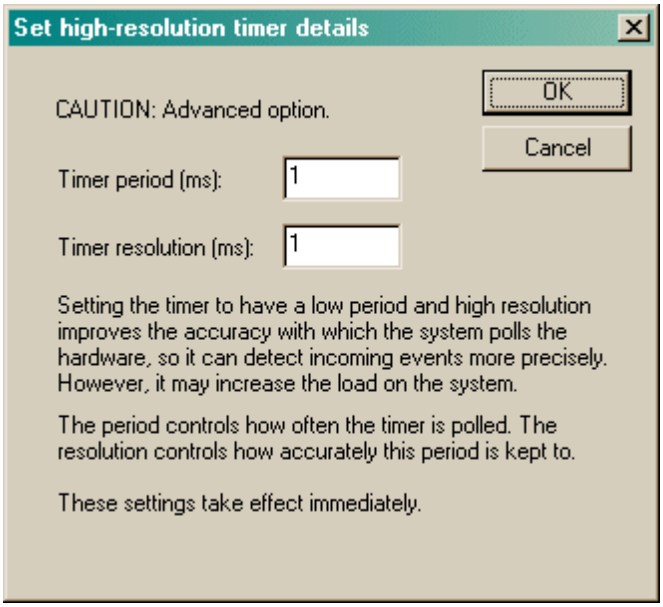
When you alter these settings, the changes take effect immediately and are remembered the next time you start WhiskerServer.

For a detailed discussion, see Performance Considerations.

6.4.5.2 Set internal timer resolution

Server → Set internal timer resolution

Caution: advanced option



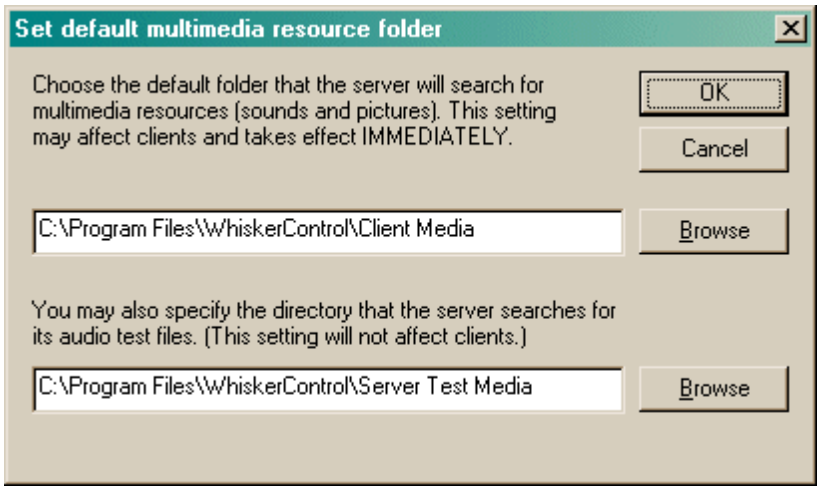
This configures the speed of the timer with which WhiskerServer polls digital I/O hardware.

I suggest you leave the values at the defaults of 1 ms period, 1 ms resolution; these settings give the best performance for WhiskerServer and do not interfere with other software on any of the computers on which Whisker has been tested.

6.4.5.3 Set default multimedia resource folder

Server → Set default multimedia resource folder

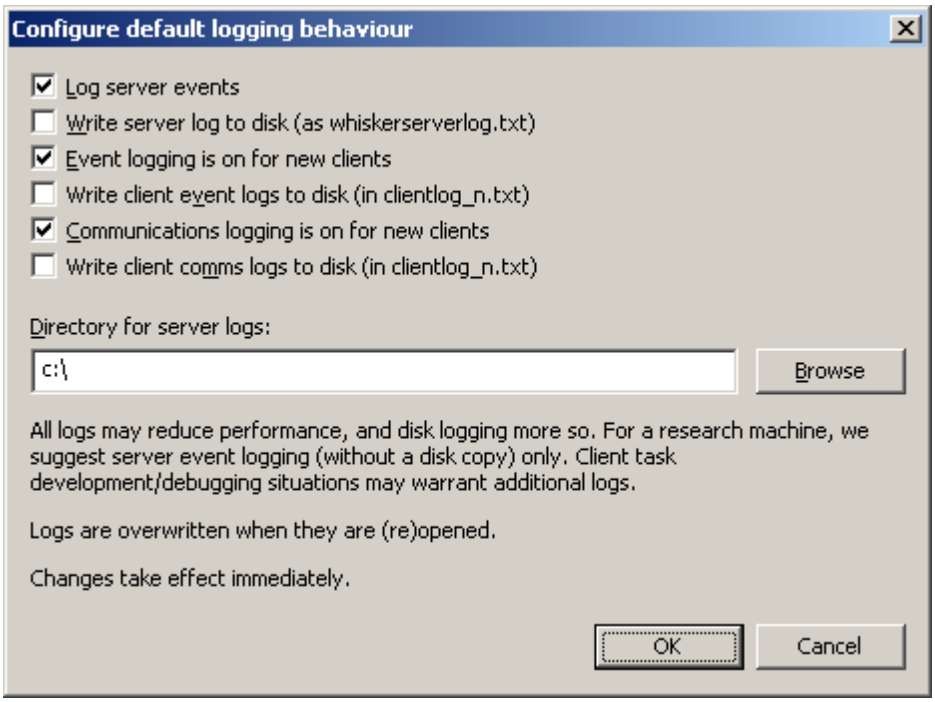
When clients use multimedia files (bitmaps, WAV files, etc.) and do not specify exactly where to find them, the server will try looking in the directory (folder) specified here.



The second folder you can specify, the *server test media directory*, is used to find the sound files used by the server to test audio devices (see Audio Menu).

6.4.5.4 Logging behaviour

Server → Logging behaviour



Log server events? If on (as is the default), the server's event log keeps track of important events.

Write server log to disk? If server event logging is on, and this option is on, a file called **whiskerserverlog.txt** is written (live) to the directory specified; this is a copy of the server's event log. This is principally to aid developers in task debugging situations, and is off by default to improve performance.

Event logging is on for new clients? If true, the client event log is enabled for new clients. (Subsequently, event logging can be turned on and off for each client.) This is principally to aid developers in task debugging situations, and is off by default to improve performance.

Write client event logs to disk? If a client's event log is enabled, and this option is on, the contents of the event log are also written (live) to disk in **clientlog_*n*.txt** (e.g. clientlog_5.txt for client number 5) in the directory specified. This is principally to aid developers in task debugging situations, and is off by default to improve performance.

Communications logging is on for new clients? If true, the client communications log is enabled for new clients. (Subsequently, communications logging can be turned on and off for each client.) This is principally to aid developers in task debugging situations, and is off by default to improve performance. It is a more comprehensive log than the event log.

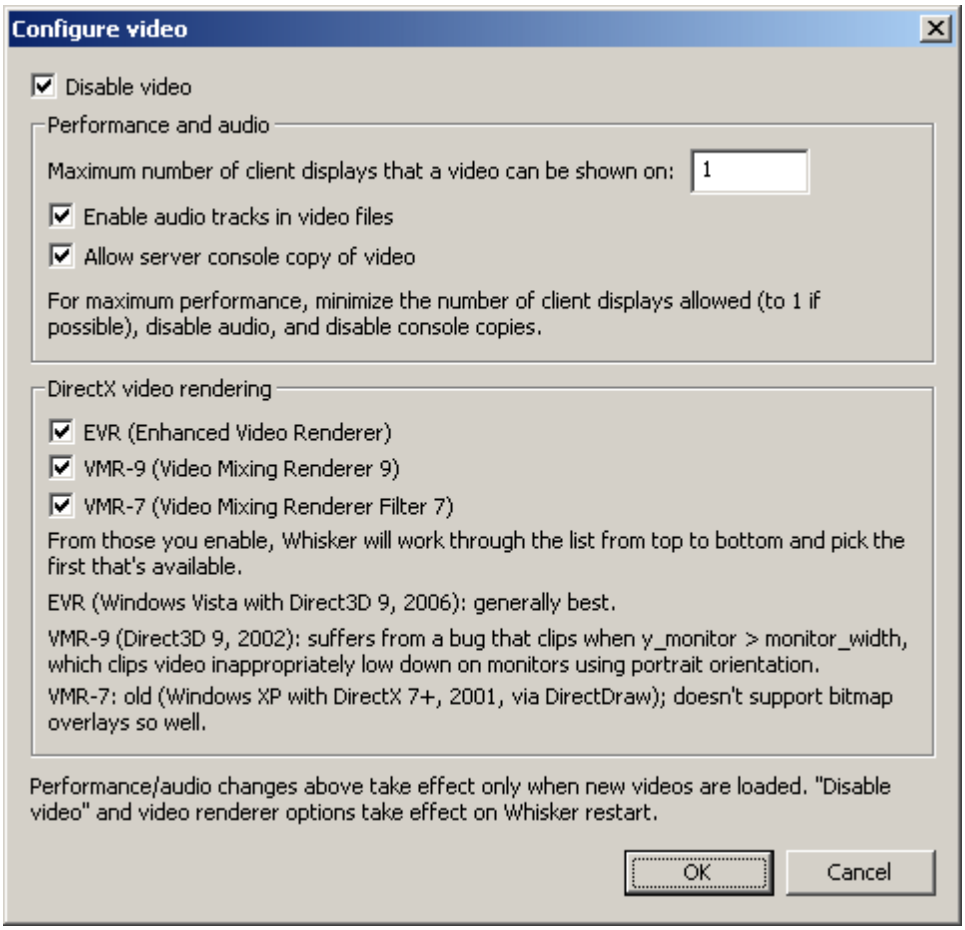
Write client comms logs to disk? If a client's communications log is enabled, and this option is on, the contents of the communications log are also written (live) to disk in **clientlog_*n*.txt** (e.g. clientlog_5.txt for client number 5) in the directory specified. This is principally to aid developers in task debugging situations, and is off by default to improve performance.

Directory for server logs? Choose where these disk logs, if selected, should be stored. The

default is C:\, since that always exists, but if you plan to use these disk logs it would be preferable to make and then choose a different directory (e.g. C:\WHISKERLOGS).

6.4.5.5 Video configuration

Server → Video configuration



Configure settings relevant for playing video clips. See also Video objects.

Disable video. Turns off video entirely.

Performance and audio

Maximum number of client displays that a video can be shown on? There is a tradeoff between performance and the ability to show a video on lots of displays. If you plan to show multiple synchronized copies of the same video on multiple server displays (i.e. yoking using a video stimulus), you may want this number to be >1. For normal use, pick the lowest possible value; the default is 1.

Enable audio tracks in video files? If you (a) enable this, and (b) use DisplaySetAudioDevice correctly, and (c) enable audio in your DisplayAddObject (video) command, then you can get audio playing with your video. There's a performance cost, so disable this if it's not needed.

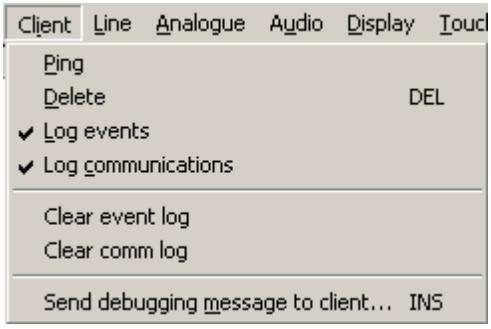
Allow server console copy of video? If enabled, you'll see a live copy of any videos on the

server's console. If disabled, you'll see a placeholder. There's a performance cost to this option, so disable it if it's not needed.

DirectX video rendering

Choose the renderers to use. Of those selected, the first that's available will be used. See the text in the dialogue box for more explanation.

6.4.6 Client



The client menu is only available when a client is selected (either in the tree view on the left, or in the client summary view).

Ping sends a 'Ping' message to the client in an attempt to get it to respond. It is good practice to write clients that do respond to *Ping* messages (see discussion of client/server communications below). Pinging a client that has been forcibly shut down by the operating system (closing its link to the server) may cause the server to notice that it has gone.

Delete enables you to disconnect the currently selected client from the server. It will be removed from the server's list and all communication links to the client will be severed. You would normally use this only as a last resort to get rid of 'dead' (crashed) clients. If you are unsure whether a client is dead, check for network errors in the client status/summary view and see when the client last sent a message to the server. If the client is programmed to respond to 'Ping' events, you may use the *Ping* command to try to provoke a reaction from it.

Log events toggles the recording of significant events relating to the client. A tick by the entry indicates that logging is active.

Log communications toggles the recording of communication messages sent to/from the client. A tick by the entry indicates that logging is active.

Communications and event logging are off by default, to save time and memory. However, the settings you choose are remembered for the next time you run WhiskerServer (they are stored in the registry).

When you turn the logs on, messages will start to be recorded for the client, and any new clients that connect will take the new settings as their default. Therefore, if you want to monitor communications at the very start of a client's session, turn on communications (using any other client) before the client of interest connects.

Clear event log removes all entries from the selected client's event log.

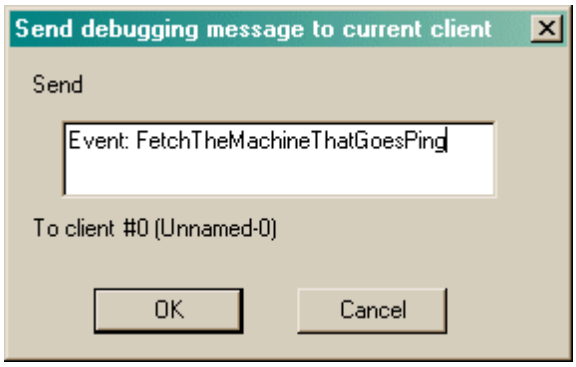
Clear comm log removes all entries from the selected client's communications log.

Send debugging message to client allows you to type a message that is sent to the client immediately.

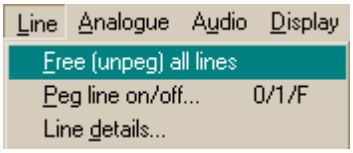
6.4.6.1 Send debugging message to client

Client → Send debugging message to client

This allows you to type a message that is sent to the client immediately. Prefix the message with "Event:" if you want to mimic an event message. Note that clients are allowed to assume a rigid pattern of messages from the server – they can be case-sensitive, if they want, as the server normally guarantees them a predictable message – so it's your job to get this right if you mimic the server yourself.



6.4.7 Line



Free (unpeg) all lines allows you to remove any overrides (or "pegs") that you have manually applied (see below).

Some options on the line menu are only available when a line view is selected:

Peg line on/off is a debugging feature that allows you to force any line **ON** or **OFF,** or to **Free** it up again.

Line details brings up a dialogue box giving you detailed information on a single line.

6.4.7.1 Free (unpeg) all lines

Line → Free (unpeg) all lines

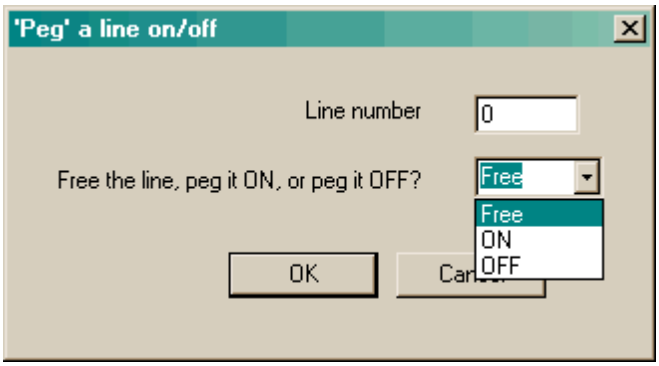
This option clears any pegs you have set (see also Line Details).

6.4.7.2 Peg line on/off

Line → Peg line on/off

This is a debugging feature that allows you to force any line **ON** or **OFF,** or to **Free** it up again. The 'pegged' state overrides the true state (of input lines) or the commanded state (of output lines).

In other words, pegging input lines allows you to manipulate what your client software sees (including the generation of events), independently of the state of the device responsible for the input; pegging output lines allows you to manipulate the devices without the client sending any instructions. This facility is therefore useful for testing client software and checking your apparatus is working. It is *not* intended to be used during normal operation.



When you free a line, input lines return to their 'real-world' state (the state of the device that is connected). Output lines return to the state that the client expects them to be in.

Pegging may also be done from the Line Details dialogue box, or from the Digital Line Status view.

Pegged lines send warning messages if events are triggered by them. That is, if setting a line to be ON, OFF, or FREE changes its effective state, and an event is triggered, that event is sent and followed immediately by a warning, like this:

```
Event: AAA
Warning: Event generated from a pegged line (line turned on, event AAA)

Event: BBB
Warning: Event generated from a pegged line (line turned off, event BBB)

Event: AAA
Warning: Event generated from a line being freed (line turned on, event AAA)

Event: BBB
Warning: Event generated from a line being freed (line turned off, event BBB)
```

Warning are also triggered if the client tries to set the state of a pegged output:

```
Attempting to set the state of a pegged output, won't do anything
```

Note also that lines remain pegged when a client releases them. *The Pegging Rule:* if you pegged it, it's your job to free it again.

You can free all pegged lines simultaneously with the *Free (unpeg) all lines* command.

6.4.7.3 Line details

Line → Line details

Brings up a dialogue box giving you detailed information on a single line. The dialogue box is 'modeless', meaning that it will stay there so you can use the server's other facilities and watch the line's activity at the same time.

To select a line, type its number into the box and click *Update*. Use *Next* and *Previous* to cycle through the available lines.

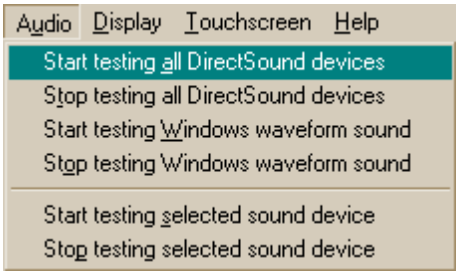
The **light** is black if the line is off, yellow if it is an input that is on, and red if it is an output that is on.

You may set the line's **peg state** by clicking the radio buttons (*Free… Forced ON… Forced OFF…*).

The **number of 'on' transitions** indicates the number of times that the line has gone from OFF to ON (whether spontaneously [inputs], as a result of a client command [outputs], or as a result of pegging its state manually). Click one of the *Reset statistics…* buttons to reset this number. This facility is intended to help you spot hardware faults — if many thousands of transitions occur in a short space of time, for example, a hardware fault should be suspected.

The other information given in the box is exactly the same as that given in the line status view (described earlier).

6.4.8 Audio



The Audio menu enables you to test audio output devices.

Start testing all DirectSound devices causes sounds to be played on every sound device that is not claimed by a client. ***See below for the filenames used to look for the test WAV files.*** The sound(s) are played on a continuous loop. Multiple tests can be started, which tends to lead to a cacophony (and possibly severe bandwidth overload - see below), but there you go; that's why it's a test facility!

Stop testing all DirectSound devices cancels any ongoing DirectSound tests.

Start testing Windows waveform sound tests a facility *not* used by WhiskerServer. It sends a sound to the Windows primary sound device; this enables you to establish which sound device is vulnerable to interference from other applications. (It is the primary sound device that receives all the beeps and bongs that your wordprocessor generates when you press the wrong key.) The sound is played on a continuous loop.

Stop testing Windows waveform sound cancels any ongoing Windows waveform sound test.

Start testing selected sound device. If you have clicked on a sound device, this option tests that device on its own. (Devices that are claimed by a client are not tested.)

Stop testing selected sound device cancels any tests on the selected device.

What sounds are played? WhiskerServer looks in the server's test media directory for files named:

`testsound.wav`	Used to test Windows waveform sound.
`testdevice0.wav`	Used to test sound device 0.
`testdevice1.wav`	Used to test sound device 1.
`testdevice2.wav`	Used to test sound device 2… and so on.
`...`	

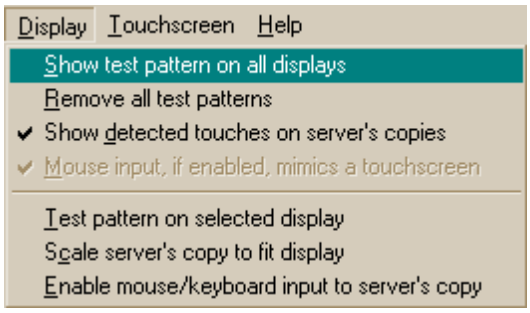
You can therefore choose your own test sounds. In the test release of Whisker v2.0, two sets of sounds are supplied on the CD:

1. a lightweight set, each a few seconds long in 8-bit 22-kHz mono or less, with the introduction to a familiar TV show's theme music as the Windows waveform test, and voices counting from 0 to 9 as the device test files;
2. a heavyweight set, each about a minute long in 16-bit 44-kHz stereo, with an excerpt from Sibelius's violin concerto in D minor as the Windows waveform test and symphonies numbered from 0 to 9 (Bruckner 0, Beethoven 1–9) as the device test files. The second set of files are of near-perfect sound quality, so these sounds test the quality of your sound

system as well as the capacity of the server to deliver 176 kb s^{-1} audio to your sound device without interruption. If you have multiple (or split) sound devices, you can stress-test the server further; for example, by testing ten sound cards simultaneously you require the server to provide 1.76 Mb s^{-1}. It might sound awful, though, if the speakers are near each other!

If Whisker can't find a suitably-named file, it tries again in the default multimedia resource directory; if this fails, an error message is written to the server's event log.

6.4.9 Display



Show test pattern on all displays. Displays a test pattern on all display devices that are not in use by a client.

Remove all test patterns. Does what it says on the tin.

Show detected touches. If this is selected, then any touches detected on the display will be shown (using a gunsight picture showing the last touch event). This allows the user to monitor the behaviour of an animal & operation of a touchscreen from the Server's console.

Mouse input mimics a touchscreen

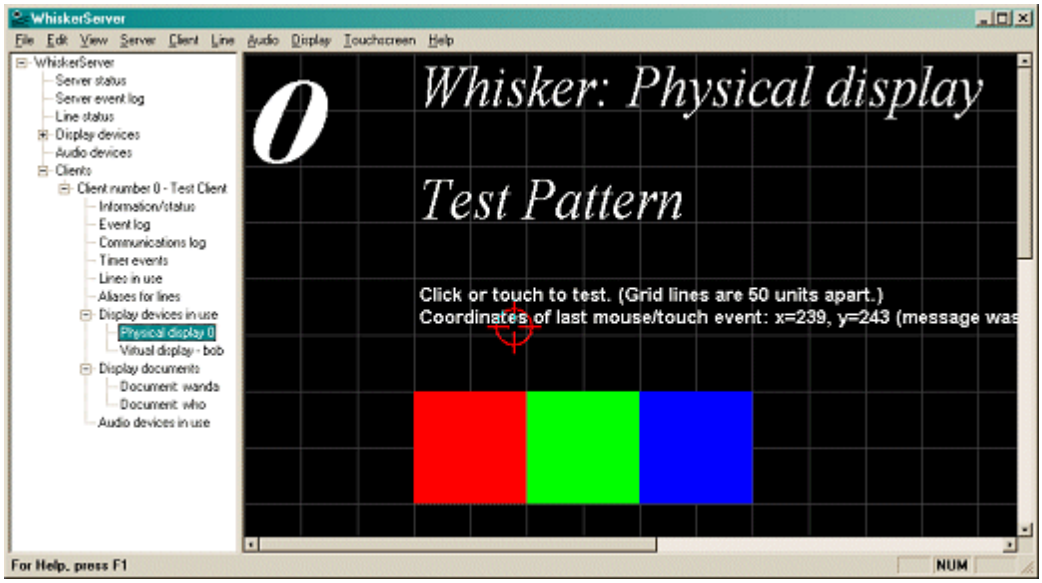
If this is selected, then if mouse input to the server's copy is enabled (see below), this input will generate Touch events (rather than Mouse events) on the selected displayed document. These touches will be displayed on the Server's view if the option to display touches is selected.

Test pattern on selected display. Turns a test pattern on or off for the selected display (if it is not claimed by a client).

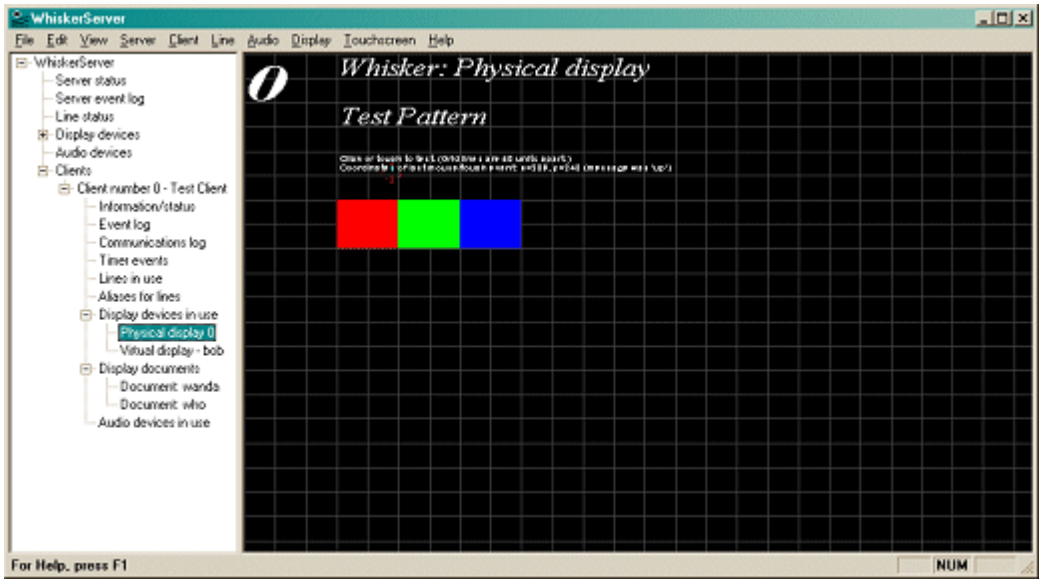
Test patterns identify each display with a large number (helpful for multimonitor computers, in conjunction with the Windows Control Panel). They display a 50-pixel rectangular grid so you can assess the display's size, resolution and alignment. They display boxes of pure red, green, and blue to check that colours are being displayed correctly. They also respond to mouse clicks (and, if you have touchscreens fitted to the display, to touches).

Scale the server's copy to fit [the true] display.

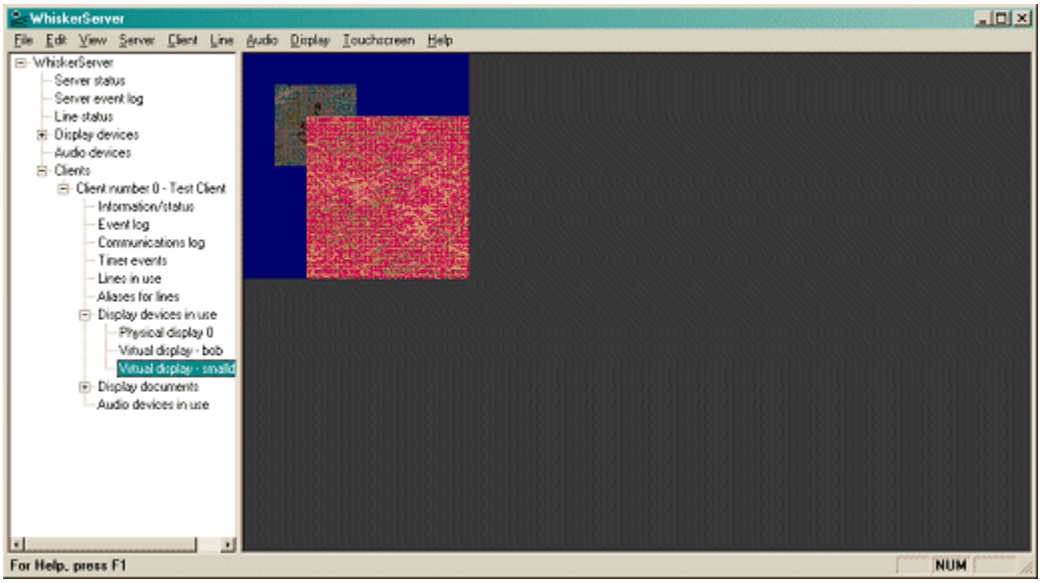
Displays vary in size: physical displays may be on different monitors (640 × 480, 800 × 600, 1024 × 768…) and virtual displays may be any size. By default, the server's copy is shown at 100% size (pixel for pixel) as the display itself. However, you may want to see the whole display at once, in which case you can turn scaling on. Examples are shown below:



A server console view of a large (1280 × 1024) monitor displaying a test pattern.



The same view, but with 'Scale server's copy…' switched on. The scroll bars have gone and the display has been shrunk to fit the server window. This allows you to see everything that is happening on a display device.

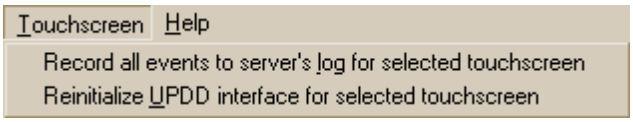


In this case, a virtual display is smaller than the server console window and scaling is off. The grey area at the edge is the area beyond that of the virtual display. Were 'Scale server's copy…' to be switched on, the picture would be enlarged to fit the server's window.

Enable mouse/keyboard input to the server's copy of the display.

Although the display itself normally responds to touch events, and to mouse events if it is a virtual display or the mouse has been enabled on a physical display, the *server's copy* that you see within the WhiskerServer console does not normally respond — the server's copy is principally for you to see what's going on, not to interfere. However, you can turn on mouse input to make **debugging** simple. With mouse input enabled, you can test your new touchscreen task without grubbing inside the operant chamber.

6.4.10 Touchscreen



Record all events to server's log for selected touchscreen. To select a touchscreen, click on it in the Touchscreens view. This option, when ticked, records all touchscreen events for the selected touchscreen in the server's event log.

Reinitialize UPDD interface for selected touchscreen. To select a touchscreen, click on it in the Touchscreens view. If this command is clicked, the UPDD interface for this touchscreen is asked to re-initialize itself. Try this command if a touchscreen appears to be malfunctioning.

6.4.11 Help

Help topics brings up this help system.

About WhiskerServer displays version and copyright information for Whisker, like this:

6.5 Keyboard shortcuts

Press **DELETE** to remove a client forcibly. This key works when the desired client is selected (either in the left-hand view or in the client summary list).

Press **INSERT** to send a debugging message to a client. (You can press INSERT in a few more places than you can press DELETE, because it's a lot safer!)

Press **1** to peg a line ON, when a line is selected.

Press **0** to peg a line OFF.

Press **F** to free up a line.

6.6 Use of the registry by WhiskerServer

WhiskerServer stores information in two places in the registry, that mysterious database at the heart of Windows. The first place is for storing information specific to each user of the system. That is in

\HKEY_CURRENT_USER\Software\WhiskerControl\WhiskerServer

This is used for window size/position, default settings for client communication/event logging and other options (e.g. allowing nonlocal clients to control devices), the server media path, the device definition filename, etc.

The other place is for storing information that applies to all users of a computer, because it relates to the hardware (digital I/O boards, display devices, audio devices, fake I/O lines, etc.). Such configuration information is kept in

\HKEY_CURRENT_CONFIG\Software\WhiskerControl\WhiskerServer

6.7 Performance considerations for Whisker

The problem

Because Whisker runs in a multitasking environment, it is unavoidable that other applications that consume processor time can slow down Whisker.
Nevertheless, we find the ability to run other applications incredibly useful, and on the hardware we have used, it does not pose a performance problem. It is up to you whether you run other

applications at the same time, and Whisker does its best to provide good performance and to help you make this decision by providing performance information.

The way Whisker works

Essentially, the server asks Windows to send it a message every millisecond, or as close to it as Windows can manage. When this message arrives, Whisker does two things. Firstly, it checks the state of the digital input lines, and if any lines have changed state since the last time Whisker looked, the server flags the change; if any clients are using those lines and have asked to be notified of a change in their state, it sends the appropriate message to the clients. Secondly, Whisker runs through all the timers being used by the clients; if any have timed out, it sends a message to the client.

If the computer is slow...

From Whisker's point of view, the computer can be slow either because it is a genuinely slow computer, or because it's doing lots of other processor-intensive things at the same time. Either way, there are two potential consequences.

1. The system's 'reaction times' may deteriorate.

For example, suppose a client is running that switches on a light whenever a rat presses a lever. The client will have requested to be informed whenever that lever is depressed. You could think of the system's reaction time as the time it takes for Whisker to send that message to the client, for the client to receive it, process it, and send a command back to the server to switch on the light, and for the server to receive that command and act on it. If the computer is slowed, this sequence might take longer. (If you're living dangerously and the client and server are on two different computers, then you're relying on the performance of both computers, plus the network that links them!)

Monitoring round-trip performance

Frankly, I don't expect this to be a problem. In an attempt to assess it, I added the `TestNetLatency` command (see the *Programmer's Guide*). If you run WhiskerTestClient, connect to the server, and type in `TestNetLatency`, the following sequence occurs:

client sends `TestNetLatency` to server
server receives `TestNetLatency`, notes the time (Time 1) and responds with `Ping`
client receives the `Ping` and responds with `PingAcknowledged`
server receives `PingAcknowledged`, notes the time (Time 2)
server sends the difference between Time 1 and Time 2 in an `Info` message

The upshot is that you have timed a complete round trip, including processing times, and you will receive a message like this:

```
Info: Network latency is 0 ms
```

On fast computers (e.g. Pentium-III/750, Windows 2000), the network latency is 0–2 ms; on one slower computer I tried (Pentium-II/233, Windows 2000) it reached 3–7 ms. I've tried to load our machines quite heavily, and have not managed to get these latencies much higher; basically, TCP/IP communication within a single computer is fast.

A delay of 7 ms might be unacceptable in an EEG experiment, but in simple behavioural control I don't think it's a problem. For a start, it is significantly shorter than the time it takes for a filament

bulb lamp to reach a reasonable proportion of its final brightness, and certainly an incredibly brief time compared to the times involved in activating lever retracting devices or infusion pumps. It is also extremely brief compared to reaction times in typical tasks. Individual neuronal action potentials last 1 ms; large myelinated neurons conduct at 70–90 $m \cdot s^{-1}$; a typical chemical synapse takes 1 ms to conduct; peak muscle contraction takes 15 ms to develop in the fastest muscles in the mammalian body (the extraocular muscles) and 30–100 ms for most skeletal muscles; similarly, the somatosensory foot–to–cortex nerve conduction time in a human is of the order of 25 ms, and conscious awareness of stimulus perception requires of the order of 100 ms.

However, repeated measurement of the same task allows much finer resolution to be reached; thus, reaction time effects of 10–15 ms may be reliably measured. For such experiments, I would suggest that you use a fast computer and follow the tips given below for optimal performance. The most accurate technique for measuring reaction times is to use the time-stamping feature of the server, removing inaccuracy due to network latencies; see TimeStamps in the *Programmer's Guide* for a worked example.

2. The system may miss events.

In the situations discussed above, the system might respond to events slowly, but it would still notice every single one. The potential to miss events completely is a much more important problem. It might arise like this:

detector device is initially OFF
poll #1: Whisker scans the hardware, finds the device off
device goes ON
device goes OFF
poll #2: Whisker scans the hardware, finds the device off

In this situation, the server would never notice the change. Is this a real problem? Well, if the time between successive polls is 1 ms, probably not. There are no actions a rat can make that are completely over in less than 1 ms! If you have a detector device that genuinely does generate pulses in the sub-millisecond range, there is a real problem. We don't.

However, if the inter-poll time is considerably prolonged, a problem might arise. Therefore, Whisker provides facilities to check that the inter-poll time is acceptable.

Monitoring inter-poll times

Part of the server status view is dedicated to performance monitoring. Here's a snapshot of this view:

```
Worst inter-poll interval so far (ms): 2
This display is scheduled to be updated every 1000 ms
Worst inter-poll interval since last update (ms): 2
Since last update, have had 1004 polls and 3 yields
Of those polls, 100.0% were <=10ms, 0.0% were 11-20 ms, 0.0% were >20ms
Longest poll since last update took 96 microseconds
On the high-performance CPU timer, last poll took 186 ticks and last interpoll took 3397 ticks
High-performance CPU timer is running at 3579545 Hz
Server process priority: Real-time
```

Every second, this display is updated (together with several other clock-related server displays).

This display shows that since the display was last updated, there have been 1100 polls, so the average inter-poll time was 1 ms. All well and good. The worst single inter-poll time in the last second was 2 ms; some indication of the distribution of inter-poll times is also given on the last line of the display.

In this case, the longest inter-poll time since the server started running was also 2 ms. If this number were higher, the long inter-poll time might have occurred when the server first started (this sometimes occurs and is nothing to worry about, because no clients are connected then). When the first client connects, and the server is not in real-time mode, there is sometimes a brief (e.g. 30 ms) pause. But it's just possible (or if you're not running the server in real-time mode, fairly likely) that the long inter-poll time might have occurred because you just ran some very processor-intensive program.

By using the *Server → Reset timing statistics* command, you can assess the impact of running a given program. When you have no clients running (or none that matter!), reset the statistics, then run your program and see how bad the inter-poll time gets. If it exceeds your personal threshold for worrying, don't run that program when you have an experiment going.

Bear in mind that you only have a problem if you run a processor-intensive program *and it hogs the processor at the same instant that your subject makes a response*. The server display gives you the worst-case scenario.

3. Timer accuracy

Timer accuracy is *not* affected by long inter-poll times occurring while the timer is running (though potentially affected by a long inter-poll time occurring just at the moment when the timer expires). All Whisker does on every poll is to calculate the absolute time that has elapsed since the timer was created, and if this time equals or exceeds the programmed timer duration, it sends the timer message.

Technical notes: Repetitive timers

Suppose a timer is set up to fire repetitively, once every 100 ms. What could cause this to be inaccurate?

Transmission delay. If a system delay slows message transmission down by 10 ms, one message will arrive late, and the subsequent message will 'catch up' and be on the originally scheduled time.

A late clock tick. If the server isn't polled by Windows when the timer elapses, the message will be sent late. In this situation, the server could do two things. (1) If it schedules the next occurrence 100 ms later, the client will receive one late message, and the subsequent messages will be phase-shifted and all of them will be late from the point of view of the original command, so the whole sequence will finish late. (2) Alternatively, the server could schedule the next occurrence so it is 'on time', in which case the client will see one late message and one that 'catches up', just as with a transmission delay. The overall sequence will be of the desired length.

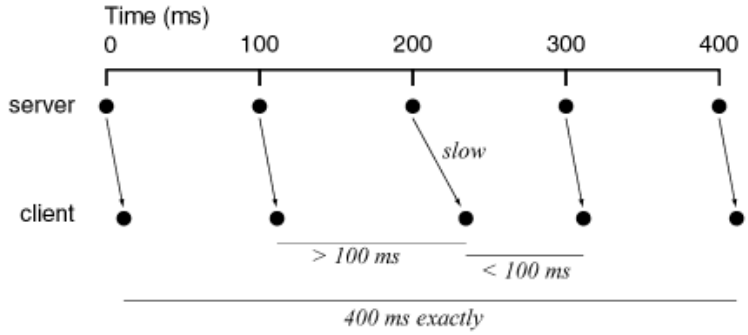
Whisker adopts strategy (2).

These scenarios are illustrated below:

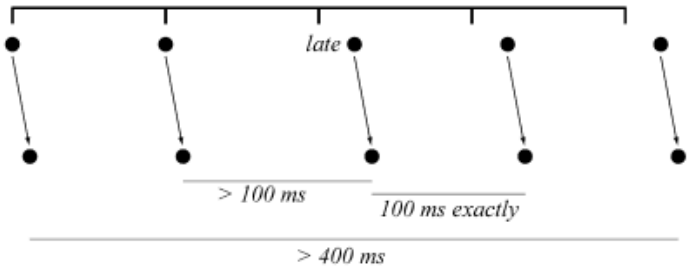
Technical schematic

Types of timer inaccuracy that are possible, illustrated with a repetitive timer that fires once every 100 ms.

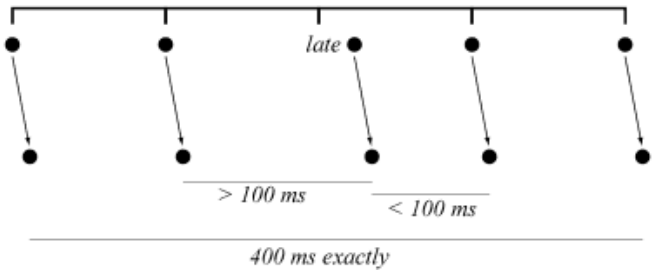
TRANSMISSION DELAY



UNCOMPENSATED CLOCK ERROR. This should not happen in Whisker.



LATE CLOCK TICK, COMPENSATED FOR



Ways that Whisker tries to improve performance

Unless you're interested in the technical aspects of Whisker, skip this section.

High process priority

Under Windows NT, it is possible to set the *priority* of a running program. WhiskerServer, by default, runs as a *real-time* (very high priority) process.

If Whisker consumed significant CPU time, running it as a high-priority process would detract from the CPU time available to Whisker's clients. However, Whisker does *not* use significant processor time for its critical functions — it makes frequent, brief checks of the hardware.

Multithreading

WhiskerServer is effectively composed of several mini-programs or *threads* running simultaneously. A consequence of this is that delays in communication with one client (or in updating the graphical console displays) do not affect any other clients, or the regular polling of hardware devices. Multithreading is discussed in more detail below.

But remember, it's not just about the server…

Although the *server's* speed is much more critical than the clients', as clients do not have to poll the hardware, client speed will be affected by other programs. Programs that consume significant CPU resources (e.g. Microsoft Access 97; Adobe Illustrator 8.0) may slow your clients down if run at critical moments.

Suggestions for optimal performance

All common sense, really; I suggest you view the ability to use the testing computer for other tasks as a luxury and judge for yourself whether it would cause problems for your task.

- If you have really time-critical experiments running, don't use the computer for anything else at the same time.
- Know what programs slow the system down and avoid them when you're testing. You may find that no program ever slows Whisker enough, because Whisker works hard to receive frequent attention from Windows.

Don't push it. If you buy a slow computer, disable Whisker's high-performance features, set the computer up as a web server, dedicate 25% of its CPU time to scanning for extraterrestrial intelligence, and defragment your disk in the background while checking for viruses, running a complicated screensaver and redrawing your plans for a nuclear power station, your system may find itself a little overloaded.

Implement some general tips for high-performance computing.

Technical notes

Process and thread priorities

Windows allows processes (effectively, programs) to run in four different priority classes: *idle*, *normal*, *high*, and *real-time*, in ascending order. Within each process, Windows NT allows you to specify the *thread* priority. The process priority is designed to reflect the importance of the application in the system, and the thread priority to reflect the importance of different threads within that application. Windows NT itself sometimes alters the thread priority slightly – for example, to give a boost to threads that are receiving user input.

Whisker sets the process priority class, but does not alter thread priorities within the class (leaving them at the default, 'normal' priority). These settings can be configured on a per-user basis. They are stored in the registry within *\\HKEY_CURRENT_USER\Software\WhiskerControl\WhiskerServer*.

Hardware interrupts

Whisker uses polling to respond to two classes of event: changes in the state of the digital input lines, and timers. If the digital I/O cards were capable of generating interrupts, then a different approach could be taken with the input lines: instead of checking them regularly, Whisker could sit back and wait to be informed via a high-priority

interrupt signal when one changes. Of course, timers would still have to be serviced by regular polling, and I don't know whether a whole flood of hardware interrupts would have adverse performance consequences, but it would be an additional mechanism of gaining performance.

Extremely technical notes

Threads used by WhiskerServer

Whisker is multithreaded. Multithreading is a complex topic. These notes are here as a reminder to me and as something that may interest you – nothing more.

WhiskerServer is written in C++, using Microsoft Function Classes (MFC) and the MCL library (Cohen A & Woodring M, 1998, *Win32 Multithreaded Programming*, O'Reilly), with permission. It uses the following threads:

- A user-interface (UI) thread, which (1) is the primary owner of all MFC objects, including the main server, client, and socket classes; (2) deals with all user input, including mouse/keyboard input; (3) owns and draws all visible windows.
- Client communication threads (one thread per client), handling the TCP sockets used to communicate with the client. All communication from the client arrives via this thread. (Under Windows, TCP sockets are attached to an invisible window, which receive WM_SOCKET_NOTIFY messages that are passed through to the MFC socket classes.) If this thread needs something drawn to a window (e.g. if the client requested that a bitmap be displayed), it does not do it directly, but sends a message to the UI thread requesting that the window be redrawn at the next available opportunity.
- A high-performance hardware polling thread. A multimedia timer ('MM timer') is set up to fire every 1 ms, using its own thread. When it does so, the digital I/O boards and timers are polled through this thread, and if communication with the client is necessary, this thread will send messages to the client's primary socket.

Insuring against starvation

A design such as this is vulnerable to 'starvation locking' if the threads are of high enough priority. For example, if the polling thread took 1ms or more to execute, it would 'starve' all other threads of CPU time (the OS may allow other high-prioirty threads within Whisker to pre-empt, but essentially Whisker would consume all of the CPU resources). This could be monitored by communicating with a low-priority thread, and checking that it was being serviced occasionally. However, under Win32 all threads within a process (application) must **share** their priority group, so even if the lowest priority thread in Whisker was being serviced, other application's threads may not be. The solution we have come up with is for the polling thread to pause for 1ms as soon as it is called. This means that the polling thread on a normally running machine will still execute every ms; however a machine which was slowed for some reason outside of Whisker the polling will occur 'as quickly as possible without starving other threads'.

The threads access the same data, and as two threads must not access the same data *simultaneously*, 'thread safety' is ensured by locking all data against such access. MFC classes do not expect to be addressed by multiple threads, requiring particular precautions.

The server's process priority setting applies, as the name suggests, to the *process* – all the threads are in the same process, and all receive the priority boost from this setting. The default (real-time) setting is recommended.

Limits on performance – 1 – multimedia display update processing is shared between clients. Threads execute simultaneously. Real-time priority threads are rarely slowed down by other programs running under Windows (it's possible, but it's very unlikely). Therefore, it should be apparent from this list that nothing will interfere with client–server communication or hardware polling. However, one task executed by the UI thread will pause execution of another task also executed by the UI thread. This primarily affects multimedia displays, as they are drawn by the UI thread.

For example, display-document access won't affect other clients, but *display creation* may temporarily pause another client that's *also* trying to create a display / destroy a display / show a document on a display (and vice versa for any of these functions) – because both tasks use the UI thread. *Display redrawing* may slow down redrawing of other displays (though it won't affect client communication speed at all).

I estimate these delays as a few milliseconds or less (most such delays will be sub-millisecond). They will also occur rarely. I doubt that they will cause problems.

The time taken to load bitmaps etc. from disk does *not* contribute to these delays, as that is not part of the redrawing process (the object is loaded from disk by the client thread and only *drawn* by the UI thread). If disk access delays slow displays noticeably, simply create a document in advance and load the bitmaps into it, then switch that document into the display window (which is a fast operation).

In principle, slow activities on the WhiskerServer console can affect and be affected by the speed of displays. Such delays are also incredibly small. WhiskerServer is even invulnerable to a test that pauses execution of many Windows programs – a situation generated by (1) disabling the Windows option to 'show window contents while dragging' (under Windows 2000, this is in *Control Panel* → *Display* → *Effects*); (2) dragging the WhiskerServer main window; (3) holding the mouse still with the button still down. Even this combination of events, which prevents Windows messages getting through to many windows, does not affect WhiskerServer.

If this limit were significant, it would be possible to create further threads (one client, one display thread). However, due to MFC restrictions, this would create the requirement to make temporary duplicates of the display data (specifically, because view windows on an MFC document must all be in the same thread, and the display documents used by Whisker may be viewed from the console – the UI thread – as well as in the window intended to be seen by the test subject) and this would incur a performance hit and use memory. As it's not at all clear that there's a problem with display speed at the moment, I haven't gone down this road. WhiskerServer is fast.

Limits on performance – 2 – network subsystem. If you run your tasks across a network, you are subject to the network's performance. This is not advised – the network facilities of Whisker are designed for monitoring task status (it doesn't matter if the monitor program run slowly), not for running behavioural tasks themselves. Keep the task and the server on the same computer; this uses the internal TCP stack and is fast.

Limits on performance – 3 – the impact of other programs on Whisker and its clients. As discussed above, the high priority with which WhiskerServer runs means that it is unlikely to be significantly affected by anything else happening on your computer (this includes disk accesses, intense computation, etc.). It is very rare for Windows programs to use the 'real-time' priority, because very few Windows programs need real-time performance. Consequently, few programs (including most internal Windows functions) use a priority high enough to interfere with WhiskerServer.

However, this may not be true of behavioural clients. Their performance is usually less critical, because they don't have to poll the hardware – they just wait for the server to inform them of a hardware event. If task performance is critical, you must consider whether you want to run other programs (like your database program) that may slow down your task.

As part of this topic, consider implementing my general performance suggestions.

Limits on performance – 4 – CPU speed and memory. It goes without saying that faster computers are more capable of real-time performance. More memory improves performance by reducing the chance that any program gets swapped out to virtual memory (i.e. hard disk).

6.8 Tips for a fast computing experience

For performance considerations relating specifically to Whisker, see Performance considerations for Whisker. These tips are not specific to Whisker.

Tips for optimal client performance — avoid screen savers

To be conservative, and have maximum system performance, I suggest *not* running a screen saver. Switch your monitor off to save electricity.

General performance tips — turn off 'smooth scrolling'

Windows NT often installs itself to use 'smooth scrolling' – when you drag a scroll bar, the screen scrolls pixel row by pixel row, giving a pleasant but terribly slow scrolling effect. Personally, I hate this; also, while the system is busy scrolling, it isn't doing useful things. To speed up your work and improve system performance, disable this feature.

The easiest way is to install **TweakUI,** from Microsoft, part of the 'PowerToys' kit. (UI stands for user interface.) Once installed, you can choose *Start* → *Settings* → *Control Panel* → *Tweak UI* → *General* and turn off *Smooth scrolling* — and also turn off *Window animation*, another annoying decoration.

If you can't find TweakUI, run RegEdit instead. Choose *Start* → *Run* and type in `regedit`↵ (where ↵ stands for Enter). The smooth scrolling setting is in HKEY_CURRENT_USER\Control Panel\Desktop\SmoothScroll, which is a DWORD value and should be 0 for 'off'. The window animation setting is in HKEY_CURRENT_USER\Control Panel\Desktop\WindowMetrics\MinAnimate, which is a string value, and should be '0' for off.

Further tips for a fast computing experience

Practical ways to speed up your Windows computer

These tips won't have much of an impact on WhiskerServer performance, but might affect your client tasks' performance if you use the computer for other things (wordprocessing, analysis) at the same time. They also make for happier high-performance computer users.

User interface:

- Turn off all special effects. Under Windows 2000, choose *Control Panel* → *Display* → *Effects*. Turn off 'transition effects'; turn off 'show window contents while dragging'.
- Avoid large desktop 'wallpaper' pictures.
- Install Microsoft TweakUI. Then choose *Control Panel* → *TweakUI* → *General*. Turn off 'list box animation', 'menu animation', 'menu fading', 'menu selection fading', 'smooth scrolling' (an option particularly capable of crippling fast computers), 'tooltip fading', 'window animation'. (You may also like to turn on 'prevent applications stealing focus', which doesn't affect the computer's speed but may reduce your tension level.)

Memory and disk:

- Install plenty of memory. Under Windows NT/2000, 128 Mb gives comfortable performance with a fair number of applications running simultaneously; 256 Mb gives good performance. Many computer manufacturers install fast processors and little memory; for example, I recently used a Pentium-III, 750-Mhz machine with 64 Mb RAM, which was vastly inferior to an AMD K6 at 233 MHz with 256 Mb.
- Defragment your hard disk occasionally. A minor point in comparison to everything else.

Shortcuts

Keyboard shortcuts:

- If you have a Windows key (labelled ⊞), remember that ⊞–E brings up Windows Explorer, ⊞–M minimizes all windows and ⊞ alone brings up the Start menu. (There are other such shortcuts.)

- Alt–Tab cycles through running programs; Alt–Escape cycles through open windows.

Explorer shortcuts:

- *Command Prompt Here* — Windows NT 4. To be able to right-click a folder and pop up a command window in that folder, run Explorer / View / Options / File Types / File Folder [NB different from Folder] / Edit / Actions / Command Prompt Here / Edit. Then change d:\winnt\command.exe /k cd "%1" to d:\winnt\system32\cmd.exe /k cd "%1"
- *Command Prompt Here* — Windows 2000. Paste the following text into a file and save it with a .REG extension:

```
Windows Registry Editor Version 5.00
[HKEY_CLASSES_ROOT\Directory\shell\Command]
@="Command &Prompt Here"
[HKEY_CLASSES_ROOT\Directory\shell\Command\command]
@="cmd.exe \\\"%1\\\""
```

  Then double-click the file to import it into the registry. You should then be able to right-click any folder in Windows Explorer, and choose *Command Prompt Here* to open one at that location.

Part VII

Auxiliary programs

7 Auxiliary programs

A number of programs are supplied with Whisker, other than research tasks themselves. They are:

- **WhiskerStatus**, a status monitoring client.
- **WebStatus**, a web-based status monitoring client.
- **WhiskerTestClient**, a program to communicate with and test WhiskerServer directly via a network.
- **WhiskerReset**, to ensure all devices are in a consistent state when a computer is started.
- **Whisker Database Manager**, to help you manage your databases.

7.1 WhiskerStatus

This program allows you to find out the status of your server(s). In conjunction with well-written clients (behavioural tasks), you will be able to find out the status of each of your subjects without leaving your office.

Run the WhiskerStatus program (*Start* → *Whisker* → *Utility clients* → *Whisker Status Client*). When you start it, and whenever you open a new window, it will ask you for the name and port number of the server you wish to connect to.

The name *loopback* (or *localhost*) may be used as a pseudonym for the machine you are presently using, i.e. if the server is running on the same computer as the status program. If the server you are interested in is on a different computer, enter its IP name/address (e.g. *somewhere.psychol.cam.ac.uk* or *131.111.190.22*).You should normally leave the port number at its default value of 3233.

The status program will then attempt to open a connection to that server, and will display messages telling you how it's doing. Once it's connected, then you can click the lightning symbol, or choose *WhiskerStatus* → *Update server status* to request a status display from the server.

The default status display lists all the clients (behavioural tasks) connected to the server, showing important information about who they are and what they are doing:

```
--- Start of WhiskerStatus report

Client 4
Client name: Second-order IVSA (box 0)
Client status: Box 0 (F6) - active 34, inactive 5, stim 3, reinf 0 - Task started
Client last communicated 0 min 3 sec ago

Client 5
Client name: WhiskerStatus client (C++ version)
```

```
Client status: OK
Client last communicated 0 min 0 sec ago

--- End of WhiskerStatus report
```

You will always see the WhiskerStatus program itself in the display (in this example, it's client 5). The status display may be improved in future versions!

The **Name** and **Status** fields are provided by the client. You can see how useful it is to program your clients so they tell the server what the subject is doing, because the server can then pass this information on to the status program somewhere else on the network.

Remember that if a client closes the connection to the server, the server won't know about it. If your client closes the connection when the task is finished, you will have to learn to equate 'absence of client on the status list' with 'the client has finished'.

By clicking on the blank page icon, or choosing *File* → *New*, you can simultaneously open connections to several servers, one per window. The title of each window shows the server it is connected or attempting to connect to.

You can copy the status program, `WhiskerStatus.exe`, to another computer (its default location is *\Program Files\WhiskerControl\Whisker\WhiskerStatus*); it requires no extra software to run (except that TCP/IP must be installed on the computer).

7.2 WebStatus

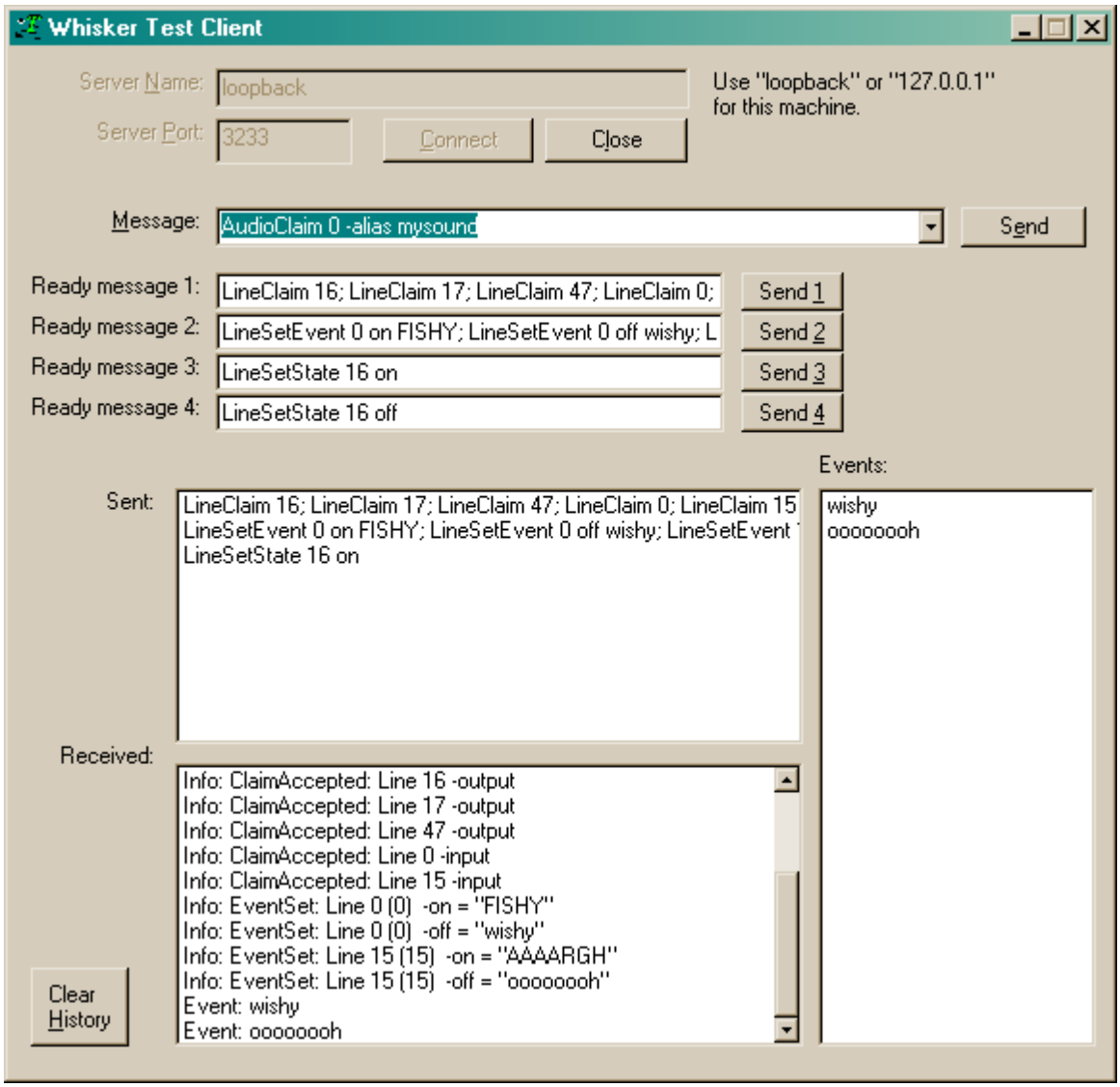
You can use a web browser to access the same status information that is displayed by WhiskerStatus. In order to do this, you must have configured the machine that is running WhiskerServer to host a 'status' web page (stored in \Program Files\WhiskerControl\Whisker\WhiskerStatus\Web). A suitable web server (Xitami) is distributed with Whisker; instructions for this package can be found with the installer.

Technical note: Why do we require a web server on the same machine?

In order to get the status information onto a web page, the applet hosted by that page must make a connection to the WhiskerServer. Within the security model for Java Applets is the restriction that Java Applets may only open connections to the host from which they were loaded.

7.3 WhiskerTestClient

You can use this program to communicate with the server manually. It does incredibly little but is a handy programming and test tool nonetheless.



Connecting to the server. First, you need to choose a server and click *Connect*. The default server address is 'loopback' (or 127.0.0.1), which is computer-speak for 'this machine'. To communicate with WhiskerServer when it is running on another machine, enter that machine's IP address – the address that looks like 'somewhere.psychol.cam.ac.uk'. The default *port number* is 3233; you probably won't need to change this.

Sending commands. You can type into the box marked 'Message' and click *Send* (or press Enter). Click the drop-down arrow to see a list of commands that the server should recognize.

Have a play. Try claiming an input line, requesting an event on it, and playing with the sensor it's connected to. Try claiming an output line and setting its state. (The commands to perform these tasks are described fully later; see the *Programmer's Guide*.) Explore the server console at the same time. Try enabling the communication logs for client you are using. You'll be able to find a log of everything that occurred, and be able to watch the lines and events fly back and forth.

Every message you send is shown in the window labelled 'Sent'. Messages that come back from the server, plus a few status messages from the program itself, are shown in the box labelled 'Received'. Any messages that come from the server and begin with 'Event:' are shown in the 'Events' window, with the event prefix removed.

You can type in (or copy/paste in) a further set of messages into the Ready Message boxes, and send them rapidly, either by clicking the buttons "Send 1" to "Send 4", or by pressing **Alt-1** to **Alt-4** .

Click *Clear event history* to wipe the slates clean.

Technical note

Using an immediate socket system with this client requires that you fire up a second copy, using a different port number. This program hides almost nothing from you - useful for learning how the system is put together, if you like that sort of thing.

The client was written in Visual C++ as a dialogue-based application. It does not use the client library that was used to develop the behavioural tasks. It is a derivative of the first Windows socket-based program written as part of the Whisker project. It's done well.

7.4 WhiskerReset

This is a command-line utility that can be used to switch all the lines on or off when the server is not running (or when Windows is not running).

Usage:

```
WhiskerReset [ReverseOutputs | On | Off]
```

ReverseOutputs and *On* both turn all lines ON.
Off, the default, turns all lines OFF.

By default, WhiskerReset is run automatically when you boot a computer with Whisker installed; a Technical Note on the installation process explains how to alter this behaviour.

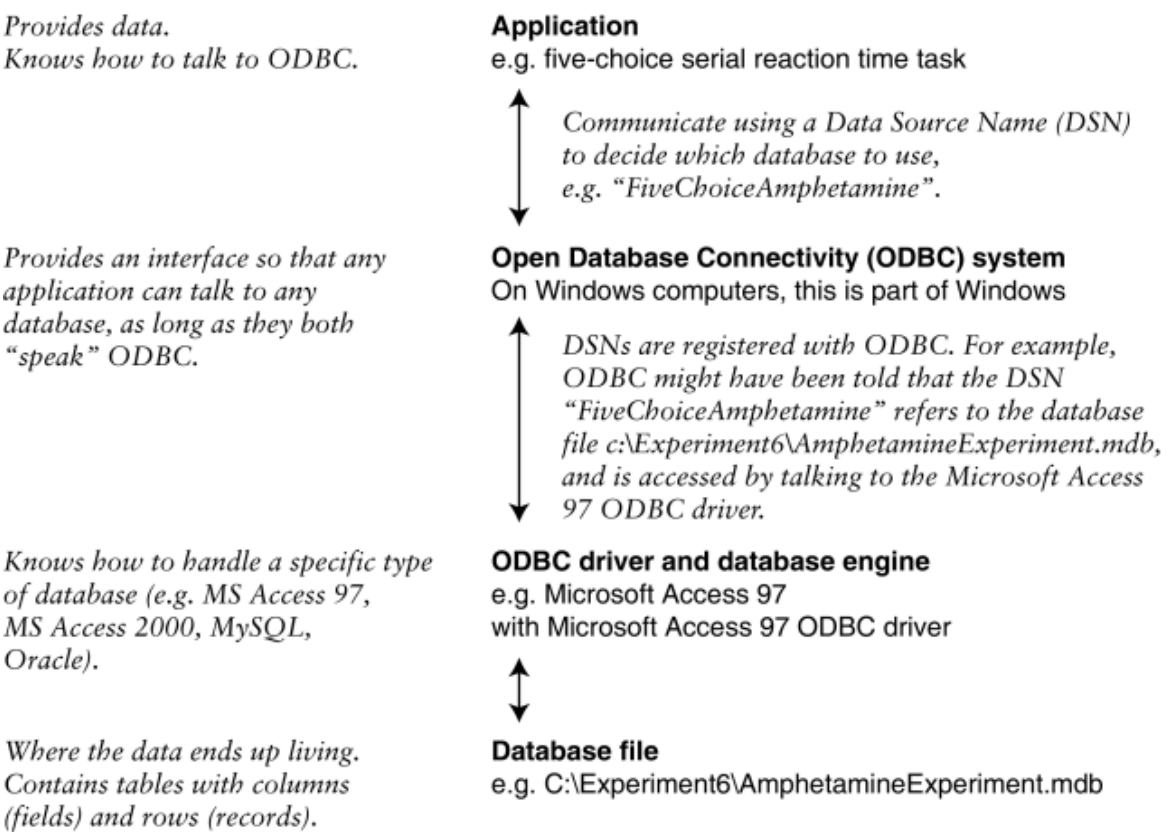
Note

- Digital IO lines connected to a BNC controller device or an ICS / Advent card will not be affected by the WhiskerReset program.

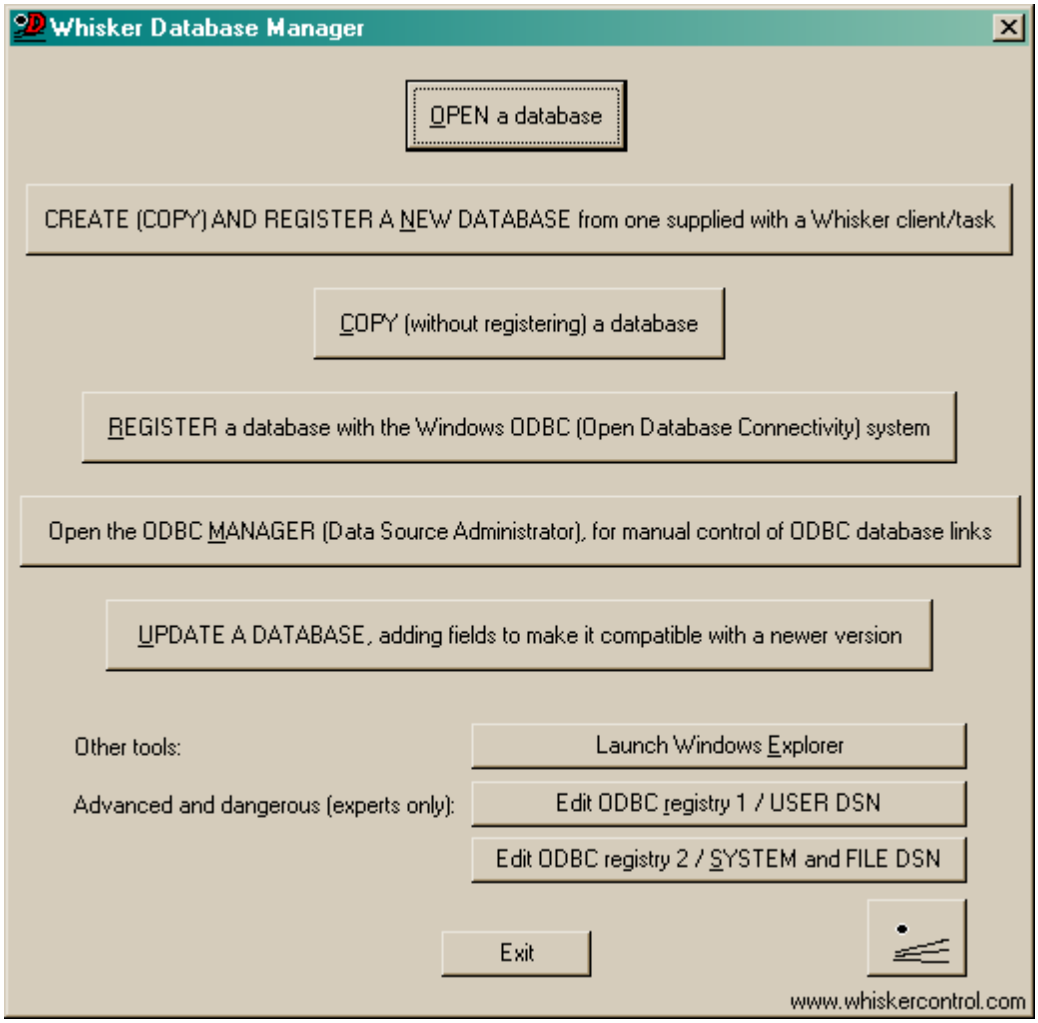
7.5 Whisker Database Manager

Many Whisker clients (certainly those written by the authors of Whisker) use relational databases to store data, since this is one of the most powerful and flexible ways of managing the large quantities of data that can be generated by behavioural experiments. They do so using the **Open Database Connectivity (ODBC)** interface.

The principles of ODBC are as follows:



The **Whisker Database Manager** is a small program that may assist with database management. For example, it can help you register your databases with ODBC so that they are accessible from Whisker clients. The Database Manager looks like this:

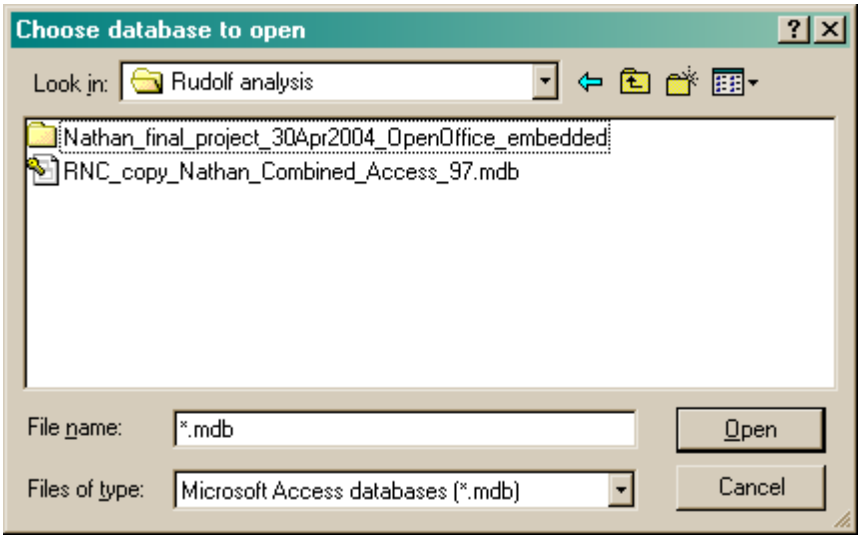


The **Exit** button quite, and the **www.whiskercontrol.com** button launches the Whisker web site in your default web browser. For details of the other options, see the following entries:

- Open a database
- Create (copy) and register a database
- Copy a database
- Register a database with ODBC
- Open the ODBC Manager
- Update a database
- Launch Windows Explorer
- Edit ODBC registry 1 / user DSN
- Edit ODBC registry 2 / system and file DSN

7.5.1 Open a database

Open a database. This opens a database file using the program that knows how to do so. It is directly equivalent to double-clicking on the file in Windows Explorer. For example, if you have Microsoft Access 97 installed, and you open a .MDB file, the database will be opened in Access.



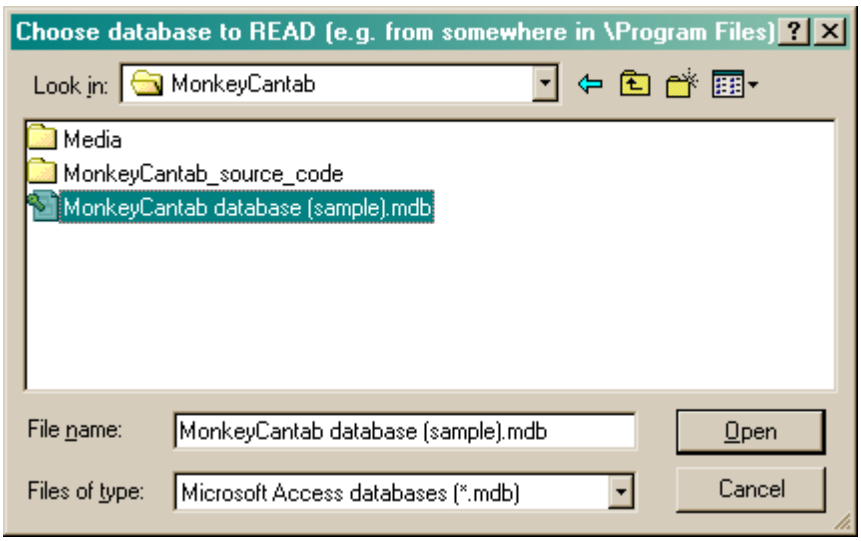
7.5.2 Create (copy) and register a database

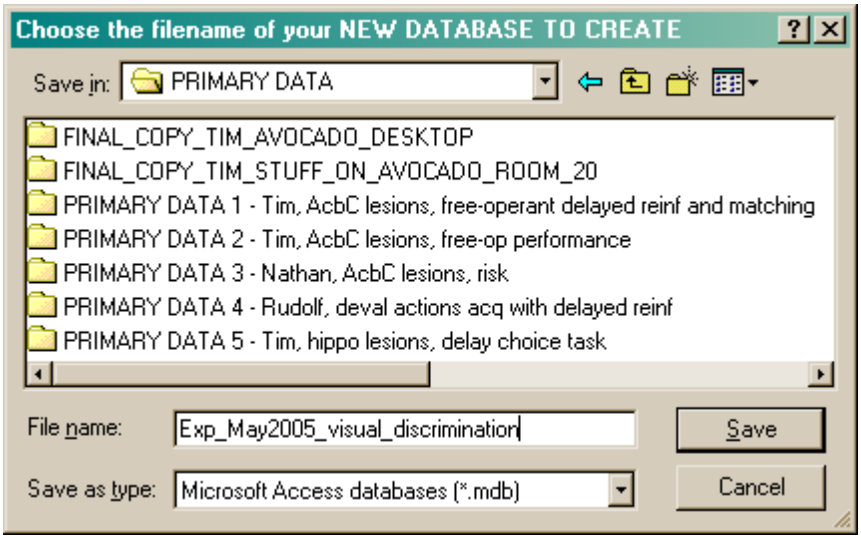
This combines the steps of *copying* a database and *registering the new copy with ODBC*. These steps (copying and registration) are shown separately below.

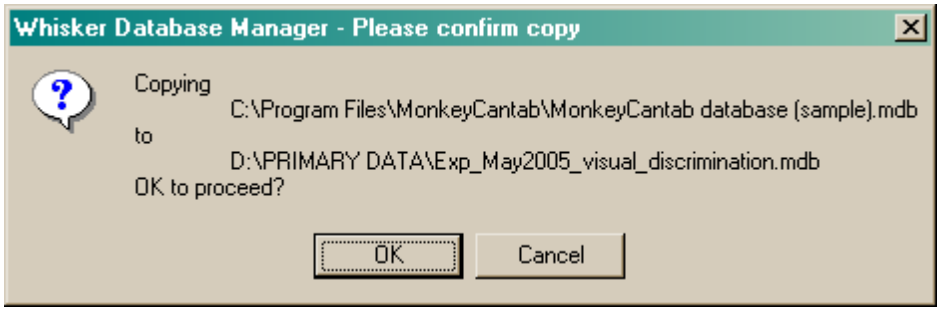
- Copy a database
- Register a database

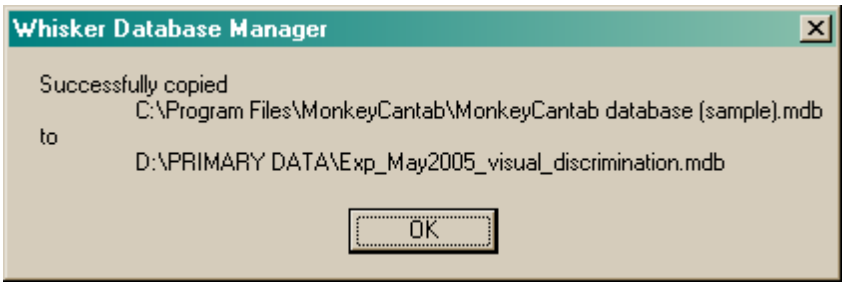
7.5.3 Copy a database

This simply makes a copy of an existing database file! You could also use Windows Explorer for this, or a DOS prompt. The program will not let you overwrite an existing database, for safety reasons.



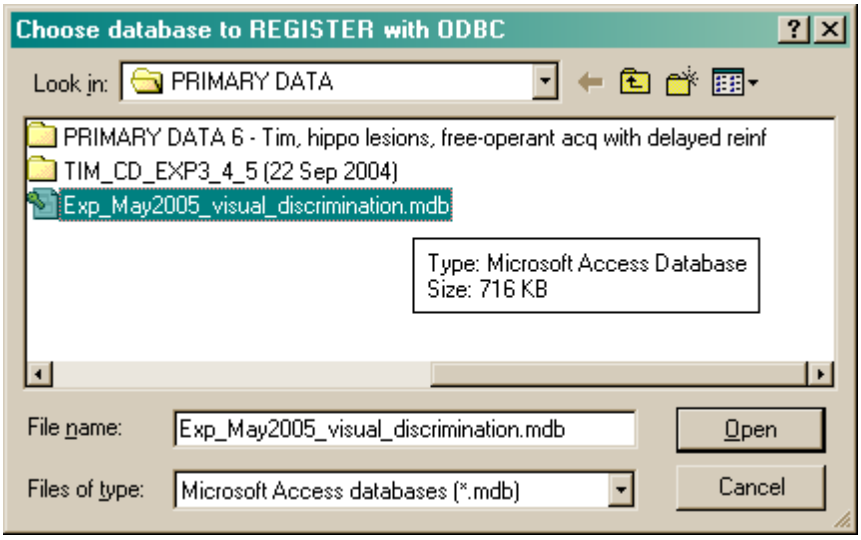




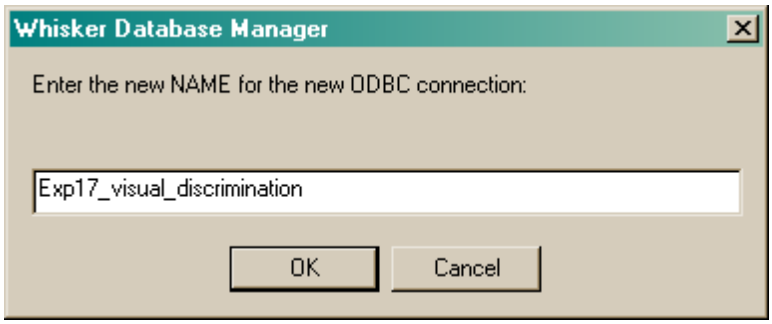


7.5.4 Register a database with ODBC

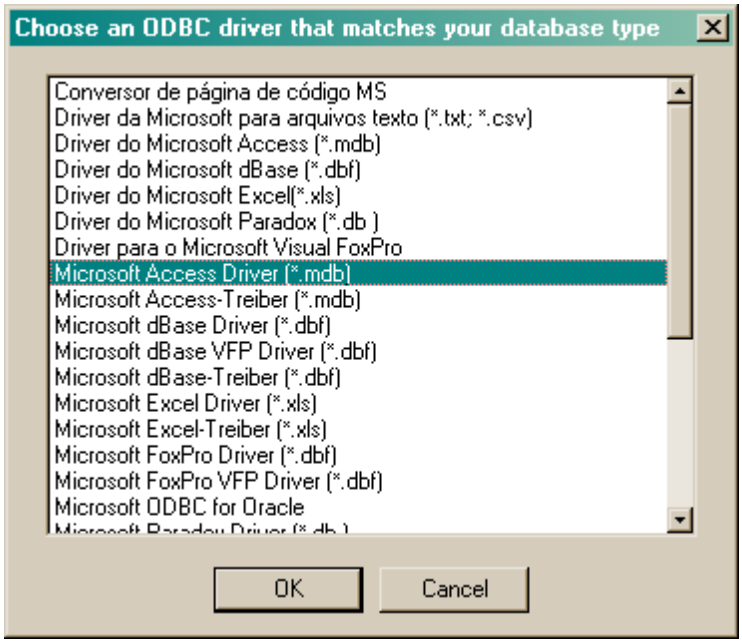
This allows you to register a database with ODBC, giving it a Data Source Name (DSN) in the process. You can do the same directly with the ODBC Manager, but this may be slightly simpler for some people. First, choose your database file:



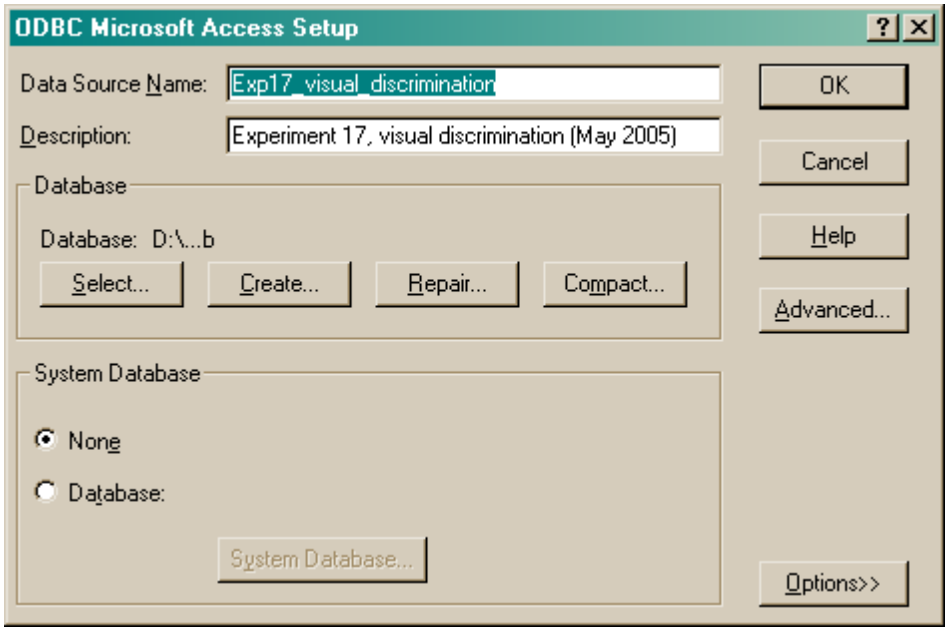
Now, on to the registration process:



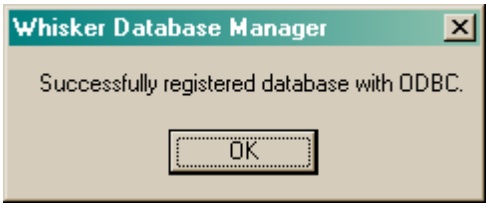
You'll need to specify what kind of database file it is:



Having gathered this information, it is fed into the ODBC Manager, which appears like this:

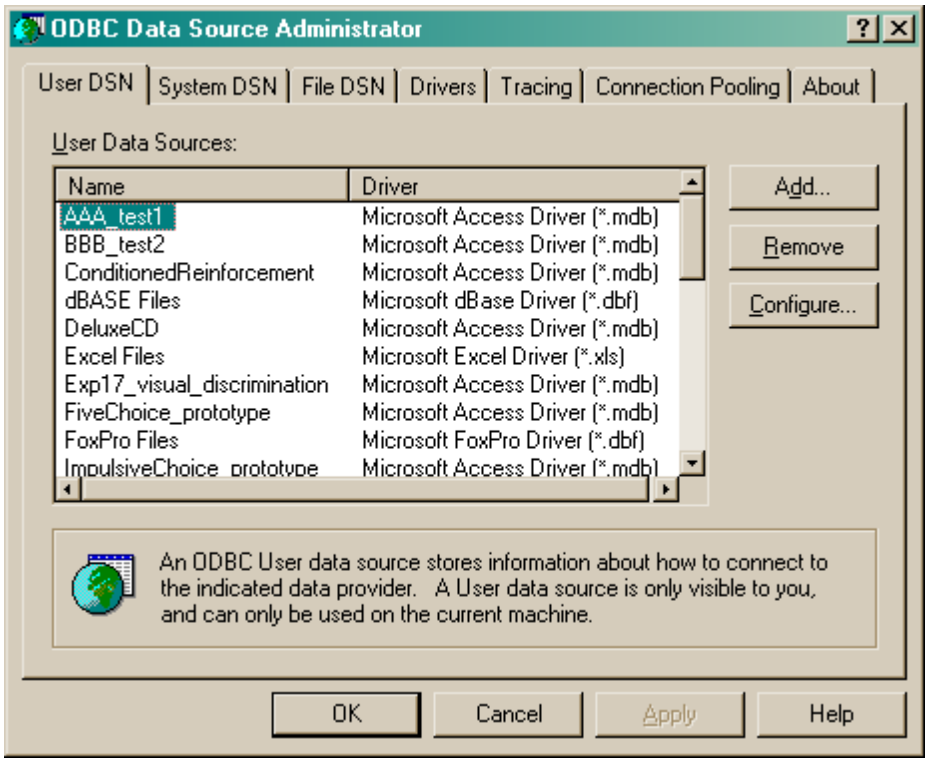


You can edit the settings at this point, and when you click OK the database is registered.



7.5.5 Open ODBC Manager

This opens the ODBC manager itself [you also get there by clicking *Start* → *Settings* → *Control Panel* → *Administrative Tools* → *Data Sources (ODBC)*]. You can add, remove, and modify DSNs (and what they refer to) here. I suggest you keep your data sources (DSNs) in the "User" category, which are visible to the current user but not to other users of the same system. This is where the Whisker Database Manager will register DSNs for you. The "System DSN" category is an alternative; DSNs here are visible to all users of the system.



7.5.6 Update a database

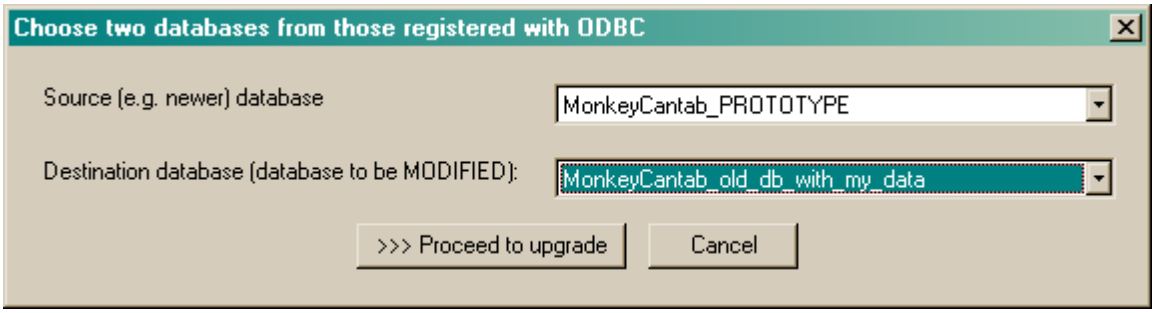
To explain this option, let's imagine a common scenario. Your pet programmer or software supplier gives you a lovely behavioural task. The task knows that it should store its data in a database (as well as in text files, just to make sure no data ever get lost). The task knows the names of the **tables** and the **fields (columns)** in those tables. For example, it might want to store the total number of responses in a task involving simple schedules of reinforcement, and it might know that its database should contain the *SimpleSchedules_ResultSummary.TotalResponses* field (that is, the field named TotalResponses in the table named SimpleSchedules_ResultSummary). All well and good - and with the task itself is supplied a **prototype** database - i.e. one that contains no data, but contains the **table structure** that the task needs. You are **not recommended to store data in the prototype database,** in case you accidentally delete it while cleaning out your *C:\Program Files* directory, but you are recommended to **make your own copy of the prototype,** or several copies for different experiments, and use those copies.

But now suppose you receive a software upgrade. Let's suppose you use the MonkeyCantab test battery. Now, suppose you or somebody else requests a new feature (for example, the ability to rotate target stimuli in the delayed matching/nonmatching to sample or DMTS task). And then you receive a new version of MonkeyCantab, with a brand new prototype database. There may be a

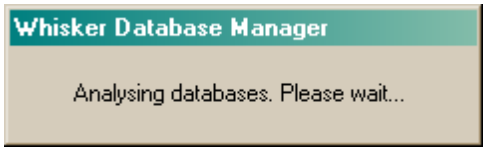
new field in the prototype database, such as *DMTS_Config.UseRotation*.

- If you use the old task and your current working copy of the (old version of the) database, all is fine.
- If you use the old task and a copy of the new prototype database, all is fine (since a field has been added, not deleted, in the new version).
- If you use the new task and a copy of the new prototype database, all is fine.
- But suppose you want to use the new task, but continue adding data to the same database as before? Well, you need to add this new field, *DMTS_Config.UseRotation*, because the new task will be expecting it. **The Whisker Database Manager does this for you.**

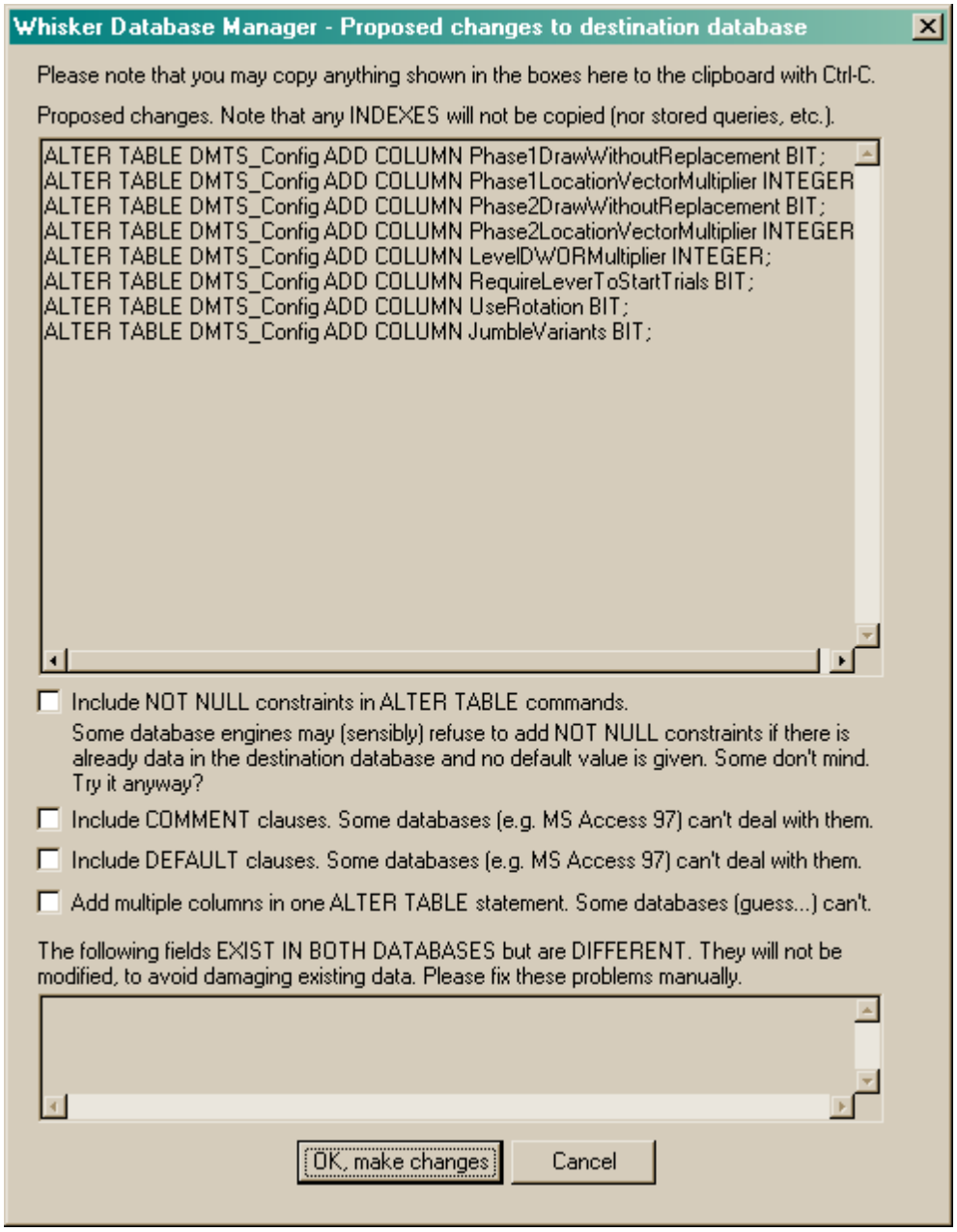
To use this feature, the new prototype database and the old (working) database must both be registered with ODBC. Then, simply choose the Data Source Names (DSNs) of the new (source) database, and the destination database (the one you want to upgrade):



Then click *Proceed*.



The differences between the databases will then be analysed, and commands generated in a universal database language called **SQL (Structured Query Language)** to make the destination database compatible with the source database. Various tick boxes are provided to alter the SQL slightly, since some databases (notably MS Access 97) don't accept standard SQL exactly. In the example below, there are no CREATE TABLE commands (so there are no whole tables that are present in the source but not the destination), but there are eight fields (columns) that are present in the source database's DMTS_Config table that aren't yet in the destination database's version.

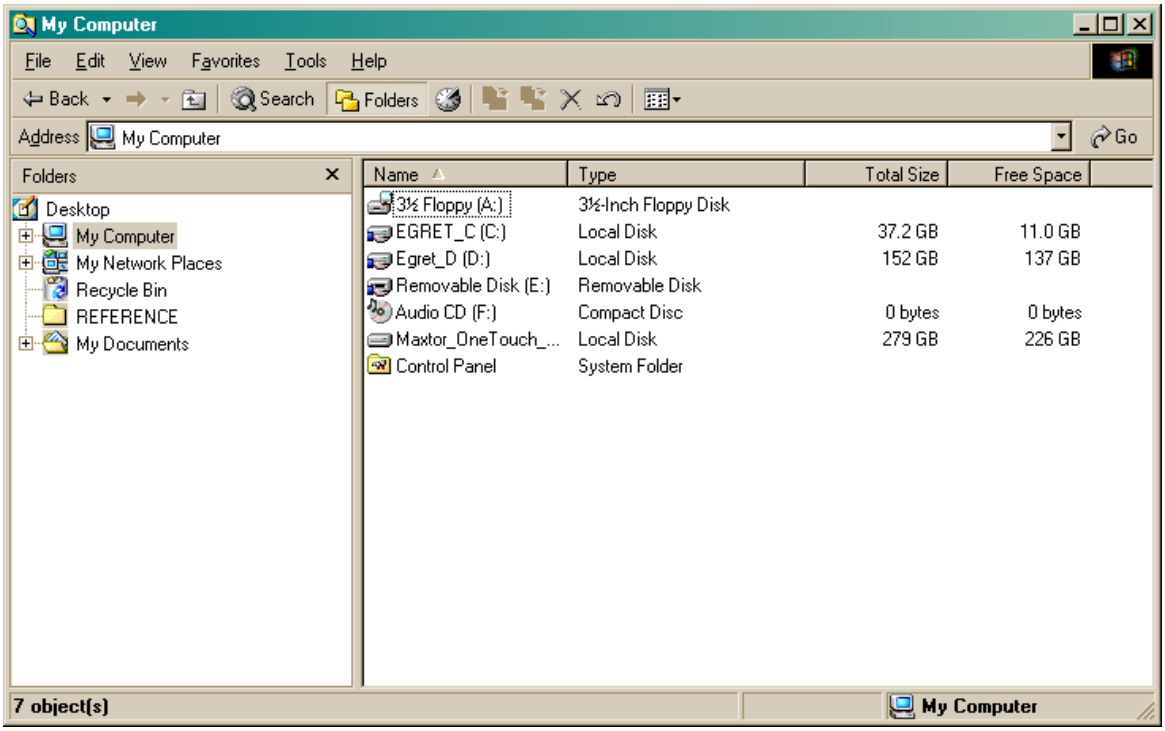


It may be that a field exists in both the databases but the field has a different type in each database. For example, if Table1.Field1 in the *source* database is of type VARCHAR(50) (meaning variable-length character data, or text, up to 50 characters long) and Table1.Field1 in the *destination* database is of type INTEGER (i.e. whole numbers), then there is an incompatibility. The Whisker Database Manager highlights these in the bottom edit box, and will **not** attempt to alter those fields (because that might cause loss of data). It's up to you to deal with this problem manually.

Click OK to make the proposed changes. If any errors are generated, a window will pop up showing the SQL command that failed, so you can work out what the problem might be.

7.5.7 Launch Windows Explorer

Opens the familiar Windows Explorer, allowing you to manage files. You might like to know that pressing the Windows key (⊞) and E together also opens Windows Explorer!

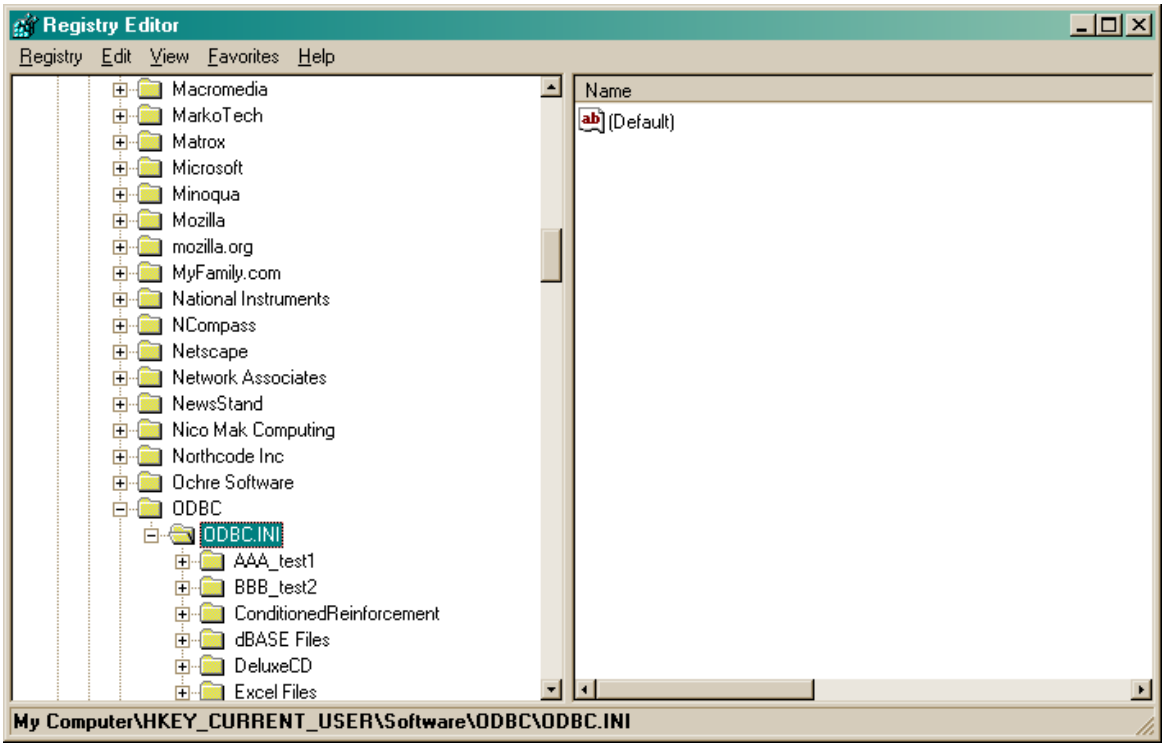


7.5.8 Edit ODBC registry 1 - user DSNs

WARNING: EXPERTS ONLY.

Launches RegEdit to examine the HKEY_CURRENT_USER\Software\ODBC.INI tree, where user data sources are managed.

DO NOT MODIFY SETTINGS UNLESS YOU UNDERSTAND THE CONSEQUENCES FULLY -- YOU COULD BREAK YOUR OPERATING SYSTEM USING REGEDIT INAPPROPRIATELY.

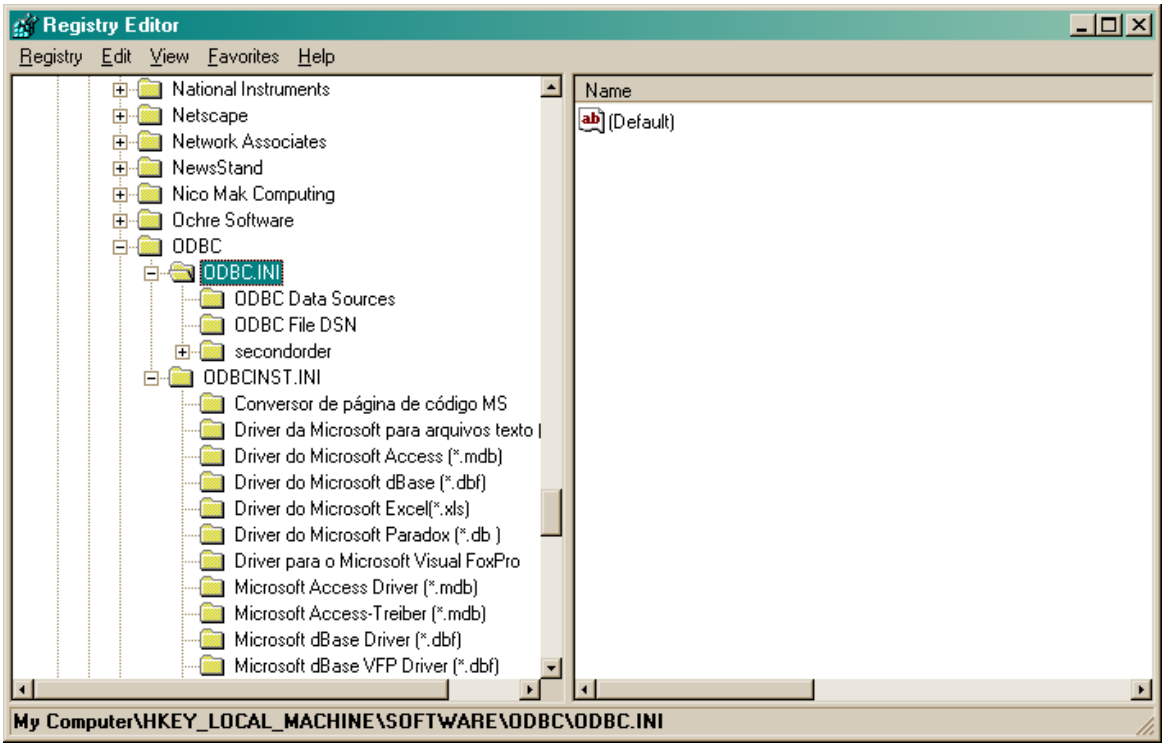


7.5.9 Edit ODBC registry 2 - system and file DSNs

WARNING: EXPERTS ONLY.

Launches RegEdit to examine the HKEY_LOCAL_MACHINE\Software\ODBC.INI tree, where system data sources are managed and file DSNs are pointed to.

DO NOT MODIFY SETTINGS UNLESS YOU UNDERSTAND THE CONSEQUENCES FULLY -- YOU COULD BREAK YOUR OPERATING SYSTEM USING REGEDIT INAPPROPRIATELY.



Part VIII

Programmer's Reference

8 Programmer's Reference

8.1 Introduction

The Whisker system is based on a **client–server** model. The **server** is a program that talks directly to the hardware controlling the operant chambers. The **clients** are programs that you can write that instruct the server *how* to control the chambers. A client implements a behavioural task, such as an FR-5 schedule of responding on a lever for pellets, or a five-choice serial reaction time task.

Several clients can communicate with a single server, and they operate independently. Because of this, you need only program the task client to cope with a single operant chamber, and then run several copies of the client when you want to use several boxes at once.

This brings several immediate advantages: (1) it simplifies the process of writing a task, because you only need to be concerned about one box; (2) it becomes almost impossible for a task to accidentally activate a device in the wrong box; (3) you can run any combination of different tasks at the same time without special programming.

Whisker commands are text messages sent between the WhiskerServer and the Client. How you write your programs will affect how much detail you need to understand about this communication. If you write your programs in VisualBasic, using the SDK Control, or in C++ using the ClientLibrary, you need not worry about this communication model at all – the tools will take care of it for you!

8.2 Technical description

The system is based on standard PC-compatible computers running Microsoft Windows® NT 4 or Windows 2000®. Standard digital input/output (I/O) control hardware from an electrical engineering company is used, along with Windows-compatible multimedia devices (in Multimedia editions of Whisker). Proprietary hardware has been avoided to minimize cost.

The Whisker server controls the digital I/O and multimedia hardware directly, and treats each input and output line as a resource to be managed. Clients communicate with the server (via the TCP/IP network protocol) and may request the use of a set of I/O lines, typically to control one or more operant chambers. Each client then instructs the server to switch output / control lines on or off, and asks to be informed of certain *events*. The client declares which events it wishes to be told about. For example, it may request to know about any of the following *events*: an input / response line goes on or off, an area of a touchscreen is pressed, or a certain time has elapsed. With this information, each client can implement a behavioural task.

This system was adopted in order to enable independent and different tasks to be used simultaneously without restricting the clients to a pre-specified programming language, as this can only reduce the flexibility of the system.

The inner workings of the system are described in full in the *Programmers Guide* so that you can program your own behavioural tasks with it if you choose.

Advantages

The use of a TCP event-driven server brings the following advantages:

1. It enables client applications to be written in any programming language that can use TCP communications. Users are therefore not tied to a particular language, client applications can be as sophisticated as you wish, and programming skills will transfer to other tasks. You can use a compiled language so that syntactic errors are picked up before you go live.
2. The server and clients can run on different computers if desired (though this relies on good network performance for real-time control).
3. Applications that are already geared towards TCP communications will readily lend themselves to further extensions. One such feature built into the system is the ability to find out your subject's progress, or even to control a session, from another computer in a remote location.
4. The communications system is designed for easy, logical programming. Particular attention was paid to avoiding the potential for programming errors that are common in some other operant control systems, such as conflicts between timers.
5. The system can run different client applications simultaneously, allowing different tasks to be used in different boxes at the same time. While applications may be written to control (say) four boxes simultaneously, it is easier and more useful to write a task for a single box; the system will then run four copies of this task and so every task written for the system allows boxes to be run asynchronously without special programming. You can even use one chamber for writing and debugging a new task without fear of disrupting the other chambers.

Other helpful features:

6. Behavioural tasks can be expanded to include functionality not supported by more specific pre-defined languages. For example, many clients in use within the Cambridge laboratories file their data directly into a relational database, as well as saving a human-readable account of the data.
7. The server allows certain output/control lines to be given special properties: *Failsafe* lines are used to ensure that the digital i/o system reverts to a safe state if there is a catastrophic failure (e.g. power loss to the computer). *Safety timers* can be set for any output/control line to ensure that a design flaw or communication loss with a client does not leave the system in a dangerous state for longer than a specified time. These features may be used with IV pumps, shock generators, etc.
8. Extensive support for the commonly used languages. The *Software Development Kit* for Whisker provides support to allow tasks to be very quickly and easily developed in the commonest languages: VisualBasic & C++. These tools can also be used in other RAD environments (such as Delphi, C#) which support the ActiveX model.

The computers are standard PC running Microsoft Windows®, so they can be used for other tasks (word-processing, data analysis) when not running behavioural tasks. Depending on your system's speed, you may even be able to use the PC for other things when it *is* running behavioural tasks. You can connect your operant control computers to your local area network to move data around the network, print to remote printers, etc.

Note: Advanced clients

- Clients could be written to implement any possible function, such as to interpret other popular behavioural control languages (e.g. there is no technical reason why a client could not be written to interpret Arachnid programs, or Med-PC® scripts).
- Clients do not have to run on the same machine as the server. They do not even have to run under the same operating system; while the server is Windows-based, clients could run under UNIX or any operating system that supports TCP/IP.

Limitations

Limitations of the system:

1. Typically, clients run on the same computer as the server; thus task responsiveness depends on the speed with which TCP messages get sent from one program to another within the computer. Whisker provides facilities to monitor this performance, and we consistently achieve time resolution within 1 or 2 ms. However, as with any multitasking system, very heavy processing loads can slow this down for brief periods - although resolution of >10ms is extremely rare. However, it is inadvisable to run other software at the same time as the operant control system until you have checked the effect of this software on your system's performance.
2. If the control computer is connected to a network (this is not a requirement for the WhiskerServer, but is required to take advantage of the network functions such as remote monitoring) it is potentially vulnerable to denial-of-service attacks from other computers on that network. If that network is part of the Internet, the range of potential attackers is large. This problem applies to any networked computer, not just the Whisker system, but the data collected via Whisker is likely to be very important to you! Similarly, it is a matter of choice whether you allow computers outside your network to connect to your Whisker servers. (By default, other computers are allowed to ask the server for status information, but not to gain control of any digital I/O hardware.)

Features not currently implemented:

1. A limited range digital I/O hardware is presently supported (Amplicon PC272 & PC272E cards and, as of version 2.4.0, Advantech PCI 1753 cards).
2. Devices other than simple digital I/O lines – for example, serial communications or analogue control – are not presently supported, although support for these devices is under development.
3. Clients must be independent, stand-alone programs, requiring users to have programming skills, although the choice of such languages is wide. (This was a deliberate design feature, to make the system as powerful as possible.) At present, there is no 'special' client that would allow you to design behavioural tasks in graphical form with no programming skills, but such a client could be written.

8.3 Communicating with WhiskerServer

This section details the precise implementation details of the format of the messages which are passed between WhiskerServer and Client tasks. This information is for those who wish to program WhiskerClients directly, and for reference.

Note that if you are using the SDK (ActiveX WhiskerControl for VisualBasic, or the C++-compatible WhiskerClientLib library) you will not need to worry about this section: the SDK will handle all socket implementation and communication details for you.

8.3.1 Communication: principles and message formats

Messages are passed between the client and server through a 'socket', which is a link between two programs through a network. The two programs can be on the same computer (I expect you will normally use this mode) or thousands of miles apart. The client/server computer(s) must have **TCP/IP** installed – the standard protocol for Internet communications. The client connects to the server by opening a connection with a standard **port number,** which is **3233.**

Each message constitutes a single **string.** This is a very general format, which is why I chose it. Most modern computer languages have no difficulty understanding and manipulating strings, and humans can understand string-based output. If the client or server needs to send a number, it is simply encoded in a string; thus 15 is sent as '1' (ASCII 49) followed immediately by '5' (ASCII 53).

For technical reasons, it is impossible to guarantee that two separate messages do not get concatenated when sent over the network (or that a single message is not split into two). Therefore, each string sent by the server ends in a **linefeed** (LF; ASCII code 10 decimal, 0A hex, also written as CTRL-J, `^J` or `\n`). When you send commands to the server, you **must** also end them with a linefeed or carriage return (CR), in case two of your commands become concatenated in transit, or one of your commands is split over two packets. You may also use a **semicolon** to separate commands.

If you need to pass a parameter that has a *space* or a *semicolon* in it, enclose the parameter in inverted commas (""). For example,

Message (↵ denotes carriage return or linefeed)	**Interpreted as… (• denotes separation of parameters within one message)**
`messageone;messagetwo`	`messageone` `messagetwo`
`messageone; messagetwo`	`messageone` `messagetwo`
`messageone↵messagetwo`	`messageone` `messagetwo`
`message param1 param2`	`message•param1•param2`
`message x y z`	`message•x•y•z`
`message "x y" z`	`message•x y•z`
`message x y;z`	`message•x•y` `z`
`message x "y;z"`	`message•x•y;z`
`message x"y;z"`	`message•x•y;z`

`message x "y↵z"`	`message•x•y`
	`z`
	this last example is probably an error on the part of the client!

The messages are described in the Whisker Command Set documentation. Strings to be quoted literally (in computer-speak, 'string literals') are shown in plain or `monospaced` type; parameters are shown in <*italic*> type surrounded by angle brackets (< >).

Client commands do not use a colon and are not case-sensitive. The server will accept multiple commands in one packet if they are separated by a carriage return (CR, `^M`, CTRL-M, ASCII 13, `\r`), a linefeed (LF, `^J`, CTRL-J, ASCII 10, `\n`) or a semicolon (; or ASCII 59). Quotes (inverted comma, ", ASCII 34) may be used to surround parameters that include spaces or semicolons, as described above.

Server responses use a colon, and will always give messages as they are described in this manual – CapitalizedWords with no space between them, a colon, a space and the rest of the message. (Exceptions: a colon is not given for Ping and PingAcknowledged commands, which can be sent by either the client or the server.) The server will always terminate each of its messages with a LF (\n, ASCII 10).

8.3.2 Creating and connecting sockets

'So how on earth do I make a socket?'

By far the easiest way to do this is by using the Software Development Kit. This provides all the sockets and message programming you need, and you will never have to worry about how messages work. If you are not using the SDK, then the way of making sockets will depend on the language you are using – but be assured that it is easy! Concrete examples are given later.

8.3.3 A dual-channel communication system: a general-purpose and an immediate-response socket

Caution: technical information!

The event-driven system and the separation of client and server is an enormously powerful technique, but it involves one area of complexity. That concerns queries that require an immediate and specific response.

For example, if (as the client) you wish to know whether an infra-red nosepoke detector is on or off, you may ask the server. But you do not wish to pause while waiting for the answer (you wish to know the device's state *now*, and you do not want to hold up processing of other ongoing events).

Imagine that as you begin to ask this question of the server, the server has detected a lever-press for which you have requested notification. It sends the event message at the same instant you send the query about the nosepoke detector. We have a dilemma: you don't want a lever-press event in response to your nosepoke query, you don't want to process the lever-press event until you know the nosepoke detector status (let's say), and you don't want to have your 'nosepoke query' routine handle lever-press events itself (that would be needlessly difficult to program in a task of realistic complexity!).

This is a problem whenever we have *one channel of communication* and when the server may send event messages at *unpredictable times*. If the 'client' and 'server' were the same program (as in every other operant control system currently on the market, to my knowledge) it would be easy to organize the program so the nosepoke detector request could take priority and hold up processing of the lever-press event. But this would defeat the many advantages of client–server separation. So we must get around the problem.

The way I have chosen to do this is to use **two communication channels** between the client and the server. Now, you don't have to use two channels, but whenever this problem crops up it is a good solution (and believe me, it crops up all the time). In the network system we are using, communication channels are known as *sockets*, so this is a two-socket system.

One socket is called the **main socket.** You may send commands to the server through this socket, and the server will send all of its event notification messages (and other unsolicited information) down this socket. The other is a called the **immediate socket.** The server guarantees two things

about communication down the immediate socket:

1. The server will send nothing to you through this socket, except in direct response to commands that you send to it through this socket.
2. Every command you send will be responded to immediately, and that response will comprise **a single message,** or one line of text.

As the client, you should implement the immediate socket as a **'blocking'** socket (I apologize for the networking terminology; this means that when you say to the socket 'give me the server's reply', it waits for that reply if necessary and then returns it to you; the socket never says 'sorry, no information has come from the server yet').

Don't worry, you shouldn't have to work with this system directly – all the technical detail is implemented for you in the SDK.

You cannot connect more than one immediate socket to a given client.

Clients can issue commands on either the main or immediate socket. If the comand is issued through the immediate socket, as most will be, the Server will send a reply (an `Info:` message about success, or an `Error:` or `SyntaxError:` message about failure) to the main socket only. The possible replies are shown in the reference above. Responses down the main socket give a good deal of ancillary information and are useful to the watching human.

If a command is sent to the immediate socket, then a single reply will be sent to the immediate socket. In general, responses to commands down the immediate socket are much simpler and more sharply defined, and consequently easier to deal with for the client task. If the command fails, the server will also send the relevant `SyntaxError:` or `Error:` report to the main socket, in addition to the message on the immediate socket. If the command succeeds, no `Info:` message will usually be sent to the main socket.

Note once more that every message coming from the server will be terminated by a linefeed (LF, ASCII 10, `\n`), and that client messages should be terminated by a linefeed, CR or semicolon.

8.3.4 Network responsiveness

One final technical point: you must disable the Nagle algorithm on your client's sockets (by instructing them to set the parameter TCP_NODELAY). This guarantees crisper network transmission and better latencies (albeit at the expense of more traffic). Otherwise, the client's TCP stack might sit around waiting for you to send enough information to fill up a packet – meaning that your vital command hasn't got through to the server on time.

The SDK performs this chore for you automatically.

8.3.5 Technical notes on TCP/IP methods

Why two communications channels?

The network administrators out there may be wondering why I didn't use out-of-band transmission on the main socket: because not all client languages (and not even all TCP implementations) support it.

Wide-area networks

The more network is in between the client and server, however, the less rapid the system's response becomes (especially

on a busy network). This is why I recommend keeping the client and server on the same computer. However, it is easy to find out the status of the operant chambers from a different computer; I imagine this will be the main practical use of this feature. The design didn't depend on this, though; if you're going to separate the client and server, you need a communication protocol, and TCP/IP happens to carry associated benefits.

TCP port numbers

Every type of program that communicates over the Internet (or similar TCP/IP networks) has a *port number*. For example, the Web uses the HTTP protocol, which runs on port 80. The Whisker defaults to using **3233** as its main port. This use is registered with the Internet Assigned Numbers Authority (IANA), so should not clash with any other functions on a network.

Network speed and concatenation of packets

For those interested in such technical matters, the server disables the Nagle TCP algorithm to ensure that all packets are sent immediately (this is achieved by instructing its sockets to set the parameter TCP_NODELAY). Nevertheless, if the server sends two packets in very quick succession, the TCP algorithm amalgamates them. This does make sense from a network performance point of view.

A similar problem is encountered when a large(ish) packet is sent – it is possible that the TCP/IP stack splits a message into two packets. For these reasons, there must be enough information in the data itself to establish where one command ends and the next begins, so a terminator signal (CR / LF / semicolon) is used.

8.4 SUMMARY OF WHISKER COMMANDS

The heart of the Whisker control system is the exchange of messages between Whisker server and those clients to which it is connected. First, the scope of Whisker will be outlined by listing the purpose of all the commands you can send to it. After that, the communications system will be described in detail, and in Part 2, each command and message will be described in full.

In the text, messages sent by the server will be `blue` and the commands sent by the client will be `red`.

Messages from the Server

These messages are sent down the main communication channel. They are unpredictable in that they may be sent at any time. The arrival of these should cause the client to deal with them 'as soon as possible': this is handled for you in most event-driven programming environments (such as Visual Basic and Visual C++).

- `Event:`
  When the server detects an event that the client has expressed an interest in, such as a digital line changing state, a timer triggering, or a picture being touched, it sends an `Event:` message to the client.
- `Info:`
  `Info:` messages are sent to inform a client about the completion of commands issued by that client.
- `Error:`
  If a command cannot be completed, an `Error:` message will be sent.
- `SyntaxError:`
  If a command is unrecognised, or malformed, a `SyntaxError:` message will be sent.

- ClientMessage:
  If a client allows, the server may pass on a ClientMessage: from one client to another (typically a 'remote-control' message).
- KeyEvent:
  If a client attached to a multimedia edition of Whisker has asked for keyboard notification for a display which has the input focus on the server, that client may receive KeyEvent: messages when a key is pressed on the keyboard.

Commands to the Server

Clients may issue a variety of commands to the server. These commands can be grouped into *device* commands - which control or interrogate the hardware managed by the server, and *general* commands - for example, those to alter the way in which the server communicates with the client. Devices which can be controlled by a client are **timers**, digital I/O **lines,** and (for multimedia editions of Whisker only) video **display** devices and **audio** output channels.

Timer Devices

Timers are simple software devices which are used to generate events after pre-specified intervals.

- TimerSetEvent
  Creates a timer.
- TimerClearEvent
  Cancels (kills) a timer.
- TimerClearAllEvents
  Cancels (kills) all timers.

Line Devices

- LineClaim
  Takes control of a digital input or output line.
- LineSetAlias
  Gives the line one or more alternative names.
- LineRelinquishAll
  Gives up control of claimed lines.
- LineSetState
  Sets the state of an output line (or lines).
- LineReadState
  Reads the state of a line (be it an input or an output line).
- LineSetEvent
  Causes Whisker to generate an event when the state of the line changes in a specified way.
- LineClearEvent
  Clears a specific event associated with a line.
- LineClearEventByLine
  Clears all events associated with a line.
- LineClearAllEvents
  Clears all events on all line devices.
- LineSetSafetyTimer
  Protects critical devices (such as intravenous infusion pumps) by specifying that the line should return to a safe state after a certain time.

- LineClearSafetyTimer
  Clears a safety timer.

Audio Devices *(Multimedia Edition only)*

- AudioClaim
  Takes control of an audio output device.
- AudioSetAlias
  Gives an audio device an alternative name.
- AudioRelinquishAll
  Relinquishes control of all audio devices.
- AudioPlayFile
  Plays a WAV (waveformat) file directly.
- AudioLoadSound
  Loads a WAV file, attaching it to a specified audio device.
- AudioLoadTone
  Creates simple sounds without the need for a WAV file.
- AudioPlaySound
  Plays a sound that has been loaded with AudioLoadSound or AudioLoadTone.
- AudioSetSoundVolume
  Sets playback volume for the device.
- AudioStopSound
  Stops playback, keeping the sound in memory.
- AudioUnloadSound
  Unloads the specified sound, freeing the memory it occupied.
- AudioSilenceDevice
  Stops playback of all sounds currently playing on the specified device.
- AudioUnloadAll
  Unloads all sounds from the device.

Display Devices *(Multimedia Edition only)*

To display things using Whisker, you place **objects** into **documents** and show those documents on **display devices.** A document's content may be altered whether or not it is being displayed.

- DisplayClaim
  Takes control of a 'physical' display (usually a dedicated monitor on a multimonitor computer).
- DisplaySetAlias
  Gives a display an alternative name.
- DisplayRelinquishAll
  Releases all display devices.
- DisplayCreateDevice
  Creates a 'virtual' device (usually a new window on the Windows desktop).
- DisplayDeleteDevice
  Deletes a virtual device created with DisplayCreateDevice.
- DisplayGetSize
  Finds out how big a display device is.
- DisplayKeyboardEvents
  Causes the display to respond to keypresses (only applicable when your display has the Windows input focus but useful if you want to use the keyboard as an input device in your

testing). Keyboard events are sent to the client with a special `KeyEvent:` message.

- `DisplayCreateDocument`
  Creates a document.
- `DisplayShowDocument`
  Places a document on a display device.
- `DisplayBlank`
  Removes a document from a display device.
- `DisplayDeleteDocument`
  Deletes a document.
- `DisplaySetBackgroundColour`
  Sets the background colour for a document.
- `DisplayAddObject`
  Adds an object to a document. (Objects can be of many types.)
- `DisplayBringToFront`
  Brings an object to the front of a document.
- `DisplaySendToBack`
  Sends an object to the back of a document.
- `DisplayDeleteObject`
  Removes an object from a document.
- `DisplaySetEvent`
  Makes an object respond to being touched or clicked with the mouse.
- `DisplayClearEvent`
  Clears an event created with DisplaySetEvent.
- `DisplaySetObjectEventTransparency`
  Determines whether objects behind the specified object are also allowed to respond to touches or mouse clicks.
- `DisplayEventCoords`
  Obtain high-precision information about the location of touches and mouse clicks.
- `DisplayScaleDocuments`
  Although Whisker will display documents sensibly, you can gain extra control over the way your documents are scaled (if need be, adopting your own coordinate system to scale documents for different monitor sizes) using DisplayScaleCoordinates and DisplaySetDocumentSize.
- `DisplaySetDocumentSize`
  Sets the logical size of the document (for use with DisplayScaleCoordinates).

Claiming groups of devices

- `ClaimGroup`
  You may claim digital I/O lines, displays, and audio devices by device number, or by using names that you define to the server using a device definition file. This allows you to define groups of devices (e.g. all those that correspond to a single operant chamber) and claim them all at once using ClaimGroup.

Timing of events

- `TimeStamps`
  Determines whether the server appends a time-stamp to all messages it sends to the client. This facility can be used to record accurate timing information.
- `ResetClock`
  Resets the timestamp clock to zero.

- `RequestTime`
  Requests the time as judged by the timestamp clock (in milliseconds).

Communications management

These commands control the exchange of messages with the server.

- `Shutdown`
  Closes all communications with the server, freeing all claimed resources in the process.
- `ImmPort:`
  Informs the client which TCP port number to use in order to establish a second communication channel with the server. (The second channel is called the "immediate" socket, as opposed to the "main" socket that is established when the client first connects to the server).
- `Code:`
  Informs the client of the code to use when using the Link command.
- `Link`
  Informs the server that a particular "immediate" socket is to be functionally linked to a particular "main" socket because both are owned by the same client.
- `TestNetworkLatency`
  Tests the speed of the network connecting the client and the server.

Status reporting and miscellany

- `Version`
  Detects the server's version information.
- `ReportName`
  Reports the client's name to the server.
- `ReportStatus`
  Reports the client's current status to the server.
- `ReportComment`
  Sends a comment to the server (primarily for debugging).
- `WhiskerStatus`
  Requests information about the clients connected to the server, and their current status. (Used to monitor the Whisker system as a whole.)
- `SetMediaDirectory`
  Audio and display devices may use multimedia resource files such as wave-format (WAV) sound files and bitmap (BMP) pictures. These must be stored on the WhiskerServer machine. Tell the server where to find them with SetMediaDirectory. If a file cannot be found here, the server will try its own resource directory before trying the raw filename.

Client–client communication

The server assists clients to communicate with each other directly, if so desired.

- `SendToClient`
  Sends a message to another client. (If the server and recipient permit the message, the recipient will receive the message in a `ClientMessage:` message from the server; *q.v.*).
- `PermitClientMessages`
  Tells the server whether the client will permit incoming messages from other clients.

Video commands

Video objects are controlled using these conventional object-manipulation commands:

- DisplayAddObject (option: video)
- DisplayDeleteObject
- DisplaySetEvent
- DisplayClearEvent
- DisplaySetEventTransparency
- DisplayEventCoords
- DisplayBringToFront
- DisplaySendToBack

... and these video-specific commands:

- DisplaySetAudioDevice
- VideoPlay
- VideoPause
- VideoStop
- VideoSeekAbsolute
- VideoSeekRelative
- VideoGetTime
- VideoGetDuration
- VideoTimestamps
- VideoSetVolume

8.5 Events: the core of the system

Events are the key to the Whisker system. What a typical client (behavioural task) program does is to say, in effect, 'Say *FISH* to me when the rat depresses a lever; say *TIME_UP* to me when 30 minutes elapse from now.' It then sits quietly and waits for the server to send it messages. When the appropriate lever is pressed (or the time elapses), the server sends it a message containing the desired text; the client may then take action (such as activating the pellet dispenser, or ending the session). The client may choose freely the text used for these events.

Event messages are marked in a special way (they're preceded by 'Event:') to distinguish them from other types of message. The detailed list of messages given on subsequent pages shows how this works.

8.6 Exploring the system with WhiskerTestClient

In the *User Guide*, we glanced at the WhiskerTestClient program. This might be a good time to run it, connect to the server, and try sending some messages. As a very simple example, try sending the following command:

```
TimerSetEvent 5000 0 Your_Time_Is_Up
```

This is one of the commands detailed in the rest of this section. It requests the server to set up a *timer* event that elapses in 5000 ms (5 s), never reloads, and sends the event 'Your_Time_Is_Up' when the timer elapses. You should see that the server responds immediately with

```
Info: EventSet: Timer 5000ms -reloads 0 = "Your_Time_Is_Up"
```

and five seconds later, the server should send

```
Event: Your_Time_Is_Up
```

This is a message that has the '`Event:`' prefix, and WhiskerTestClient is alert to these; in addition to displaying the message from the server in its Received window, it also displays

```
Your_Time_Is_Up
```

in its Events window. Events are the class of message that clients are most interested in responding to; were this a real client, it might be programmed to terminate a schedule at this point. Events are not restricted to timers; by using the `LineSetEvent` or `DisplaySetEvent` command, you can ask the server to send you a message of your choosing when your subject breaks an infrared detector beam, presses a lever, or touches a picture of an apple.

8.7 A reminder about non-local machines

Unless Allow non-local machines to control lines is ticked on the Server menu, non-local machines can only ping the server, test network latencies and request status information. All other commands generate an error message.

(If Reject all non-local connections is ticked, of course, non-local machines won't even be able to connect.)

8.8 THE WHISKER COMMAND SET

This guide explains the information you would need to write your own powerful behavioural tasks.

For every entry that details a Whisker command or message, most or all of the following headings may be shown:

Message

Shows the syntax of the command.
Bold or plain text is quoted literally.
<*parameters*> are shown in italics and in angle brackets (< >).
[<optional parameters>] are enclosed in square brackets ([ ]).
[Optional literal text] is also enclosed in square brackets.
Square brackets are also sent by the server to indicate timestamps. (See TimeStamps.)
Alternatives are separated by a vertical bar (|), e.g. one | two | three.
Information sent by the client is shown in red.
Information sent by the server is shown in blue.

Originator

Whether the message or command is initiated by the Server or the Client.

Response (immediate socket)

(For client commands only.) Gives the possible responses the server may make if the client sends the command via the immediate socket.

Response (main socket)

(For client commands only.) Gives the possible responses the server may make if the client sends the command via the main socket.

Details

Explains the command in detail.

SDK Control / SDK Control Method

For server messages, explains the manner in which the SDK responds to the server's message.
For client commands, gives the syntax used to pass the command to the server via the SDK control.

> **THE SDK DOCUMENTATION IS NOT YET COMPLETE. All commands have SDK versions: these have the same name as the WhiskerCommand messages that they instantiate. The documentation is in the process of being updated to reflect the SDK options; in the meantime, their usage can be easily surmised from their parameter names and the documentation for those commands that have been properly documented!**

ClientLib

Explains the manner in which the command is handled by the C++ client library (WhiskerClientLib).

Note: two versions of each of the virtual functions are provided: a version which contains timestamped information, and another which does not. If you do not override the timestamped version, the function without timing information will be called by the library. See WhiskerTask.h file for details.

Examples

Illustrates examples of the message.

8.8.1 The two-socket system

This table shows the sequence of events that occurs when a client connects to the server and wishes to use two sockets (a 'main' socket for normal communication, and an 'immediate' socket for instant, well-defined replies).

Main socket		Immediate socket	
Client	**Server**	**Client**	**Server**
(connects a socket on main port)		(socket not connected yet)	

	`Info: WhiskerServer v1.0, ...`
	`Info: You are client number XXX`
	`ImmPort: 1234`
	`Code: 1_41`		

		(connects a socket on port 1234)	
			(no response)

		Link 1_41	
			Success
The two sockets are now linked and operate as a functional pair.			

The main port is usually port **3233** (the Whisker port number as registered with the Internet Assigned Numbers Authority, IANA). The server will always inform the client of the immediate port number, which is a currently unused socket, picked by the operating system.

The C++ client library (WhiskerClientLib) and the SDK Control perform these tasks automatically when you ask for a connection to the server.

This section explains the syntax of:

- ImmPort
- Code
- Link

8.8.1.1 ImmPort

Message

ImmPort: *<port_number>*

Originator

Server

Details

The server sends this message when the client first connects to the main socket. It gives the port number to use for the immediate socket connection, should one be desired.

Revision history

Implemented in WhiskerServer version 1.

See also

- The two-socket system
- Code
- Link

8.8.1.2 Code

Message

Code: *<code>*

Originator

Server

Details

The server sends this message when the client first connects to the main socket, *after* the ImmPort message. It gives a code that the client needs to prove to the server that it is the genuine owner of its immediate socket. As soon as the client receives the Code command, it has all the information it needs to connect and link an immediate socket using the Link command.

Revision history

Implemented in WhiskerServer version 1.

See also

- The two-socket system
- ImmPort
- Link

8.8.1.3 Link

Message

Link *<code>*

Originator

Client

Response

This command does not work through a normal socket.

Response (immediate socket)

Success
Failure

Details

When a new immediate socket has been connected, the server does not know which client it belongs to. At this stage, the only command that the server will respond to on this socket is the Link command. As the client, you should send a Link command with the code you received from the server (see Code). Once the socket has been successfully linked to the client, it is fully operational.

This command only works when the immediate socket is not yet connected to a client. If you send it once the immediate socket has been connected, an error will be generated.

Revision history

Implemented in WhiskerServer version 1.

See also

- The two-socket system
- Code

8.8.2 Messages sent by the server

These are the messages that the Server may send to the Client's main socket. For each message, the raw message that is sent is shown, along with the programming information for the SDK tools.

For example, looking at the table for `Event:`, you should be able to work out the following:

- **Visual Basic:** to handle `Event:` messages, if you have a form containing an SDK control named, say, *SDKControl1*, then you should put your event handling code into the `SDKControl1_WhiskerEvent()` Subroutine. This code will be called whenever an `Event:` message is received.
- **C++:** If you have a class named `CMyTask` derived from the CWhiskerTask class that contains a function called `CMyTask::IncomingEvent()`, then the receipt of an `Event:` message will result in this function being called.

If the paragraph corresponding to your chosen language doesn't make sense, then you should make sure you learn more about the language before attempting to write behavioural tasks.

This section explains the syntax of

- Event
- Info
- KeyEvent
- SyntaxError
- Error
- Fault
- Warning
- Ping
- ClientMessage

8.8.2.1 Event

Message

Event: *<eventname> [<timestamp>]*

In full, there are several possible formats. Below, some are shown literally for an event called "PictureTouched".

```
Event: PictureTouched
```

... minimal

```
Event: PictureTouched (18005) 572 827 [18376]
       event          video   x   y   timestamp
                      t'stamp         []
                      ()
```

... event from a video object with video timestamps (round brackets), event coordinates (no brackets), and event timestamps (square brackets) all turned on

Originator

Server

Details

A line, timer or display touch/click event has occurred. This is the main message that the client needs to set up and respond to. It is always sent to the main socket.

SDK Control

WhiskerEvent fires *(Visual Basic: calls Control_WhiskerEvent Sub)*, with the *EventMessage* and *Time* parameters set to the appropriate values.

ClientLib

`virtual CWhiskerTask::IncomingEvent()` is called – if your task overrides this function, the override will be called with the *strEvent* and *lTime* parameters set to the appropriate values.

Examples

Event: SEEKING_LEVER_PRESSED_BOX_3
Event: ITI_expired

Revision history

Implemented in WhiskerServer version 1.
Video timestamps in version 4.0.

See also

- LineSetEvent
- TimerSetEvent
- DisplaySetEvent
- DisplayEventCoords
- Timestamps
- VideoTimestamps

8.8.2.2 Info

Message

Info: *<text> [<timestamp>]*

Originator

Server

Details

Used for all user-oriented reports (such as the response to a WhiskerStatus call). Box control programs can safely ignore Info messages. Info messages are usually sent confirming the successful operation of commands sent to the main socket, and are therefore useful for debugging; most of these messages are not sent for commands received on the immediate socket.

SDK Control

WhiskerInfo fires (*Visual Basic: calls Control_WhiskerInfo Sub*), with the *InfoMessage* and *Time* parameters set to the appropriate values.

ClientLib

`virtual CWhiskerTask::IncomingInfo()` is called – if your task overrides this function, the override will be called with the *msg* and *lTime* parameters set to the appropriate values.

Revision history

Implemented in WhiskerServer version 1.

Example

Info: You are client number 4

8.8.2.3 KeyEvent

Message

KeyEvent: *<key>* **on|off** *<document> [<timestamp>]*

Originator

Server

Details

Notifies the client that a key has been pressed or released on the Server's keyboard (if a document for which a DisplayKeyboardEvents command has been received has the input focus on the server). See the section on DisplayKeyboardEvents.

SDK Control

WhiskerKeyEvent fires (*Visual Basic: calls Control_WhiskerKeyEvent Sub*), with the *VKey*, *KeyDown*, *Document* and *Time* parameters set to the appropriate values.

ClientLib

`virtual CWhiskerTask::IncomingKeyEvent()` is called – if your task overrides this function, the override will be called with the *iKey*, *bDown*, *strDocument* and *lTime* parameters set to the appropriate values.

Revision history

Implemented in WhiskerServer version 2.

Example

KeyEvent: 68 on fish

See also

- DisplayKeyboardEvents

8.8.2.4 SyntaxError

Message

SyntaxError: *<report> [<timestamp>]*

Originator

Server

Details

This will be generated in response to unrecognized and badly-formed commands.

Note that while most commands respond with Success or Failure on the immediate socket when their parameters are improperly given, unrecognized commands generate the usual 'SyntaxError: Unrecognized command' message. This allows failed commands to be distinguished from nonexistent commands via the immediate socket.

SDK Control

WhiskerSyntaxError fires (*Visual Basic: calls Control_WhiskerSyntaxError Sub*), with the *ErrorMessage* and *Time* parameters set to the appropriate values.

ClientLib

`virtual CWhiskerTask::IncomingSyntaxError()` is called – if your task overrides this function, the override will be called with the *strMessage* and *lTime* parameters set to the appropriate values.

Revision history

Implemented in WhiskerServer version 1.

Examples

SyntaxError: Unrecognized command
SyntaxError: insufficient parameters to LineClaim
SyntaxError: invalid parameters to DisplayCreateDevice

8.8.2.5 Error

Message

Error: *<report> [<timestamp>]*

Originator

Server

Details

This will be generated (on the main socket) in response to commands that fail. In general if you receive a SyntaxError, there is probably a programming fault – the command did not make sense to the server. If you receive an Error, the command made sense but did not succeed.

SDK Control

WhiskerError fires (*Visual Basic: calls Control_WhiskerError Sub*), with the *ErrorMessage* and *Time* parameters set to the appropriate values.

ClientLib

`virtual CWhiskerTask::IncomingError()` is called – if your task overrides this function, the override will be called with the *strMessage* and *lTime* parameters set to the appropriate values.

Revision history

Implemented in WhiskerServer version 1.

Example

Error: LineReadState attempted on a line you do not own.

8.8.2.6 Fault

Message

Fault: *<report> [<timestamp>]*

Originator

Server

Details

Not implemented in the current versions of WhiskerControl; included in the command set for future expansion.

SDK Control

WhiskerFault fires (*Visual Basic: calls Control_WhiskerFault Sub*), with the *ErrorMessage* and *Time* parameters set to the appropriate values.

ClientLib

`virtual CWhiskerTask::IncomingFault()` is called – if your task overrides this function, the override will be called with the *strMessage* and *lTime* parameters set to the appropriate values.

Revision history

Implemented in WhiskerServer version 1.

Example

Fault: some_sort_of_fault_message

8.8.2.7 Warning

Message

Warning: *<report>*

Originator

Server

Details

A warning message is sent when a command (recevied on either socket) is either useless or unnecessary. The difference between an error and a warning is that the server successfully completes the useless command when a warning is issued (but has failed to do so when an error is generated). For example if you had created a display object was called 'mydoc' , attempts to add an object to 'mydoct' (misspelling) would generate an error (critical failure), whereas an attempt to delete 'mydoct' would generate a warning.

SDK Control

WhiskerWarning fires (*Visual Basic: calls Control_WhiskerWarning Sub*), with the *ErrorMessage* and *Time* parameters set to the appropriate values.

ClientLib

`virtual CWhiskerTask::IncomingWarning()` is called – if your task overrides this function, the override will be called with the *strMessage* and *lTime* parameters set to the appropriate values.

Revision history

Implemented in WhiskerServer version 1.

Examples

Warning: LineReadState called with multiline alias "XXX": only one state returned!
Warning: AnalogueReadState called with multiline alias "XXX": only one state returned!
Warning: AnalogueReadConfig called with multiline alias "XXX": only one config returned!
Warning: No line events to clear called "XXX"!
Warning: No timer events to clear called "XXX"!
Warning: No client-owned displays to delete (called "XXX")!
Warning: No display documents to delete (called "XXX")!
Warning: No sound buffer called "XXX" found on "YYY" by "ZZZ"!
Warning: Sending a client-client message when you cannot receive a reply!
Warning: Sending a client-client message to yourself!
Warning: No listening clients receving broadcast message!
Warning: Event generated from a pegged line
Warning: Attempting to set the state of a pegged output, won't do anything

8.8.2.8 Ping

Message

Ping

Originator

Server

Details

The client should respond with PingAcknowledged. This is an optional facility, but if your client supports this it makes it much easier for the user to determine whether or not the client has crashed (by whether or

not it responds to the Ping command).

SDK (Control or ClientLib)

The SDK will handle Ping messages automatically, keeping the Server informed that the Client is alive. These messages are not passed on to the client code.

Revision history

Implemented in WhiskerServer version 1.

See also

- TestNetLatency

8.8.2.9 ClientMessage

Message

ClientMessage: *<fromclientnum> <message>*

Originator

Server

Details

Indicates that a message has arrived from another client, namely client number *fromclientnum*.

Just as for other unpredictable messages (such as Event), this message will never arrive via the immediate socket.

Example

ClientMessage: 7 motor_cortex_response_detected_please_give_reward

Revision history

Implemented in WhiskerServer version 2.0.

See also

- Client-client communications

8.8.3 Commands sent by the client

These are the backbone of the Whisker system. They are issued by the client sending the relevant formatted message to the WhiskerServer, which will respond with information about the success of the command.

The commands all have SDK functions of the same name. Usage of these is often simpler than using the raw message, especially when the functions are requests for information. The SDK function declarations, along with the raw messages that are actually used to carry out the functions, are outlined below, with notes on their usage.

All SDK Control functions are ActiveX Methods or Properties, and are called in the standard way, using the period (full stop) operator, after the SDK control's name. For example, in **Visual Basic,** for a Control named Whisker, the following would be possible:

```
result = Whisker.TimerSetEvent("tick",1000,-1)
Whisker.TimerSetEvent "tick", 1000, -1
Call Whisker.TimerSetEvent("tick",1000,-1)
```

For the **C++** Client Library, all commands are declared in WhiskerTask.h, as public members of CWhiskerTask.

8.8.4 Digital I/O devices

Digital I/O devices, or **lines,** are controlled using the following commands:

- LineClaim
- LineSetState
- LineReadState
- LineSetEvent
- LineSetAlias
- LineSetSafetyTimer
- LineClearSafetyTimer
- LineClearEvent
- LineClearEventByLine
- LineClearAllEvents
- LineRelinquishAll

8.8.4.1 LineClaim

Message

LineClaim <*linenumber*> **[-input|-output] [-ResetOn|-ResetOff|-Leave] [–alias** <*alias*>**]**
LineClaim <devicegroup> <devicename> **[-input|-output] [-ResetOn|-ResetOff|-Leave] [–alias** <*alias*>**]**

Originator

Client

Response

Info: ClaimAccepted: Line <*linenumber*> For input|output As <*alias*>
Error: ClaimRejected: Line <*linenumber*> <*reason*>
SyntaxError: insufficient parameters to LineClaim
SyntaxError: invalid parameters to LineClaim

For example,

Error: ClaimRejected: Line 112 is a non-existent line
Error: ClaimRejected: Line 5 is already claimed
Error: ClaimRejected: Line 7 is not an input line

If you set an alias, you will also get messages concerning the success or failure of the alias command. You may also get a message of the form

Info: ClaimAccepted: Line 5 For Input As 5 (ALIAS NOT SET)

if the claim succeeded but the alias was improperly formed.

Response (immediate socket)

Success
Failure

If you request an alias, you will only get Success if the alias command also succeeds.

Details

Claims hardware line *linenumber* for input or output. By default, or if 'reset' is specified, output lines are turned off. If 'leave' is specified, the line is left in its current state. This parameter also determines what happens if the client falls off the edge of the earth. **Lines are numbered from 0,** so a typical single-board system has lines 0–71.

Input | output. Whether a line is an input or an output line is *not* determined by the input/output parameter, but by the configuration of the server (which you can set, on the server, depending on how you wire up the apparatus). This parameter is merely a statement of your intent. You do not have to specify 'input' or 'output' when you claim the line. However, it is advisable. If you claim an input line and subsequently try to set its state, you will get an error message. It is better to state when you claim the line that you think it is an output line; if you are wrong, then the error message comes immediately and you know things aren't safe to proceed.

ResetOn | ResetOff | Leave. If you specify 'ResetOn' or 'ResetOff' and the line is an output line, it will be turned on/off immediately, *and when the client gives up the line*. Otherwise it will be left in its current state, whatever that is. The default is 'Leave'. (Note that it is only possible to set the reset state when you claim a line, and not later; this is deliberate, as it is meant as a safety feature and not to be fiddled around with willy-nilly.) This feature is currently not included in the SDK control version of LineClaim. If it is important to you, it can be achieved by using the SendToServer command, e.g.

```
Whisker.SendToServer "LineClaim box1 IVPump -output -alias pump -ResetOff"
```

Server device names. If you use the second form of the command, the line specified by the combination of the *devicegroup* and *devicename* parameters is claimed.

SDK Control Method

```
Function Whisker.LineClaim(GroupName As String, DeviceName As String,
ServerLineNumber As Integer, LineName As String, IsInput As Boolean)
```

Attempts to claim the line given in the DDF as *DeviceName* in Group *GroupName*. If either of these is an empty string, attempts to claim line number *ServerLineNumber*. Once claimed, it asks for the line to be referred to by the alias *LineName* (see Details above). *IsInput* should be True if the Client believes the line to be an Input, or response line.
Evaluates to True if the claim succeeds.

ClientLib

```
bool LineClaim(int iLine, bool bOutput, enum _whisker_reset_state reset, const
CString& strAlias);
bool LineClaim(const CString& strDeviceGroup, const CString& strDeviceName, bool
bOutput, enum _whisker_reset_state reset = WHISKER_RESET_LEAVE, const CString&
```

```
strAlias = "");
```

Attempts to claim line number *iLine* (first overloaded syntax), or the line described in the DDF by *strDeviceName* and *strDeviceGroup* (second syntax), with the I/O function reset state described by *bOutput* and *reset*. If *alias* is no a zero-length string, attempts to set this as the claimed line's alias. Returns true for success, false for failure.

Revision history

Implemented in WhiskerServer version 1 as ClaimLine.
Renamed in WhiskerServer version 2.3.

See also

- Digital I/O devices
- Device names and aliases
- LineSetAlias
- LineRelinquishAll

8.8.4.2 LineSetState

Message

LineSetState <*linenumber*>|<*alias*> **on|off**

Originator

Client

Response (main socket)

A successful call generates no response. Otherwise an error message will be generated:

Error: can't set line *lineNumber*
SyntaxError: insufficient parameters to LineSetState
SyntaxError: invalid parameters to LineSetState
SyntaxError: invalid line to LineSetState

Response (immediate socket)

Success
Failure

Details

Turns output lines on/off. The LineSetState command supports **grouping.** That is, you can do this:

```
LineClaim 4 –alias Houselight
LineClaim 5 –alias LeftStimLight
LineClaim 6 –alias RightStimLight
LineSetAlias LeftStimLight LIGHTGROUP
LineSetAlias RightStimLight LIGHTGROUP
LineSetState LIGHTGROUP on
                    –– This will turn lines 5 and 6 on.
LineSetState LeftStimLight off; LineSetState RightStimLight off
                    –– Remember that multiple commands can be given in one line.
```

If a group is set, multiple error messages may be generated on the main socket, but only a single value will return on the immediate socket – this value will be Success if all lines were set correctly, and Failure if a single line could not be set.

From WhiskerServer v2.9.04, **if you attempt to set the state of a pegged output, a Warning is also sent.**

SDK Control

```
Function LineSetState(MyLine As String, NewState As wsLineState) As Boolean
```

Sets the state of digital I/O line *MyLine* to *NewState*. Evaluates to True if request succeeds, otherwise False.

A simpler property format can also be used to set a line state

```
Property LineState(MyLine As String) As wsLineState
```

Example VB syntax:

```
'To check if it succeeds
Dim bSetLineOK As Boolean
bSetLineOK = Whisker.LineSetState(MyLine,NewState)
'simpler syntax would be
Whisker.LineState(MyLine) = NewState
```

Notes

NewState can be either wsOn (defined as 1) or wsOff (defined as 0). You can also use the value -1 (`True` in VisualBasic), which is interpreted as wsOn. Attempting to set the state of a line to any other value will fail, and generate a WhiskerError message.

ClientLib

```
bool LineSetState(const CString& strLine, bool bState);
```

Attempts to set the state of *strLine* to *bState*. Returns true for success, false for failure.

Revision history

Implemented in WhiskerServer version 1 as SetState.
Renamed in WhiskerServer version 2.3.

See also

- LineReadState
- Controlling groups of lines
- Digital I/O devices

8.8.4.3 LineReadState

Message

LineReadState <*linenumber*>|<*alias*>

Originator

Client

Response

Info: State: <*linenumber*> (<*alias*>) on|off
SyntaxError: invalid parameters to LineReadState
SyntaxError: insufficient parameters to LineReadState
Error: no such line or alias
Error: LineReadState cannot read line <*linenumber*>

Response (immediate socket)

on
off
error

Details

Asks the server what the current status of the line is (on or off). Applies to input and output lines.

From WhiskerServer v2.11, hardware is supported that can only detect ON transitions, not OFF transitions (e.g. the Lafayette/ABET hardware); consequently, Whisker can only respond to ON events on such lines, and if you attempt to read the actual state of the line (which Whisker has not been informed of by the external hardware and cannot be aware of), an error is generated.

Advanced point: if a group alias is given on the main socket, the server will respond with a State message for each applicable line. If a group alias is given on the immediate socket, the server will only return the state of one line, chosen (effectively) at random. This situation will generate a Warning: message. I would discourage you from using group aliases with LineReadState.

SDK Control

```
Function LineReadState(MyLine As String) As wsLineState
Property LineState(MyLine As String) As wsLineState
```

Read the state of digital I/O line *MyLine*. Returns `wsOn` (1) if the line is On, `wsOff` (0) if it is Off, or `wsUnknown` (2) if the line cannot be read (e.g. if the control is not connected to a WhiskerServer)

Example VB syntax:

```
'To check if it succeeds
Dim CurrentState As wsLineState
CurrentState = Whisker.LineReadState(MyLine)
'Using alternative syntax:
If(Whisker.LineState(MyLine) = wsOn) Then
       'line is on
       Call MySub
End If
```

Remarks

The VB values `True` (-1) and `False` (0) can be used as synonyms for wsOn and wsOff when turning lines on with LineSetState, but the values returned by LineReadState cannot simply be treated as if they were `Boolean` values (`True` or `False`) .

For example, in VisualBasic the operator `Not` can be used to turn `True` into `False`, and vice versa. This cannot be used in the same way with wsLineState values:

```
'Example
'We wish to make sure 'Light2' is off if 'Light1' is on, and vice versa

'The following code WILL WORK in the intended way
Whisker.LineState(Light2) =  Not (Whisker.LineState(Light1) = wsOn)
'This DOES NOT work and will cause an error if Light1 is wsOn (=1)
Whisker.LineState(Light2) =  Not Whisker.LineState(Light1)
```

ClientLib

```
bool LineReadState(const CString& strLine, bool& bState);
```

Sets bState to true if *strLine* is currently on, false if it is off. Returns true for success, false for failure.

Revision history

Implemented in WhiskerServer version 1 as ReadState.
Renamed in WhiskerServer version 2.3.

See also

- LineSetState
- Digital I/O devices

8.8.4.4 LineSetEvent

Message

LineSetEvent *<linenumber>|<alias>* **on|off|both** *<eventtext>*

Originator

Client

Response (main socket)

Info: EventCreated: Line *linenumber* on|off|both
Error: EventRefused: Line *reason for refusal*
SyntaxError: insufficient parameters to LineSetEvent
SyntaxError: invalid parameters to LineSetEvent

Response (immediate socket)

Success
Failure

Details

You can have separate events for on and off. It is not sensible to request both an 'on' and an 'off' event for a device such as a lever (you would get two events for each press), but could sometimes be useful (e.g. to determine the proportion of time spent on a certain perch).

If a grouped alias is used, multiple messages may be generated on the main socket, but only a single value will return on the immediate socket (Success if all line events were set correctly, and Failure if there was a problem with any one line). However, it is not recommended that you use grouped lines with this

command.

Note that line events can be requested from input *or* output lines.

From WhiskerServer v2.9.04, **if a pegged input generates an event, a Warning is also sent.**

From WhiskerServer v2.11, hardware is supported that can only detect ON transitions, not OFF transitions (e.g. the Lafayette/ABET hardware); if you attempt to set an OFF event on such a line, an error is generated. If you attempt to set BOTH events, the ON event succeeds (and the overall command succeeds) but a warning is generated about the OFF event failing to be set.

Revision history

Implemented in WhiskerServer version 1 as RequestLineEvent.
Renamed in WhiskerServer version 2.3.

See also

- Digital I/O devices
- Event
- LineClearEvent

8.8.4.5 LineSetAlias

Message

LineSetAlias <*line_number***>|<***existing_alias***> <***new_alias***>**

Originator

Client

Response

No response (implies success) or an error message:

SyntaxError: insufficient parameters to LineSetAlias
SyntaxError: invalid parameters to LineSetAlias
Error: invalid line or alias

Response (immediate socket)

Success
Failure

Details

You can define aliases for lines. If you issue the command 'LineSetAlias 5 leftlever', you can then use 'leftlever' in place of the line number in subsequent commands (meaning you can do 'LineClaim leftlever', 'LineSetEvent leftlever on LEFT_LEVER_PRESSED').

Restrictions: there can be no spaces in the alias. Aliases cannot begin with a number (to distinguish them from ordinary line numbers). It is not case-sensitive.

Aliases can be used with any command that takes a line number, except LineClaim (for which you obviously have to know the actual line number!). **Note:** you can claim and alias a line in one step (see LineClaim).

Aliases are designed to make programs independent of the box (chamber) number. Once you've defined your houselight (which might be line 5 in box 1, or line 28 in box 3) you shouldn't need to know its actual number again. Aliases also help to simplify counterbalancing and task design. For example,

```
LineClaim 8 –alias LeftLever
LineClaim 9 –alias RightLever
...
        L/R counterbalancing determined by client program at this point
...
LineSetAlias LeftLever ACTIVELEVER
LineSetAlias RightLever INACTIVELEVER
LineSetEvent ACTIVELEVER on active_lever_pressed
LineSetEvent ACTIVELEVER off active_lever_released
LineSetEvent INACTIVELEVER on inactive_lever_pressed
LineSetEvent INACTIVELEVER off inactive_lever_released
```

Before you ask, there is no need to set aliases for timers, as timers have names anyway and you define the names. Aliases for timers are not supported.

Revision history

Implemented in WhiskerServer version 1 as SetLineAlias.
Renamed in WhiskerServer version 2.3.

See also

- Controlling groups of lines
- Digital I/O devices
- LineClaim

8.8.4.6 LineSetSafetyTimer

Message

LineSetSafetyTimer <*linenumber*>|<*alias*> <*time*> **on|off**

Originator

Client

Response

Info: line <*line*> will be switched off after <*time*> ms of inactivity
SyntaxError: something wrong with SafetyTimer call
Error: invalid line <*linenumber*>
Error: you do not own line <*linenumber*>
Error: line <*linenumber*> is not an output line

Response (immediate socket)

Success
Failure

Example

SafetyTimer PUMP 5000 off

This tells the server to ensure that PUMP is always switched off if its state has not been set by the client in the last 5 s. (The timer continually compares the current time with a record of when the line last had its state set, or was claimed by a client.)

If the safety timer is triggered, a warning message will be sent to the client, as in the following example:

Warning: line 6 switched OFF after 5000 ms of inactivity

Group aliases may be used, though that seems to me like an odd thing to do.

Important note: this feature only operates while the server thinks the client is connected. It provides some protection against a mis-programmed client (including a client that crashes but leaves the link to the server open) or a network failure. If the device is really critical, like an i.v. pump, you should also specify a *reset state* when you claim the line; for example,

LineClaim 5 output ResetOff –alias PUMP

In this situation, when the client releases the line or client-server communication is lost, the line will be reset to the off state. Use both these commands for maximum safety.

Revision history

Implemented in WhiskerServer version 1 as SetSafetyTimer.
Renamed in WhiskerServer version 2.3.

See also

- Digital I/O devices
- LineClearSafetyTimer

8.8.4.7 LineClearSafetyTimer

Message

LineClearSafetyTimer <*linenumber***>|<***alias***>**

Originator

Client

Response

Error: invalid line <*line*>
Error: you do not own line <*line*>
Error: line <*line*> is not an output line
Info: safety timer cleared from line <*line*>
SyntaxError: invalid parameters to LineClearSafetyTimer
SyntaxError: insufficient parameters to LineClearSafetyTimer

Response (immediate socket)

Success
Failure

Revision history

Implemented in WhiskerServer version 2.3 (approx).

See also

- Digital I/O devices
- LineSetSafetyTimer

8.8.4.8 LineClearEvent

Message

LineClearEvent *<eventname>*

Originator

Client

Response

Info: killed all line events called **<***eventname***>**
Error: no line events to kill called **<***eventname***>**
SyntaxError: insufficient parameters to LineClearEvent
SyntaxError: invalid parameters to LineClearEvent

Response (immediate socket)

Success
Failure

Details

Kills a line event, or events, by the message associated with them. May kill multiple events. Does not kill timers.

Revision history

Implemented in WhiskerServer version 1 as KillEvent.
Renamed in WhiskerServer version 2.3.

See also

- Digital I/O devices
- LineClearEventByLine
- LineClearAllEvents
- LineSetEvent

8.8.4.9 LineClearEventByLine

Message

LineClearEventByLine <*linenumber***>|<***alias***> on|off|both|all**

Originator

Client

Response

SyntaxError: insufficient parameters to LineClearEventByLine
SyntaxError: invalid parameters to LineClearEventByLine
Error: invalid line <*linenumber*>
Error: you do not own line <*linenumber*>
Info: killed all events on line <*linenumber*>

Response (immediate socket)

Success
Failure

Details

Kills events on the specified line. The default is to kill all events, but you can selectively kill 'on' or 'off' events. 'Both' and 'all' do the same thing.

Revision history

Implemented in WhiskerServer version 1 as KillEventByLine.
Renamed in WhiskerServer version 2.3.

See also

- Digital I/O devices
- LineClearEvent
- LineClearAllEvents
- LineSetEvent

8.8.4.10 LineClearAllEvents

Message

LineClearAllEvents

Originator

Client

Response

Info: All line events killed
SyntaxError: invalid parameters to LineClearAllEvents

Response (immediate socket)

Success

Details

Kills all line events. Does not affect timers.

Revision history

Implemented in WhiskerServer version 1 as KillAllEvents.
Renamed in WhiskerServer version 2.3.

See also

- Digital I/O devices
- LineClearEvent
- LineClearEventByLine
- LineSetEvent

8.8.4.11 LineRelinquishAll

Message

LineRelinquishAll

Originator

Client

Response

Info: All lines relinquished
SyntaxError: invalid parameters to LineRelinquishAll

Response (immediate socket)

Success

Details

Relinquishes control of all lines owned by the client. Also deletes all aliases.

Revision history

Implemented in WhiskerServer version 1 as RelinquishAllLines.
Renamed in WhiskerServer version 2.3.

See also

- Digital I/O devices
- LineClaim

8.8.5 Controlling groups of lines

The LineSetAlias and LineSetState commands supports **grouping.** That is, two lines can be given common (as well as distinct) aliases:

```
LineClaim 4 –alias Houselight
LineClaim 5
LineSetAlias 5 LeftStimulusLight
      –– This illustrates two ways to set an alias
LineClaim 6 –alias RightStimulusLight
```

```
LineSetAlias LeftStimulusLight LIGHTGROUP
LineSetAlias RightStimulusLight LIGHTGROUP
LineSetState 4 on
LineSetState LIGHTGROUP on
      -- This will turn lines 5 and 6 on.
LineSetState LeftStimulusLight off
      -- This will turn line 5 off.
LineSetAlias LIGHTGROUP ANOTHERALIAS
LineSetState ANOTHERALIAS off
      -- You can add another alias for a whole group. This will turn lines 5 and 6
off.
```

The grouping function makes it very easy to implement **yoking.** For example, if your task currently looks like this:

```
LineClaim 4 -alias Houselight
LineClaim 5 -alias LeftStimulusLight
LineClaim 6 -alias RightStimulusLight
                  (*)
LineSetState Houselight on
LineSetState LeftStimulusLight on
```

and you want another box to be yoked to it (which has lines 15–17 for these same devices), you can add the following lines at or before the point marked (*):

```
LineClaim 15 -alias Houselight
LineClaim 16 -alias LeftStimulusLight
LineClaim 17 -alias RightStimulusLight
```

and the main program will happily control the devices as before, except when it switches the houselight on in the main box, it will also switch it on in the yoked box.

Though you can use group aliases with LineSetEvent, LineClearEventByLine (described later) and LineReadState, I think this is unwise. In particular, this practice may lead to confusing results with LineReadState: if you issue a LineReadState command on the main socket, the server will respond with the status of all applicable lines, but if you do the same thing on the immediate socket – where a single instant response is the only reply you ever get – the server will provide the status of just one of the applicable lines (chosen at random).

See also

- ClaimGroup
- Digital I/O devices

8.8.6 Timer devices

Timers are simple software devices which are used to generate events after pre-specified intervals.

They are controlled with the following commands:

- TimerSetEvent

- TimerClearEvent
- TimerClearAllEvents

8.8.6.1 TimerSetEvent

Message

TimerSetEvent *<duration> <reloadcount> <message>*

Originator

Client

Response (immediate socket)

Success
Failure

Response (main socket)

Info: TimerCreated: duration *<duration>* reloadcount *<reloadcount>*
Error: TimerRefused: *<reason>*
SyntaxError: invalid parameters to TimerSetEvent
SyntaxError: insufficient parameters to TimerSetEvent

Details

You do not specify the duration with which the timer is reloaded, but the number of times that it is reloaded with its original duration. (For those of you used to Arachnid, this represents a significant change in the way timers are used.) The reload value –1 has a special significance: it means that the timer should be reloaded for eternity (i.e. until you manually kill it). Use 0 for a one-off timer, 1 for a timer that is reloaded once (i.e. executes twice in total), and so on. In fact, all negative numbers are treated the same way as –1.

You may request a zero-duration timer; it'll be activated as soon as the server next polls the set of active timers.

SDK Control Method

```
TimerSetEvent(Event as String, msDelay as long, Repetitions as integer) as Boolean
```

Requests event message *Event*, after a delay of *msDelay* ms, subsequently repeated *Repetitions* times. Function will evaluate to True for success, False for failure.

ClientLib

```
bool TimerSetEvent(const CString& strEvent, unsigned long timeInMs, int
iRepetitions);
```

Requests event message *strEvent*, after a delay of *timeInMS* ms, subsequently repeated *iRepetitions* times. Returns true for success, false for failure.

Revision history

Implemented in WhiskerServer version 1 as RequestTimerEvent.
Renamed in WhiskerServer version 2.3.

See also

- Timers
- Event
- TimerClearEvent
- TimerClearAllEvents

8.8.6.2 TimerClearEvent

Message

TimerClearEvent *<timername>*

Originator

Client

Response (immediate socket)

Success
Failure

Response

Info: killed all timers called *<eventname>*
Info: no timers to kill called *<eventname>*
SyntaxError: invalid parameters to TimerClearEvent
SyntaxError: insufficient parameters to TimerClearEvent

Details

I hope this is self-explanatory!

SDK Control Method

```
TimerClearEvent(Timer as String)
```

Clears any scheduled timer called *Timer*.

ClientLib

```
void TimerClearEvent(const CString& strTimer);
```

Clears any scheduled timer called *strTimer*.

Revision history

Implemented in WhiskerServer version 1 as KillTimer.
Renamed in WhiskerServer version 2.3.
"No timers to kill" message changed from Warning to Info, March 2004 (server v2.8.03), since it's common practice to delete a timer that may or may not have fired already.

See also

- Timers
- TimerSetEvent
- TimerClearAllEvents

8.8.6.3 TimerClearAllEvents

Message

TimerClearAllEvents

Originator

Client

Response (immediate socket)

Success
Failure

Response

Info: All timers killed
SyntaxError: invalid parameters to TimerClearAllEvents

Details

Kills all timers. Does not affect line events, or display object events.

SDK Control Method

```
TimerClearAllEvents()
```

Clears all scheduled timers.

ClientLib

```
void TimerClearEvent();
```

Clears any scheduled timers.

Revision history

Implemented in WhiskerServer version 1 as KillAllTimers.
Renamed in WhiskerServer version 2.3.

See also

- Timers
- TimerSetEvent
- TimerClearEvent

8.8.7 Audio devices

Note: differences between editions of WhiskerServer

If you have the Basic Edition of Whisker, all audio commands generate the following error from the server:

```
Error: command not supported by this version of WhiskerServer
```

or return `Failure` on the immediate socket.

How the server searches for media files

When the client passes a filename to the server, the server first tries to find it by appending the filename to the server's media path. (This path is configured on the server and stored in the registry.) If the server's media path is, for example, 'd:\whiskermedia', the client can then request playback of 'myclient\ping.wav' and the server will first attempt to find it as as 'd:\whiskermedia\myclient\ping.wav'.

If this attempt fails, the server will then try the filename as an *absolute* filename (so the client can pass, for example, 'd:\myclient\wavs\ping.wav').

Development note: these checks are performed by CWhiskerServer::MediaFilename(), not CSoundDevice.

Specifying filenames to the server

Whenever a media filename is passed to the server, it is the *last parameter passed*, and it *may* include spaces.

How the sound system works internally

The server detects a number of physical audio devices, and assigns one or more *logical* audio devices to each. (Normally there will be one logical device per physical device, but if you choose to use the left and right stereo channels as separate devices, you will get two logical devices per physical device.) The clients only see logical devices.

The client can claim control of logical devices and assign them aliases. (Clients always have exclusive access to logical devices that they have claimed.)

Each logical device may have a number of *sounds* associated with it. You can load WAV files or simple tones as sounds, giving them a name, and play them.

Audio devices may share aliases. For example, if sound device 0 has the aliases *left* and *stereo*, and sound device 1 has the aliases *right* and *stereo*, you can issue the command *AudioLoadFile stereo explosion explosion.wav; AudioPlaySound stereo explosion* and the sound will be played through devices 0 and 1. But the AudioLoadFile call created a sound named 'explosion' for both audio devices, so you can now also say *AudioPlaySound left explosion* and only device 0 will play.

Sounds may also share names – but this will lead to sounds being mixed together. You may issue the command *AudioLoadFile left bignoise gunfire.wav; AudioLoadFile left bignoise crowscreech.wav; AudioPlaySound left bignoise* and the guns and the crow will both make their voices heard. These commands load two sounds into one sound device but both are given the same name. (You may substitute *stereo* for *left* in this example and a veritable cacophony will arise.)

The server uses the DirectX (DirectSound) interface to control sound devices; therefore, DirectX must be installed to obtain sound support. (The devices are set to have *global focus*; that is, no matter which Windows program has the input focus, the sound is audible.)

Audio commands

Audio devices are controlled using the following commands:

- AudioClaim
- AudioSetAlias
- AudioRelinquishAll
- AudioPlayFile
- AudioLoadSound
- AudioLoadTone
- AudioPlaySound
- AudioUnloadSound
- AudioStopSound
- AudioSetSoundVolume
- AudioSilenceDevice
- AudioUnloadAll

8.8.7.1 AudioClaim

Message

AudioClaim <*audiodevicenumber*> **[–alias** <*alias*>**]**
AudioClaim <*devicegroup*> <*devicename*> **[–alias** <*alias*>**]**

Originator

Client

Details

Suppose there are four physical stereo sound cards in the server. By default, the server is configured to allocate these as audio devices 0–3. However, each sound card may be split to isolate the left and right stereo channels, in order to double the number of available sound devices. In this case, the audio devices would be numbered 0–7, where device 0 is the left channel of the first sound card, 1 is the right channel of the first card, 2 the left channel of the second card, etc.

Note that this is a *server-side* feature, configured on the server console.

If you need one client wants to address the left and right channels of a single sound card independently, you can either split the sound cards and have the client control two notional audio devices (left and right), or leave them as stereo devices and allow the client to provide a stereo sound source.

As with lines, you can give the audio device an alias as you claim it.

Server device names. If you use the second form of the command, the device specified by the combination of the *devicegroup* and *devicename* parameters is claimed.

Response

SyntaxError: insufficient parameters to AudioClaim
SyntaxError: invalid parameters to AudioClaim
ClaimAccepted: <*audiodevicenumber*>
ClaimRejected: <*audiodevicenumber*> <*reason*>
ClaimRejected: group/device <*devicegroup*>/<*devicenumber*> not recognized

For example,

ClaimRejected: 112 is a non-existent audio device
ClaimRejected: 5 is already claimed

Response (immediate socket)

Either of

Success
Failure

If you request an alias, you will only get Success if the alias command also succeeds.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- Device names and aliases
- AudioSetAlias
- AudioRelinquishAll

8.8.7.2 AudioSetAlias

Message

AudioSetAlias *<audiodevicenumber>|<existing_alias> <new_alias>*

Originator

Client

Response

SyntaxError: insufficient parameters to AudioSetAlias
SyntaxError: invalid parameters to AudioSetAlias
Info: audio device *<devicenumber>* is using the alias *<alias>*
Error: device number is not a valid number or pre-existing alias
Error: can't set alias for a device you don't own
Error: alias is blank
Error: can't have spaces in an alias
Error: aliases can't begin with a digit

Response (immediate socket)

Success
Failure

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioClaim

8.8.7.3 AudioRelinquishAll

Message

AudioRelinquishAll

Originator

Client

Response

Info: All audio devices relinquished
SyntaxError: invalid parameters to AudioRelinquishAll

Response (immediate socket)

Success

Details

Relinquishes control of all audio devices owned by the client. Also deletes all audio sounds and aliases.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioClaim

8.8.7.4 AudioPlayFile

Message

AudioPlayFile *<audiodevicenumber>|<alias> <filename>*

Originator

Client

Response

SyntaxError: insufficient parameters to AudioPlayFile
SyntaxError: invalid parameters to AudioPlayFile

Response (immediate socket)

Success
Failure

Details

Loads a .WAV file, plays it on the chosen audio device, then forgets about it. The sound cannot be stopped while it is playing. (Use AudioLoadFile / AudioPlaySound / AudioStopSound / AudioUnloadSound for greater control over the process.)

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioLoadSound
- AudioPlaySound

8.8.7.5 AudioLoadSound

Message

AudioLoadSound *<audiodevicenumber>|<alias> <soundname> <filename>*

Originator

Client

Response

SyntaxError: insufficient parameters to AudioLoadSound
SyntaxError: invalid parameters to AudioLoadSound
Error: can't find WAV file
Error: invalid WAV file
Error: no such audio device
Info: loaded WAV file as sound *<soundname>*

Response (immediate socket)

Success
Failure

Details

Loads a WAV file into a sound and gives that sound a name, but does not play the sound. (Use PlaySound to play the sound.)

The sound is loaded into a *specific* sound device. If you need to play the same sound on several audio devices, you must load it into a separate sound for each device, or create an alias that refers to sound devices.

If you wish, you may give several sounds (on one or more sound devices) the same name, though this is not recommended. However, this feature does allow you to play them all simultaneously with a single PlaySound command.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioPlaySound
- AudioPlayFile
- AudioLoadTone
- AudioUnloadSound
- AudioGetSoundLength

8.8.7.6 AudioLoadTone

Message

AudioLoadTone *<audiodevicenumber>*|*<alias>* *<soundname>* *<frequency>* **square|sine|sawtooth|tone** [<duration_ms>]

Originator

Client

Response

SyntaxError: insufficient parameters to AudioLoadTone
SyntaxError: invalid parameters to AudioLoadTone
Error: failed to load Tone on Audio *<audiodevicenumber>* (*<alias>*)
Info: loaded Tone on Audio *<audiodevicenumber>* (*<alias>*)

Response (immediate socket)

Success
Failure

Details

Loads (creates) a simple tone of given frequency (Hz) and gives the sound a name; the sound can be used in exactly the same way as a WAV sound loaded with AudioLoadSound.

The waveform of the tone can be chosen from:

- Sine – gives a pure sinusoidal form
- Square – gives a square wave
- Sawtooth – gives an asymmetric (|/|/|/|/|/) form
- Tone – a modified waveform similar to the sinusoid, but containing more energy; this results in a better sound with a range of frequencies.

The optional **<duration_ms>** parameter sets the duration in milliseconds (default: 1000 ms, or 1 s).

To play the sound for long periods, consider setting a short sound to repeat (AudioPlaySound -loop) and cancel it after the desired time.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioPlaySound

- AudioLoadSound
- AudioGetSoundLength

8.8.7.7 AudioPlaySound

Message

AudioPlaySound <*audiodevicenumber*>|<*alias*> <*soundname*> **[-loop]**

Originator

Client

Response

Error: no such audio device
SyntaxError: insufficient parameters to AudioPlaySound
SyntaxError: invalid parameters to AudioPlaySound
Error: no such audio sound
Info: playing sound(s) <*soundname*> on device <*audiodevicenum*>
Info: audio sounds playing
Error: no such audio sound, or sound problem

Response (immediate socket)

Success
Failure

Details

Plays a pre-loaded sound on the audio device it is attached to, starting at the beginning of the sound. If –loop is requested, the sound will play continuously until stopped by AudioStopSound, AudioSilenceDevice, AudioUnloadSound, AudioUnloadAll, AudioRelinquishAll, or ShutDown. If there are several sounds with this name, all are played.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioLoadSound
- AudioLoadTone
- AudioStopSound
- AudioSetSoundVolume
- AudioSilenceDevice

8.8.7.8 AudioUnloadSound

Message

AudioUnloadSound <*audiodevicenumber*>|<*alias*> <*soundname*>

Originator

Client

Response

Error: no such audio device
SyntaxError: insufficient parameters to AudioUnloadSound
SyntaxError: invalid parameters to AudioUnloadSound
Error: no such audio sound
Info: audio sound(s) deleted

Response (immediate socket)

Success
Failure

Details

Deletes an audio sound from the server's memory. (You do not need to specify the audio device name; all sounds with this name will be deleted.) If the sound is playing, it is stopped.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioUnloadAll

8.8.7.9 AudioStopSound

Message

AudioStopSound <*audiodevicenumber*>|<*alias*> <*soundname*>

Originator

Client

Response

Error: no such audio device
SyntaxError: insufficient parameters to AudioStopSound
SyntaxError: invalid parameters to AudioStopSound
Warning: No sound buffer called <*buffer*> found on <*device*> by AudioStopSound!
Info: audio sound(s) stopped

Response (immediate socket)

Success
Failure

Details

Stops playback of the selected sound. (The call still succeeds if the sound wasn't playing in the first place. If there are several sounds with this name, all will be stopped.) The sounds are not deleted; you can play them again with AudioPlaySound.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioPlaySound
- AudioUnloadSound

8.8.7.10 AudioGetSoundLength

Message

AudioGetSoundLength *<audiodevicenumber>|<alias> <soundname>*

Originator

Client

Response

Error: no such audio device
SyntaxError: insufficient parameters to AudioGetSoundLength
SyntaxError: invalid parameters to AudioGetSoundLength
Error: No sound buffer called *<buffer>* found on *<device>* by AudioGetSoundLength
Info: length is *<length>* ms

Response (immediate socket)

<length>
Failure

Details

Requests the length, in milliseconds, of the sound.

Revision history

Implemented in WhiskerServer version 2.6.07.

See also

- Audio devices
- AudioPlaySound
- AudioLoadSound
- AudioLoadTone

8.8.7.11 AudioSetSoundVolume

Message

AudioSetSoundVolume *<audiodevicenumber>|<alias> <soundname> <volume>*

Originator

Client

Response

Error: no such audio device
SyntaxError: insufficient parameters to AudioSetSoundVolume
SyntaxError: invalid parameters to AudioSetSoundVolume
Error: no such audio sound, or sound problem
Info: audio sound(s) stopped

Response (immediate socket)

Success
Failure

Details

The relative volume of the sound, in decibels (dB), with 100 being full volume (no attenuation). Therefore

```
AudioLoadSound speaker sound1 "C:\mysound.wav"
AudioLoadSound speaker sound2 "C:\mysound.wav"
AudioSetSoundVolume speaker sound1 95
```

would result in sound1 being 5dB quieter than sound 2.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioPlaySound

8.8.7.12 AudioSilenceDevice

Message

AudioSilenceDevice *<audiodevicenumber>|<alias>*

Originator

Client

Response

Error: no such audio sound
Info: audio device silenced
SyntaxError: insufficient parameters to AudioSilenceDevice

SyntaxError: invalid parameters to AudioSilenceDevice

Response (immediate socket)

Success
Failure

Details

Stops playback of all sounds on the selected device. Does not delete its sounds.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioPlaySound
- AudioSilenceAllDevices

8.8.7.13 AudioSilenceAllDevices

Message

AudioSilenceAllDevices

Originator

Client

Response

Info: all audio devices silenced
SyntaxError: invalid parameters to AudioSilenceDevice

Response (immediate socket)

Success
Failure

Details

Stops playback of all sounds on all audio devices. Does not delete the sounds.

Revision history

Implemented in WhiskerServer version 2.6.07.

See also

- Audio devices
- AudioPlaySound
- AudioSilenceDevice

8.8.7.14 AudioUnloadAll

Message

AudioUnloadAll *<audiodevicenumber>|<alias>*

Originator

Client

Response

Error: no such audio device
Error: can't delete all sounds on audio device *<device>*
Info: sounds deleted on audio device *<device>*
SyntaxError: insufficient parameters to UnloadAllSound
SyntaxError: invalid parameters to UnloadAllSounds

Response (immediate socket)

Success
Failure

Details

Deletes all sounds on the selected device, stopping playback in the process.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Audio devices
- AudioUnloadSound

8.8.8 Multimedia configuration

8.8.8.1 SetMediaDirectory

Message

SetMediaDirectory *<directory>*

Originator

Client

Response

Info: media directory set to *<directory>*
SyntaxError: insufficient parameters to SetMediaDirectory
SyntaxError: invalid parameters to SetMediaDirectory

Response (immediate socket)

Success

Failure

Details

Sets the directory that will be searched for the server for media files (.WAV wave audio files, .BMP bitmap pictures, etc.)

If the server is asked to load a file called file.bmp, it searches for it in the following order:

1. In the media directory specified by the client using SetMediaDirectory (interpreted as a raw pathname);
2. As above, but directory interpreted as relative to the server's default media directory (configured on the server console);
3. In the server's default media directory;
4. As a raw filename. (If a full path is given, e.g. c:\file.bmp, then the absolute path is used; if a simple filename, e.g. file.bmp, is given, then the server searches in the current working directory for its process, typically the directory that the server's .EXE file was started from.)

Note All pathnames are interpreted relative to the *server*, even if you are running a client on a different computer. It is possible to retrieve files from a different computer. For example, if a machine called client wants a server called server to use a picture on the client's hard disk, it could send a filename as "\\client\client_disk\directory\file.bmp". However, this method should not be unless you have good reason to store the media files remotely from the WhiskerServer machine: the performance of media loading commands will be slower and will depend heavily on network performance and reliability (especially if you need to load media quickly and often).

Revision history

Implemented by WhiskerServer version 2.3.

8.8.9 Display devices, touchscreens, and mouse events

Note: differences between versions of WhiskerServer

If you have the Basic Edition version of Whisker, all display/touchscreen/mouse commands generate the following error from the server:

```
Error: command not supported by this version of WhiskerServer
```

and return `Failure` on the immediate socket.

How the display system works internally

When the server runs, it detects physical display devices (monitors), including the primary display.

If a **physical display** device is to be used, the server creates a full-screen frame window for that device — that is, it takes full control of a monitor and places a window on that monitor to obscure any others. The server attaches a default document to this window; this default document consists of a black screen. If you choose to display a test pattern on a particular monitor, the server switches the window's view to a document containing a test pattern.

Clients may create one or more *display documents* and place text, shapes and bitmaps on them. They may assign the display documents to *display devices* (once they have claimed ownership of the display device in question); only one document may be placed on a display device at a single moment. When no client-owned display documents are attached to a device, the device displays

the default (black) document.

Clients may display the same document on several displays simultaneously; each copy will respond to mouse and touchscreen events identically.

On the server console program, you can view a copy of every display device the server is controlling.

Optionally, clients may create new, temporary display devices, known as **virtual displays,** which do not take full-screen control of an entire monitor, but appear as windows on the desktop. You may move these windows around normally; otherwise, they are treated as the other display devices. (This feature is to aid clients in creating windows, displaying graphics in them, and responding to mouse and touchscreen events. Clients can of course accomplish this by programming Windows directly, unless they are running on a different computer to the server.)

Documents and devices must have *unique* names. (Display devices may have more than one alias refer to them, but each alias must be unique.)

To communicate with the display system, the server uses DirectX (DirectDraw) with multimonitor extensions (available under Windows 2000, but not Windows NT 4).

The number of colours and display resolution used on a monitor is determined by Windows (*Control Panel* → *Display*).

Display and document sizes

The size of a physical display is obviously predetermined. You can query its size in pixels using the **DisplayGetSize** command.

The size of a virtual display may also be queried with **DisplayGetSize.** However, at the moment you create a virtual display (using **DisplayCreateDevice**) you may specify its size in pixels. You may also choose whether the user should be allowed to resize the window, and you can request a starting position on the desktop; however, you cannot prevent the user moving the window.

Physical and virtual displays may show documents unchanged (showing a scroll bar if the document is larger than the display), or *scale* their document to fit the device's window size. Choose this using the **DisplayScaleDocuments** command. Scaling is off by default.

You may specify the logical size of a *document* using the **DisplaySetDocumentSize** command. Until you do this, the document will expand to surround all the objects you place in it. The size that you set in this way affects only one thing – the size to which the document is scaled/scrolled.

This enables you to scale documents for different monitors with ease. Suppose you have created a document and laid out objects on it on the assumption that it will be viewed on a 1024 × 768 display. If you have to use it on an 800 × 600 display, the right and bottom edges will not be seen. But you can scale it instead:

```
DisplaySetDocumentSize mydoc 1024 768
DisplayScaleDocuments smallscreen on
DisplayShowDocument smallscreen mydoc
```

This facility means you can choose your coordinate space freely. However, you should note that

shrinking very detailed pictures to fit onto a smaller monitor may result in loss of detail.

Mouse and touchscreen events

If you can move the mouse cursor onto a display (and the display is enabled for mouse input, see *User Guide*), clicking the mouse will have the same effects as touching the touchscreen, and vice versa.

Keyboard events

Whisker supports keyboard events. This facility is intended primarily for human testing.

One Windows PC can only have one keyboard attached to it. It is therefore unlikely to be useful to run several clients that require keyboard input simultaneously, as only one can receive keyboard events at one time. This is exactly the same as other programs under Windows: you can't type into your wordprocessor and your Telnet session *at the same time* (you can of course switch between them) because only one window has what is termed the **focus** at one moment. The same is true of Whisker: only when a display has the focus will keyboard events work.

In practice, then, single clients for testing humans may find keyboard events useful. Keyboard events are associated with display documents. They are slightly different to other kinds of events, as you do not specify which keys you are interested in; simply that you want to be notified when keys are pressed (or released, or both). Whisker then sends a `KeyEvent:` message telling the client *which* key has been pressed. See **DisplayKeyboardEvents** for details.

Display system commands

Displays and associated devices are controlled using the following commands:

- DisplayClaim
- DisplayRelinquishAll
- DisplayCreateDevice
- DisplayDeleteDevice
- DisplaySetAlias
- DisplayGetSize
- DisplayScaleDocuments
- DisplayCreateDocument
- DisplayDeleteDocument
- DisplayShowDocument
- DisplaySetDocumentSize
- DisplayBlank
- DisplaySetBackgroundColour
- DisplayAddObject
- DisplayDeleteObject
- DisplaySetEvent
- DisplayClearEvent
- DisplaySetEventTransparency
- DisplayEventCoords
- DisplayBringToFront
- DisplaySendToBack

- DisplayKeyboardEvents

8.8.9.1 DisplayClaim

Message

DisplayClaim <*devicenumber*> **[-alias** <*newalias*>**]**
DisplayClaim <*devicegroup*> <*devicename*> **[-alias** <*newalias*>**]**

Originator

Client

Response

SyntaxError: insufficient parameters to DisplayClaim
SyntaxError: invalid parameters to DisplayClaim
Error: device number isn't a number!
ClaimAccepted: <*devicenumber*>
Error: alias is blank
Error: can't have spaces in an alias
Error: aliases can't begin with a digit
Error: alias already exists
ClaimAccepted: <*devicenumber*> (alias not set)
ClaimRejected: <*devicenumber*> is a non-existent display device
ClaimRejected: display device <*devicenumber*> is already claimed
ClaimRejected: group/device <*devicegroup*>/<*devicenumber*> not recognized
Info: new alias created for server-owned display device <*devicenumber*>

Response (immediate socket)

Success
Failure

Details

Immediate socket returns success *only* if (1) the device is claimed; and (2) if an alias was specified, that the alias was instantiated successfully.

Server device names. If you use the second form of the command, the device specified by the combination of the *devicegroup* and *devicename* parameters is claimed.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- Device names and aliases
- DisplaySetAlias
- DisplayRelinquishAll

8.8.9.2 Display Documents: size and scaling

Display Document Size:
A display document always has a size, which can be explicitly set.

If it is not set explicitly:

- The size of the document will be the extent of the objects within it.
- If an object is added to a document which exceeds the limit of its size, the size of the document will increase unless it has been explicitly set.

If it **is** set explicitly:

- The size of the document will be unaffected by the objects within it (which may exceed the size of the document).

Displaying a Document:
If a document is displayed on a device with scaling **on**:

- The document will be scaled so that the documents size corresponds to the whole of the physical display area. Any objects which exceed the size of the document will be clipped.

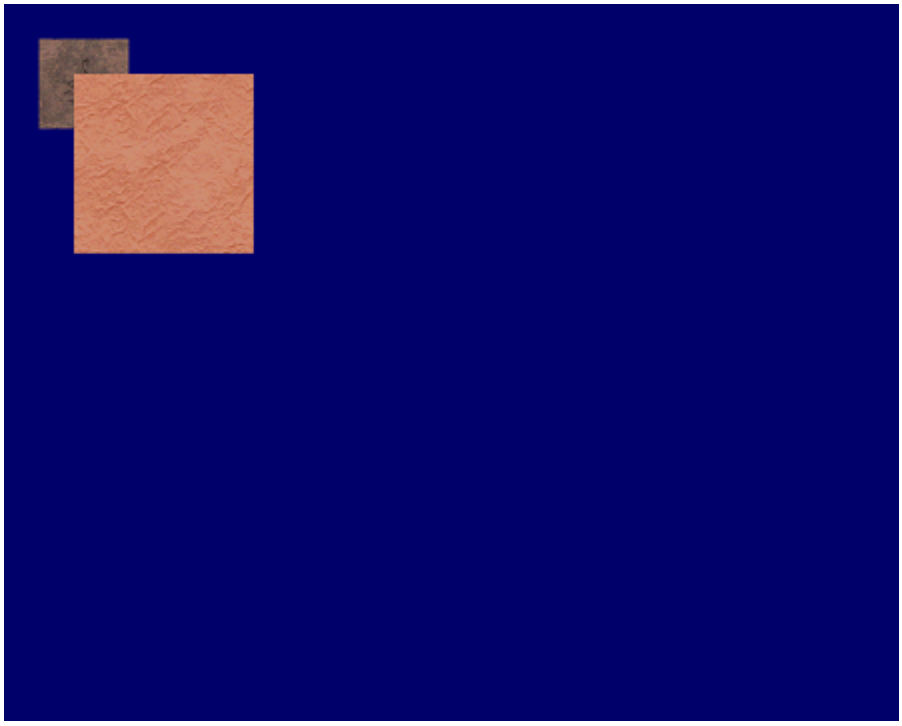
If a document is displayed on a device with scaling **off**:

- If the document is smaller than the physical display, the size of the document has no effect. The background (and any objects which exceed the document) will be displayed. Any objects which exceed the size of the **physical display** will be clipped.)

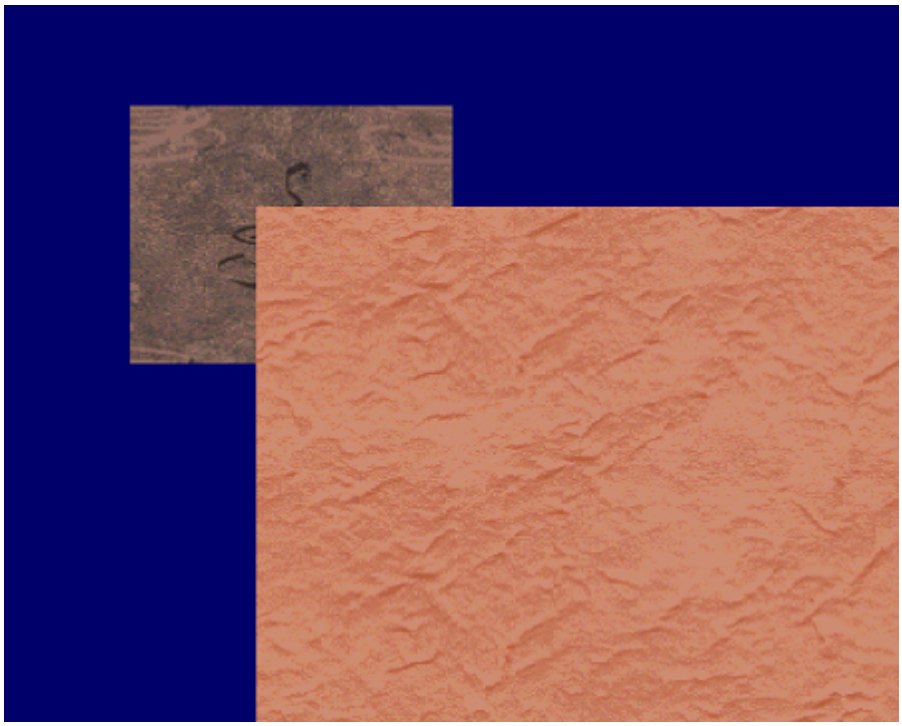
- If the document is larger than the physical display, scroll bars will appear to allow the larger document to be viewed.

If event coordinates are turned on (see *DisplayEventCoords*) the coordinates reported are the *document's* coordinates, regardless of scaling.

Example:
Here are two pictures of the same display, to the same scale, showing the same document, with scaling switched off:



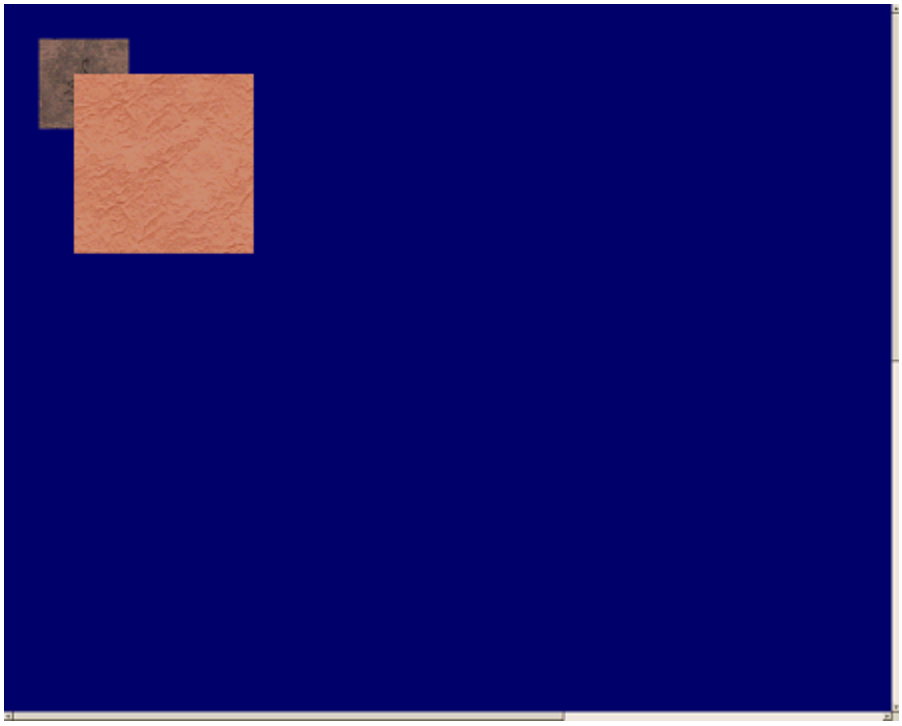
… and on:



The document has not set its size explicitly, so it is scaled to the limit of the objects within it.

If we take the same document and set its size to 2000 x 2000, we would see the following on that monitor — with scaling off:

monitor (0,0)
document (0,0)

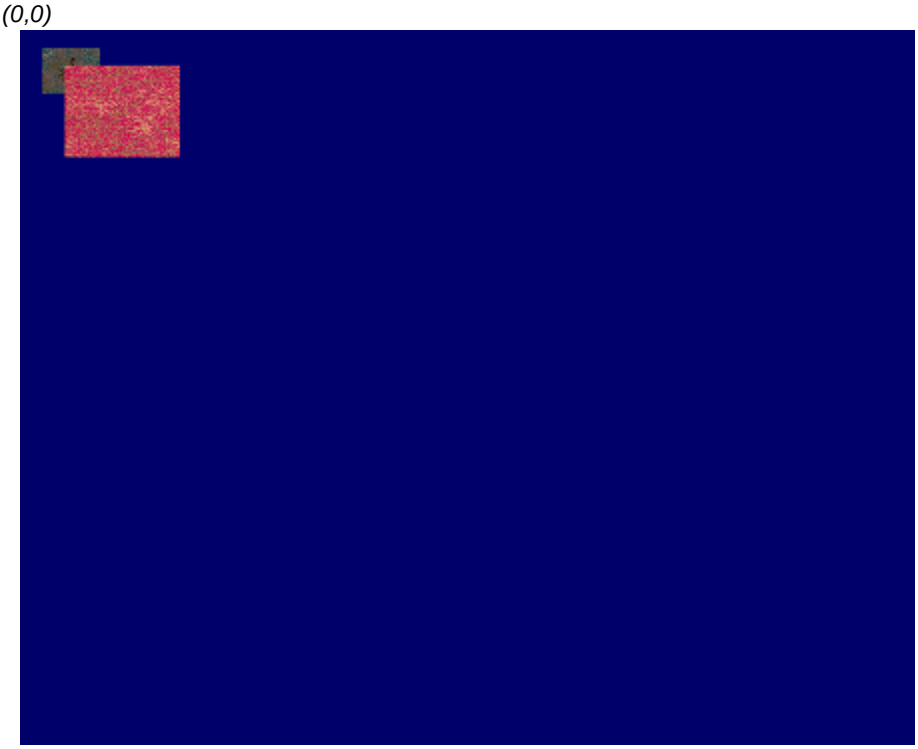


monitor (1279, 1023)
document (1279, 1023)

(Note that if the document is larger than the monitor and is not being scaled, scroll bars appear; if you want the document to be clipped with no scroll bars, set the document to the exact size of the monitor.)

… and with scaling on:

monitor (0,0)
document (0,0)



monitor (1279, 1023)
document (1999, 1999)

If the document set its size to something *smaller* than the limit of the objects within it, and scaling is switched on, then parts of the document will be clipped (just as if scaling is switched off and the document is larger than the physical display).

Note: documents are drawn faster when they are not being scaled (sometimes significantly, e.g. 12 ms versus 185 ms for a complete redraw of the document shown above on a 1020 x 717 non-exclusive, non-fullscreen DirectDraw window using a PIII-750 laptop with Whisker at 'normal' thread priority). This delay does not affect client communications (as redrawing is performed by a separate thread), and the time *during which the visible display is being updated* is less than 1 ms – for DirectDraw windows, the drawing routines draw to a memory buffer and then use a rapid technique for switching that memory to video RAM – thus, the 12/185 ms delay is the approximate lag from issuing a DisplayShowDocument command to the document appearing on the screen. If video performance is critical, use a full-screen DirectDraw window, switch WhiskerServer into real-time mode and do not scale documents.

Development note: benchmark these optimal conditions in CDisplayView.

8.8.9.3 DisplayRelinquishAll

Message

DisplayRelinquishAll

Originator

Client

Response

Info: all display devices relinquished
SyntaxError: invalid parameters to DisplayRelinquishAll

Response (immediate socket)

Success

Details

Yields control of all server-owned displays and deletes all client-owned displays.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayClaim

8.8.9.4 DisplayCreateDevice

Message

DisplayCreateDevice *<devicename>* **[–resize on|off] [-rectangle** *<left> <top> <width> <height>***] [–directdraw on|off] [–debugtouches]**

Originator

Client

Response

Info: new display device <*devicename*> created
Error: can't create new display device
Error: device already exists
Error: can't create alias for device, so can't create device
SyntaxError: invalid parameters to DisplayCreateDevice
SyntaxError: insufficient parameters to DisplayCreateDevice

Response (immediate socket)

Success
Failure

Details

Creates a new display device — that is, a new window on the Windows desktop that you may move around. (Remember, the full-screen display devices are created for you by the server, and are managed by the server as a limited resource, meaning that two clients cannot gain access to the same monitor simultaneously. However, clients can create 'normal' windows as they wish, and use them in the same way as the full-screen displays.)

The new 'device' is referred to solely by its name, and is private to the client that creates it.

The **rectangle** parameter allows you to specify the initial size of the window. The top left of the rectangle is that of the *whole window*; the width/height are that of the *client area* (the usable part of the window, inside the window's borders).

The **resize** parameter allows you to prevent the user resizing the window.

The **directdraw** parameter allows you to choose whether the new device window is to use DirectDraw (on) or only the Windows GDI (off). As it gives better performance, DirectDraw is on by default.

Device names must be unique (across virtual displays and aliases for physical displays).

Note that as the window being created has a title bar, there is a **minimum width** that can be created – on my system the minimum client area width is 102 pixels for resizable windows or 100 pixels for non-resizable windows. If you try to create a window smaller than this, a 104-pixel-wide window is created. Use DisplayGetSize to retrieve the actual size – but **note** that DisplayGetSize will not return the correct information until the user-interface thread has displayed the window, which may be a few milliseconds after the client is first able to send a DisplayGetSize command to query the new display! Before the user-interface thread has created the window, DisplayGetSize will return –1 for all sizes.

The **debugtouches** option is used by the WhiskerSDK internally when requesting virtual monitor windows for claimed physical devices. If this option is used, then touches will be shown on all documents displayed on the device.

Revision history

Implemented by WhiskerServer version 2.3. DebugTouches option added in WhiskerServer 2.12.1

See also

- Display devices

- DisplaySetAlias
- DisplayDeleteDevice

8.8.9.5 DisplayDeleteDevice

Message

DisplayDeleteDevice *<devicename>*

Originator

Client

Response

Info: display device(s) deleted
Error: no such client-owned display device
SyntaxError: invalid parameters to DisplayDeleteDevice
SyntaxError: insufficient parameters to DisplayDeleteDevice

Response (immediate socket)

Success
Failure

Details

Deletes a client-owned display device. (This cannot affect server-owned display devices.)

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayCreateDevice
- DisplayDeleteAllDevices

8.8.9.6 DisplayDeleteAllDevices

Message

DisplayDeleteAllDevices

Originator

Client

Response

Info: display device(s) deleted
Warning: no such client-owned display device
SyntaxError: invalid parameters to DisplayDeleteDevice

Response (immediate socket)

Success
Failure

Details

Deletes all client-owned display devices.

Revision history

Implemented in WhiskerServer version 2.4.01.

See also

- Display devices
- DisplayCreateDevice
- DisplayDeleteDevice

8.8.9.7 DisplaySetAlias

Message

DisplaySetAlias *<devicename> <alias>*

Originator

Client

Response

Error: no such display device – *<devicename>*
Error: alias is blank
Error: can't have spaces in an alias
Error: aliases can't begin with a digit
Error: alias already exists
Info: new alias created for client-owned display device
Info: new alias created for server-owned display device *<devicenumber>*
SyntaxError: invalid parameters to DisplaySetAlias
SyntaxError: insufficient parameters to DisplaySetAlias

Response (immediate socket)

Success
Failure

Details

The new alias must be unique to all your display devices.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayClaim
- DisplayCreateDevice

8.8.9.8 DisplayGetSize

Message

DisplayGetSize *<devicename>*

Originator

Client

Response

Error: no such display device – *<devicename>*
Info: Device *<devicename>* size: x=*<xsize>* y=*<ysize>*
SyntaxError: invalid parameters to DisplayGetSize
SyntaxError: insufficient parameters to DisplayGetSize

Response (immediate socket)

Size *<xsize>* *<ysize>*
Failure

Details

Retrieves the size of the client rectangle (the usable part of the display window).

Note that DisplayGetSize may return (x = -1, y = -1) immediately after creation of a 'virtual' window; see DisplayCreateDevice.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayCreateDevice

8.8.9.9 DisplayScaleDocuments

Message

DisplayScaleDocuments *<devicename>* **on|off**

Originator

Client

Response

Error: no such display device – *<devicename>*
Info: Scaling documents on device *<devicename>*
Info: Not scaling documents on device *<devicename>*
SyntaxError: invalid parameters to DisplayScaleDocuments
SyntaxError: insufficient parameters to DisplayScaleDocuments

Response (immediate socket)

Success
Failure

Details

Sets a device to scale, or not scale, the size of the document up or down to the resolution of the display device. See Display Documents: size and scaling for details.

Scaling is **off** by default.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplaySetDocumentSize
- Display Documents: size and scaling

8.8.9.10 DisplayCreateDocument

Message

DisplayCreateDocument *<docname>*

Originator

Client

Response

Info: new document created (*<docname>*)
Error: document already exists
Error: can't create new document
SyntaxError: invalid parameters to DisplayCreateDocument
SyntaxError: insufficient parameters to DisplayCreateDocument

Response (immediate socket)

Success
Failure

Details

Creates a new document named *docname*, but does not display it anywhere. **The document name must be unique.**

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayShowDocument
- DisplayDeleteDocument

8.8.9.11 DisplayDeleteDocument

Message

DisplayDeleteDocument <*docname*>

Originator

Client

Response

Info: document(s) deleted
Error: Document *docname* not found by DisplayDeleteDocument
SyntaxError: invalid parameters to DisplayDeleteDocument
SyntaxError: insufficient parameters to DisplayDeleteDocument

Response (immediate socket)

Success
Failure

Details

Deletes the chosen document. If the document was being displayed, the display(s) show a blank screen instead.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayCreateDocument
- DisplayDeleteAlDocuments

8.8.9.12 DisplayDeleteAllDocuments

Message

DisplayDeleteAllDocuments

Originator

Client

Response

Info: document(s) deleted
Warning: No display documents to delete!
SyntaxError: invalid parameters to DisplayDeleteAllDocuments

Response (immediate socket)

Success
Failure

Details

Deletes all display documents. If any documents were being displayed, the displays show a blank screen instead.

Revision history

Implemented in WhiskerServer version 2.4.01.

See also

- Display devices
- DisplayCreateDocument
- DisplayDeleteDocument

8.8.9.13 DisplayShowDocument

Message

DisplayShowDocument *<devicename> <docname>*

Originator

Client

Response

Info: displaying document *<docname>*
Error: no such display device – *<devicename>*
Error: Document *docname* not found by DisplayShowDocument
SyntaxError: invalid parameters to DisplayShowDocument
SyntaxError: insufficient parameters to DisplayShowDocument
Error: DisplayShowDocument not possible, would create too many video views for *document*, use fewer or reconfigure WhiskerServer
Error: DisplayShowDocument not possible, current views for *document* don't support video, reconfigure WhiskerServer

Response (immediate socket)

Success
Failure

Details

Makes the selected device display the selected document. You may show the same document on several

displays simultaneously.

Revision history

Implemented by WhiskerServer version 2.3.
Video failure codes in version 4.0

See also

- Display devices
- DisplayCreateDocument
- DisplayScaleDocuments
- DisplaySetDocumentSize
- DisplayBlank
- DisplaySetBackgroundColour
- DisplayAddObject
- DisplayKeyboardEvents
- DisplayCacheChanges
- DisplayShowChanges
- Video objects

8.8.9.14 DisplayCacheChanges

Message

DisplayCacheChanges *<docname>*

Originator

Client

Response

Info: Cacheing changes on document *docname*
Error: Document *docname* not found by DisplayCacheChanges
SyntaxError: invalid parameters to DisplayCacheChanges
SyntaxError: insufficient parameters to DisplayCacheChanges

Response (immediate socket)

Success
Failure

Details

Enables cacheing mode for the selected document. Commands that change the document's size, background colour or contents will not be shown on any view of the document until a DisplayShowChanges command for that document is issued.

Revision history

Implemented in WhiskerServer version 2.6.

See also

- DisplayShowChanges

8.8.9.15 DisplayShowChanges

Message

DisplayShowChanges *<docname>*

Originator

Client

Response

Info: Document *docname* updated from cache
Error: Document *docname* not found by DisplayShowChanges
SyntaxError: invalid parameters to DisplayShowChanges
SyntaxError: insufficient parameters to DisplayShowChanges

Response (immediate socket)

Success
Failure

Details

Disables cacheing mode for the selected document. Any changes to the document's size, background colour or contents made while cacheing was enabled will be shown on all displays.

Revision history

Implemented in WhiskerServer version 2.6.

See also

- DisplayCacheChanges

8.8.9.16 DisplaySetDocumentSize

Message

DisplaySetDocumentSize *<docname> <xsize> <ysize>*

Originator

Client

Response

Info: document *<docname>* logical size set (x=*<xsize>*, y=*<ysize>*)
SyntaxError: invalid parameters to DisplaySetDocumentSize
SyntaxError: insufficient parameters to DisplaySetDocumentSize
Error: Document *docname* not found by DisplaySetDocumentSize

Response (immediate socket)

Success
Failure

Details

Sets an explicit size for a document. Until this call is made, the size of the document will be given by the extent of objects added to it.

Can be used with DisplayScaleDocuments to provide a coordinate system for the a displayed document which is independent of the physical resolution of the display.

See Display Documents: size and scaling for more details.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayScaleDocuments
- DisplayGetDocumentSize
- Display Documents: size and scaling

8.8.9.17 DisplayGetDocumentSize

Message

DisplayGetSize *<docname>*

Originator

Client

Response

Error: Document *docname* not found by DisplayGetDocumentSize
Info: Document *<docname>* size: x=*<xsize>* y=*<ysize>*
SyntaxError: invalid parameters to DisplayGetDocumentSize
SyntaxError: insufficient parameters to DisplayGetDocumentSize

Response (immediate socket)

Size *<xsize> <ysize>*
Failure

Details

Retrieves the size of the document. If this has been set by DisplaySetDocumentSize, returns whatever was set. Otherwise, returns the extent of the document (see Display Documents: size and scaling for details).

Revision history

Implemented in WhiskerServer version 2.10.2.

See also

- Display devices
- DisplaySetDocumentSize

- DisplayScaleDocuments
- Display Documents: size and scaling

8.8.9.18 DisplayBlank

Message

DisplayBlank *<devicename>*

Originator

Client

Response

Error: no such display device – *<devicename>*
Info: blanking device *<devicename>*
SyntaxError: invalid parameters to DisplayBlank
SyntaxError: insufficient parameters to DisplayBlank

Response (immediate socket)

Success
Failure

Details

If the device is displaying a document, this command removes the document from the display, leaving a blank (black) screen.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices

8.8.9.19 DisplaySetBackgroundColour

Message

DisplaySetBackgroundColour *<docname> <red> <green> <blue>*

Originator

Client

Response

Error: Document *docname* not found by DisplaySetBackgroundColour
Error: colour parameters invalid
Info: background colour set
SyntaxError: invalid parameters to DisplaySetBackgroundColour
SyntaxError: insufficient parameters to DisplaySetBackgroundColour

Response (immediate socket)

Success
Failure

Details

Sets the background colour of the document. Each parameter (*red, green, blue*) can be a number from 0 to 255. (For example, black is 0 0 0; white is 255 255 255.)

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices

8.8.9.20 DisplayAddObject

Message

DisplayAddObject *<docname> <objectname> <objecttype> <parameters...>*

Originator

Client

Response

General:

SyntaxError: insufficient parameters to DisplayAddObject
SyntaxError: invalid parameters to DisplayAddObject
Error: document *docname* not found by DisplayAddObject
Error: unknown object type

Bitmaps:

SyntaxError: insufficient parameters to DisplayAddObject / bitmap
SyntaxError: stretched object of zero size (DisplayAddObject/bitmap)
Error: can't create bitmap
Error: bitmap has problems, deleting it
Info: bitmap added

Other responses are also possible for video object types; see DisplayAddObject: video.

Response (immediate socket)

Success
Failure

Details

Adds an object named *objectname* to the document named *docname*. The object may be of several types, listed below. The parameters differ for each type of object.

If a parameter includes spaces or semicolons, **enclose it in quotes ("").**

OBJECTTYPE = video
For video, see DisplayAddObject: video.

OBJECTTYPE = text
PARAMETERS = *<x> <y> <text>* **[-height** *<height>***] [-font** *<font>***] [-textcolour** *<red> <green> <blue>***] [-backcolour** *<red> <green> <blue>***] [-opaque] [-italic] [-underline] [-weight** *<weight>***] [-top | -baseline | -middle | -bottom] [-left | -centre | -right]**

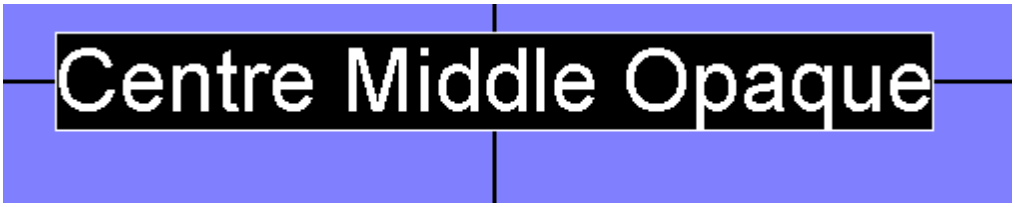
Inserts text.

- *x* = *x*-coordinate of (by default) the top left point; the meaning of these coordinates can be changed by the vertical/horizontal alignment options below
- *y* = *y*-coordinate of (by default) the top left point; the meaning of these coordinates can be changed by the vertical/horizontal alignment options below
- *text* = text to be displayed. Enclose the text in double-quotes if necessary.
- *height* = height of the font (0, the default, gives you a reasonable size)
- *font* = name of the font to use, e.g. "Times New Roman" (use quotes as the font name has spaces in it). If no font is specified, or if the font you specify cannot be found, a default system font is used.
- *textcolour* = text colour (*red, green, blue* take values from 0–255). Default colour is white (255,255,255).
- *backcolour* = background colour, if used. Default colour is black (0,0,0). Only relevant if the background mode is 'opaque'.
- **–opaque** sets opaque background mode. Opaque mode fills in the 'gaps' with the background colour; transparent mode allows you to see through the gaps in the text. Default is transparent.
- **–italic** makes the font italic
- **–underline** underlines the font
- *weight* = number from 1–1000 to specify an approximate font weight (how bold it is), or 0 to specify a default weight (which is, perhaps not surprisingly, the default).
- **–top | –baseline | –middle | –bottom** determines the vertical alignment (default is top). The top (ascent line), baseline, middle (vertical centre) or bottom (descent line) of the text can be aligned to the *y* coordinate. (What's the difference between baseline and bottom? Look at the letter y: it descends below the baseline of the text.)
- **–left | –centre | –right** determines the horizontal alignment (default is left). The left edge, centre, or right edge of the text can be aligned to the *x* coordinate.

The precise horizontal position and extent of the text depends upon whether it is opaque or not, although the difference will not be visible except for very large text.

- If it is opaque, the **background rectangle** will be exactly aligned to the coordinates given, and GetObjectExtent will return the rectangle painted with background colour (including any background painted at the left and right edges.
- If it is not opaque, the **text characters** will be exactly aligned to the coordinates given, and GetObjectExtent will return the rectangle of a close surrounding (horizontally) rectangle.

Example figures showing the alignment (black lines are coordinates, white rectangles are extents of text objects):



OBJECTTYPE = bitmap
PARAMETERS = *<x> <y> <filename>* **[-stretch|-clip] [-height** *<height>***] [-width** *<width>***] [-top | -middle | -bottom] [-left | -centre | -right]**

Inserts a bitmap from a file.

- *x* = *x*-coordinate of (by default) the top left point; the meaning of these coordinates can be changed by the vertical/horizontal alignment options below
- *y* = *y*-coordinate of (by default) the top left point; the meaning of these coordinates can be changed by the vertical/horizontal alignment options below
- *filename* = filename of bitmap
- *width* = width to display bitmap at, –1 for the bitmap's own width (default)
- *height* = height to display bitmap at, –1 for the bitmap's own height (default)
- **–stretch|–clip:** stretch or clip bitmap (clipping is the default).
- **–top | –middle | –bottom** determines the vertical alignment (default is top). The top, middle (vertical centre), or bottom of the bitmap can be aligned to the *y* coordinate.
- **–left | –centre | –right** determines the horizontal alignment (default is left). The left edge, centre, or right edge of the bitmap can be aligned to the *x* coordinate.

OBJECTTYPE = line
PARAMETERS = *<x1> <y1> <x2> <y2>* **[***<pen_options>***]**

Draws a line from (*x1, y1*) to (*x2, y2*).

OBJECTTYPE = arc
PARAMETERS = *<x1> <y1> <x2> <y2> <x3> <y3> <x4> <y4>* **[***<pen_options>***]**

Draws an arc that fits within the rectangle (*x1, y1*)–(*x2, y2*). The arc begins at (*x3, y3*) and ends at (*x4, y4*), approximately. As the Windows GDI documentation states, this function 'draws an elliptical arc. The arc drawn by using the function is a segment of the ellipse defined by the specified bounding rectangle. The actual starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The actual ending point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified ending point intersects the ellipse. The arc is drawn in [an anticlockwise] direction.'

OBJECTTYPE = bezier
PARAMETERS = *<x1> <y1> <x2> <y2> <x3> <y3> <x4> <y4>* **[***<pen_options>***]**

Draws a cubic Bézier spline. The curve is drawn from (*x1, y1*) to (*x4, y4*); the points (*x2, y2*) and (*x3, y3*) are control points (that 'pull' the curve towards them).

OBJECTTYPE = chord
PARAMETERS = *<x1> <y1> <x2> <y2> <x3> <y3> <x4> <y4>* **[***<pen_options>***] [***<brush_options>***]**

Draws a chord (a closed figure bounded by the intersection of an ellipse and a line segment). The (*x1*, *y1*) and (*x2*, *y2*) parameters specify the upper-left and lower-right corners, respectively, of a rectangle bounding the ellipse that is part of the chord. The (*x3*, *y3*) and (*x4*, *y4*) parameters specify the endpoints of a line that intersects the ellipse.

This function takes the *pen* and *brush* options described below.

OBJECTTYPE = ellipse
PARAMETERS = *<x1> <y1> <x2> <y2>* **[***<pen_options>***] [***<brush_options>***]**

Draws an ellipse that fits within the rectangle (*x1, y1*)–(*x2, y2*). The centre of the ellipse is at the centre of this rectangle.

OBJECTTYPE = pie
PARAMETERS = *<x1> <y1> <x2> <y2> <x3> <y3> <x4> <y4>* **[***<pen_options>***] [***<brush_options>***]**

Draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines. The center of the arc is the center of the bounding rectangle (*x1*, *y1*)–(*x2*, *y2*). The starting and ending points of the arc are specified by (*x3*, *y3*) and (*x4*, *y4*).

The arc is drawn anticlockwise. Two additional lines are drawn from each endpoint to the arc's center. The pie-shaped area is then filled. If (*x3*, *y3*) = (*x4*, *y4*), the result is an ellipse with a single line from the center of the ellipse to the point (*x3*, *y3*) or (*x4*, *y4*).

OBJECTTYPE = polygon
PARAMETERS = *<numpoints> <x1> <y1> <x2> <y2> <x3> <y3> <...> <...>* **[-alternate|-winding] [** *<pen_options>***] [***<brush_options>***]**

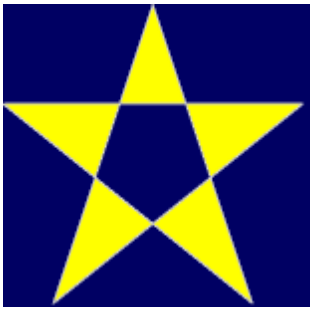
Draws a polygon (*x1, y1*)–(*x2, y2*)–(*x3, y3*)–… *Numpoints* specifies the total number of (*x*, *y*) points. The polygon is closed, if necessary, by joining the last point to the first.

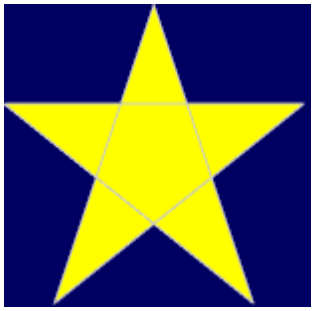
Polygons may be filled in one of two ways. As the Windows GDI documentation describes:

- **–alternate:** the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on. This mode is the default.
- **–winding:** the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or an anticlockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through an anticlockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

The following figures illustrate the two filling methods; the five points of the pentacle were drawn in the order top → bottom left → right → left → bottom right.

Alternate Winding





OBJECTTYPE = rectangle
PARAMETERS = *<x1> <y1> <x2> <y2>* **[***<pen_options>***] [***<brush_options>***]**

Draws a rectangle from (*x1, y1*) to (*x2, y2*).

OBJECTTYPE = roundrect
PARAMETERS = *<x1> <y1> <x2> <y2> <x3> <y3>* **[***<pen_options>***] [***<brush_options>***]**

Draws a rounded rectangle. The rectangle (*x1, y1*)–(*x2, y2*) is drawn but with corners that are part of an ellipse whose width is *x3* and whose height is *y3*.

OBJECTTYPE = camcogquadpattern
PARAMETERS = *<x1> <y1> <pixelwidth> <pixelheight> <topleft1>...<topleft8> <topright1>...<topright8> <bottomleft1>...<bottomleft8> <bottomright1>...<bottomright8> <topleftred> <topleftgreen> <topleftblue> <toprightred> <toprightgreen> <toprightblue> <bottomleftred> <bottomleftgreen> <bottomleftblue> <bottomrightred> <bottomrightgreen> <bottomrightblue> <backgroundred> <backgroundgreen> <backgroundblue>*

Draws a pattern composed of four quadrants, each made up of 8x8 small rectangles (pixels). For each quadrant, eight rows are specified; each row's pattern is specified by a single number from 0 to 255, of which the high bit represents the left-hand pixel and the low bit the right-hand pixel.

Pen options

Objects that consist of or are bounded by lines (i.e. line, arc, bezier, chord, ellipse, pie, polygon, rectangle, roundrect) are drawn with the following *pen* options:

-penstyle solid|dash|dot|dashdot|dashdotdot|null|insideframe
-penwidth *<width>*
-pencolour *<red> <green> <blue>*

The options *solid, dash, dot, dashdot,* and *dashdotdot* should be fairly self-explanatory. *Null* gives an invisible pen. *Insideframe* is relevant when the pen is thick; for example, if you draw a circle of diameter 100 units with a pen of width 20 units, the circle will normally end up having an *external* diameter of 120 units and an internal diameter of 80 units (as the pen overlaps by 10 units on the inside and the outside of the circle). If you specify *insideframe*, the circle's outside diameter is 100 units in this situation.

The default is a solid white pen of width 1.

Brush options

Solid figures (chord, ellipse, pie, polygon, rectangle, roundrect) all take *brush* options. The brush options determine how the object is filled. A brush may be hollow (invisible), solid, or hatched (in which case you can specify the hatching style and colour). The default is a solid white brush.

-brushhollow

-brushsolid *<red> <green> <blue>*
-brushhatched bdiagonal|cross|diagcross|fdiagonal| horizontal|vertical *<red> <green> <blue>*

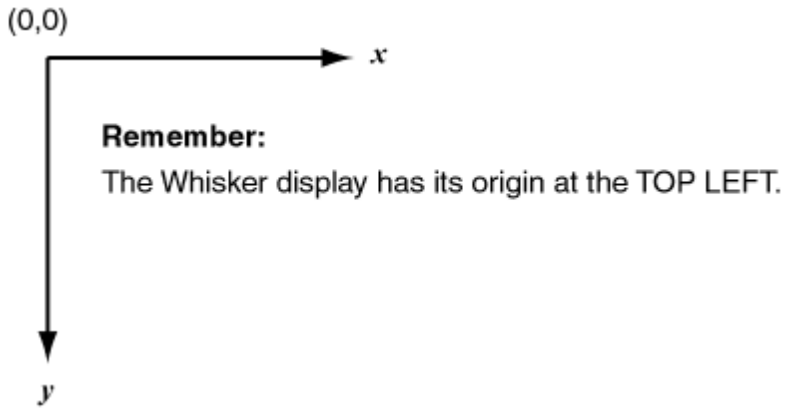
The hatching styles are:

- **bdiagonal:** lines at 45° anticlockwise from the horizontal axis;
- **cross:** horizontal and vertical lines;
- **diagcross:** lines at 45° clockwise and anticlockwise from the horizontal;
- **fdiagonal:** lines at 45° clockwise to the horizontal;
- **horizontal:** horizontal lines;
- **vertical:** vertical lines.

If a hatched brush is used, further options come into play. A hatched brush may either be opaque or transparent. If it is transparent, you can see through the hatching to whatever is beneath. If it is opaque, you may set the background colour used to fill in the gaps in the hatching. (The default settings for a hatched brush are for it to be opaque with black as the background colour.)

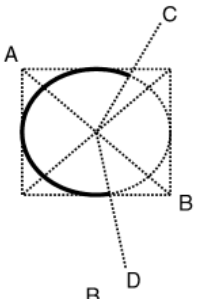
-brushtransparent
-brushopaque
-brushbackground *<red> <green> <blue>*

The coordinate system



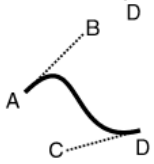
Geometrical object types

Arc



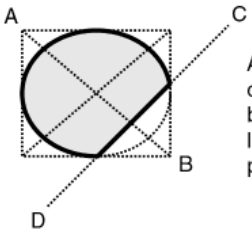
The arc is part of the ellipse bounded by the rectangle from A to B. Imaginary lines are drawn from the centre of the rectangle to C, and to D. The arc begins where these lines intersect the ellipse. It is drawn anticlockwise (in other words, if C and D were reversed, the opposite part of the ellipse would be drawn; see "Chord" for a drawn example).

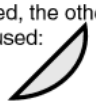
Bezier



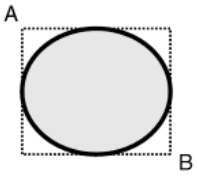
The Bezier spline is drawn from A to D. Points B and C are "control points" that pull the curve towards them.

Chord



A chord is a solid figure created by the intersection of an ellipse and a straight line. The ellipse is bounded by the rectangle between A and B. C and D specify the line. (If C and D were reversed, the other part of the ellipse would be used:  .)

Ellipse

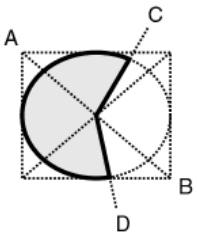


The ellipse is drawn within the rectangle bounded by A and B. The centre of the ellipse is at the centre of the rectangle.

Line

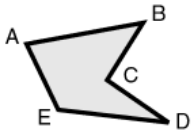
Not too complicated.

Pie



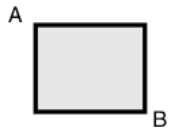
A pie is exactly like an arc but is a solid figure. (Again, it is drawn anticlockwise, and reversing C and D would cause the rest of the pie to be drawn instead.)

Polygon



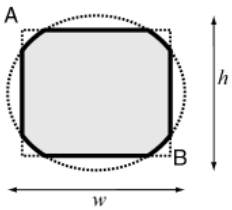
A polygon joins all the specified points, in order, completing the shape if necessary. The fill mode is complicated. **Alternate:** the system fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, the system fills the area between the first and second side, between the third and fourth side, and so on. This mode is the default. **Winding:** the system uses the direction in which a figure was drawn to determine whether to fill an area. Each line segment in a polygon is drawn in either a clockwise or an anticlockwise direction. Whenever an imaginary line drawn from an enclosed area to the outside of a figure passes through a clockwise line segment, a count is incremented. When the line passes through an anticlockwise line segment, the count is decremented. The area is filled if the count is nonzero when the line reaches the outside of the figure.

Rectangle



Not too complicated.

Roundrect



A rounded rectangle is drawn. The rectangle from A to B is drawn with corners that are part of the ellipse whose width is w and whose height is h. (The ellipse and the rectangle share their centre.)

Touch-/mouse-sensitive areas of objects

Objects that have a painted area can be set to generate events if mouse or touchscreen events over that

area, by means DisplaySetEvent.
Object types with painted areas are Bitmap, Ellipse, Chord, Pie, Rectangle, RoundRect, Polygon and Text. Line, Arc and Bezier objects cannot be used as touch- or mouse-sensitive areas.

Examples

For a document called Wanda…

```
displayaddobject wanda apple bitmap 50 50 \\coffee.bmp
displayaddobject wanda pear bitmap 100 100 \\santa_fe.bmp
displayaddobject wanda objtext text 150 150 "Test document: Wanda" -height 100 -font "times new roman" -italic
displayaddobject wanda objline line 50 50 200 600
displayaddobject wanda objarc arc 100 100 400 400 150 100 350 100 -penwidth 5 -pencolour 100 0 0 -penstyle dash
displayaddobject wanda objbezier bezier 100 100 100 400 400 100 400 400 -penstyle dash
displayaddobject wanda objchord chord 300 300 500 500 300 350 500 350 -penstyle insideframe -pencolour 0 100 0 -penwidth 5 -brushsolid 50 50 0
displayaddobject wanda objellipse ellipse 650 100 750 400 -brushhollow
displayaddobject wanda objpie pie 600 300 800 500 800 300 800 500 -brushhatched bdiagonal 128 128 128 -brushopaque -brushbackground 255 255 0
displayaddobject wanda objrect rectangle 250 550 150 450 -penstyle dashdotdot -brushsolid 128 128 128
displayaddobject wanda objroundrect roundrect 400 450 500 550 100 100 -brushhatched cross 255 255 0 -brushtransparent -brushbackground 255 0 255
displayaddobject wanda objpolygon polygon 3 400 500 600 450 600 550 -brushhatched horizontal 255 255 0 -brushtransparent -brushbackground 255 0 255
displayaddobject wanda objpolygon2 polygon 5 150 425 75 650 250 500 50 500 225 650 -alternate -brushsolid 255 255 0
```

Revision history

Implemented by WhiskerServer version 2.3.
CamcogQuadPattern added in v2.6.8.
Text alignment and touching added in v 2.11
Video added in v4.0
Vertical and horizontal alignment options for bitmaps and video added in v4.2.1.

See also

- Display devices
- DisplayCreateDocument
- DisplayShowDocument
- DisplayDeleteObject
- DisplaySetBackgroundColour
- DisplaySetEvent
- DisplayBringToFront
- DisplaySendToBack
- DisplayCacheChanges
- DisplayShowChanges

8.8.9.21 DisplayDeleteObject

Message

DisplayDeleteObject *<docname> <objectname>*

Originator

Client

Response

Info: object(s) deleted
SyntaxError: insufficient parameters to DisplayDeleteObject
SyntaxError: invalid parameters to DisplayDeleteObject
Error: document *docname* not found by DisplayDeleteObject
Error: No object called *objectname* found on display document *docname* by DisplayDeleteObject

Response (immediate socket)

Success
Failure

Details

Deletes an object from a document.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayAddObject

8.8.9.22 DisplayGetObjectExtent

Message

DisplayGetObjectExtent <*docname*> <*objectname*>

Originator

Client

Response

Info: Document <*docname*> object <*objectname*> extent: left=<*left*> top=<*top*> right=<*right*> bottom=<*bottom*>
SyntaxError: insufficient parameters to DisplayGetObjectExtent
SyntaxError: invalid parameters to DisplayGetObjectExtent
Error: Document *docname* not found by DisplayGetObjectExtent
Error: No object called *objectname* found on display document *docname* by DisplayGetObjectExtent

Response (immediate socket)

Extent <*left*> <*top*> <*right*> <*bottom*>
Failure

Details

Retrieves the extent of the object, as determined by DisplayAddObject. The (left, top) coordinate is the top

left of the **bounding rectangle** containing the object; the (right, bottom) coordinate is the bottom right. What's a bounding rectangle? The smallest rectangle that will contain the object. For example, imagine a circle: its bounding rectangle encloses the circle. For objects that the server had a problem creating, all coordinates are set to **-1**. From 17 May 2005, text objects report their extent correctly; before this, they returned all coordinates as -1.

Revision history

Implemented in WhiskerServer version 2.10.2.

See also

- DisplayAddObject

8.8.9.23 DisplaySetEvent

Message

DisplaySetEvent <*docname*> <*objectname*> <*eventclass*> <*eventmessage*>

Originator

Client

Response

SyntaxError: insufficient parameters to DisplaySetEvent
SyntaxError: invalid parameters to DisplaySetEvent
SyntaxError: no such event class
Error: document *docname* not found by DisplaySetEvent
Error: No object called *objectname* found on display document *docname* by DisplaySetEvent
Info: event created for object(s)

Response (immediate socket)

Success
Failure

Details

For every document named *docname* that contains an object named *objectname* which has a touch-sensitive region, the server adds an event to that object. The type of event is specified by *eventclass*, and may be one of the following:

EventClass	Description
MouseDown	The left mouse button is depressed over the object
MouseUp	The left mouse button is released over the object.
MouseDblClick	The object has been double-clicked (see below)
MouseMove	The mouse has moved whilst over the object
TouchDown	The touch stylus is detected over the object (object is touched)
TouchUp	The touch stylus stops being detected over the object
TouchMove	The detected touch stylus is moved whilst over the object

Note: A touch stylus or mouse pointer counts as being 'over' an object if its coordinates fall within the region which would be painted by a solid brush, or by the (clipped or stretched) bitmap contents. Thus

purely 'line-based' (Line, Bezier, Arc) objects will never receive touches (events cannot be added to these objects). As of 17 May 2005, or WhiskerServer v2.11, text can be touched (before this, it could not). Also, a mouse or stylus event over an unshaded region of a complex object (e.g. in the centre of a polygon) will not generate an event.

A note about mice: when you double-click the mouse button, Windows processes the *first* click as 'MouseDown' and the *second* click as 'MouseDblClick' — a 'MouseDown' event is *not* generated for the second click. This means that if you are interested in every mouse event, **you must request both MouseDown and MouseDblClick events** (using the same event message if you do not care about the distinction).

If the server display is double-clicked while the server's input is being used to emulate touches, then double click events are converted to ordinary 'touchdown' events.

Cacheing mode and display events

If cacheing mode is enabled, the client may set or clear events, or change event transparency, for objects not yet displayed (that is, *added since* DisplayCacheChanges), or for objects which are awaiting deletion (*deleted since* DisplayCacheChanges). Commands will take effect **immediately** for any objects currently displayed.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- Event
- DisplayAddObject
- DisplayClearEvent
- DisplaySetBackgroundEvent
- DisplayClearBackgroundEvent

8.8.9.24 DisplayClearEvent

Message

DisplayClearEvent *<docname> <objectname> <eventclass>*

Originator

Client

Response

SyntaxError: insufficient parameters to DisplayClearEvent
SyntaxError: invalid parameters to DisplayClearEvent
SyntaxError: no such event class
Error: document *docname* not found by DisplayClearEvent
Error: No object called *objectname* found on display document *docname* by DisplayClearEvent
Info: event(s) removed

Response (immediate socket)

Success
Failure

Details

The specified event class is cleared for the appropriate object(s) (see DisplaySetEvent).

If *eventclass* is blank, all events are cleared for the specified object.

Cacheing mode and display events

If cacheing mode is enabled, the client may set or clear events, or change event transparency, for objects not yet displayed (that is, *added since* DisplayCacheChanges), or for objects which are awaiting deletion (*deleted since* DisplayCacheChanges). Commands will take effect **immediately** for any objects currently displayed.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplaySetEvent
- DisplaySetBackgroundEvent
- DisplayClearBackgroundEvent

8.8.9.25 DisplaySetBackgroundEvent

Message

DisplaySetBackgroundEvent *<docname> <eventclass> <eventmessage>*

Originator

Client

Response

SyntaxError: insufficient parameters to DisplaySetEvent
SyntaxError: invalid parameters to DisplaySetEvent
SyntaxError: no such event class
Error: document *docname* not found by DisplaySetEvent
Info: event created for background

Response (immediate socket)

Success
Failure

Details

Allows you to set events on a document's background. For details of events, see DisplaySetEvent.

Cacheing mode and display events

Regardless of the caching mode, changes to background events take effect immediately.

Revision history

Implemented in WhiskerServer version 2.6.07.

See also

- Display devices
- Event
- DisplayAddObject
- DisplayClearBackgroundEvent
- DisplaySetEvent
- DisplayClearEvent

8.8.9.26 DisplayClearBackgroundEvent

Message

DisplayClearBackgroundEvent *<docname> <eventclass>*

Originator

Client

Response

SyntaxError: insufficient parameters to DisplayClearBackgroundEvent
SyntaxError: invalid parameters to DisplayClearBackgroundEvent
SyntaxError: no such event class
Error: document *docname* not found by DisplayClearBackgroundEvent
Info: event(s) removed

Response (immediate socket)

Success
Failure

Details

The specified event class is cleared for the document's background.
If *eventclass* is blank, all events are cleared for the specified object.

Cacheing mode and display events

Regardless of the caching mode, changes to background events take effect immediately.

Revision history

Implemented in WhiskerServer version 2.6.07.

See also

- Display devices
- DisplaySetBackgroundEvent
- DisplaySetEvent
- DisplayClearEvent

8.8.9.27 DisplaySetEventTransparency

Message

DisplaySetObjectEventTransparency <*docname*> <*objectname*> **on|off**

Originator

Client

Response

SyntaxError: insufficient parameters to DisplaySetObjectEventTransparency
SyntaxError: invalid parameters to DisplaySetObjectEventTransparency
Error: document *docname* not found by DisplaySetObjectEventTransparency
Error: No object called *objectname* found on display document *docname* by DisplaySetObjectEventTransparency
Info: event transparency set to true
Info: event transparency set to false

Response (immediate socket)

Success
Failure

Details

Sets the 'event transparency' property for the chosen object ("on" is transparent, "off' is not). Transparency in this sense does not refer to what you see on the screen. Instead, it refers to the object's behaviour when an object is clicked on with the mouse, touched on a touchscreen, etc. If the object is transparent, it may respond to the event, but will also allow objects *behind* it to respond (until an 'opaque' object is reached). If the object is not transparent, objects behind it are prevented from 'seeing' that their part of the screen has been touched/clicked. By default, objects are *not* transparent.

Cacheing Mode and Display Events
If cacheing mode is enabled, the client may set or clear events, or change event transparency, for objects not yet displayed (that is, *added since* DisplayCacheChanges), or for objects which are awaiting deletion (*deleted since* DisplayCacheChanges) . Commands will take effect **immediately** for any objects currently displayed.

Revision history

Implemented by WhiskerServer version 2.3.

(Bug in documentation before 12 Dec 2022: listed as DisplaySetEventTransparency instead.)

See also

- Display devices
- DisplaySetEvent

8.8.9.28 DisplayEventCoords

Message

DisplayEventCoords on|off

Originator

Client

Response

Info: Event coordinates on
Info: Event coordinates off
SyntaxError: insufficient parameters to DisplayEventCoords
SyntaxError: invalid parameters to DisplayEventCoords

Response (immediate socket)

Success
Failure

Details

When event coordinates are switched on, mouse/touchscreen events are accompanied by their x/y coordinates, to give you full ability to localize your every message coming from the server. Here's what one hypothetical message looks like with and without event coordinates:

```
Event: PictureTouched
Event: PictureTouched 572 827
                      x   y
```

If event coordinates and timestamps are both switched on (see TimeStamps), the event coordinates precede the timestamp, e.g.

```
Event: PictureTouched 572 827 [18376]
                      x   y  timestamp
```

Event coordinates are off by default.

Note. If you want to be informed about touchscreen or mouse events on the document *background*, simply create an appropriately-colour rectangle as an object and send it to the back. The true background will not respond to events. The coordinates reported are the *document*'s coordinates, so are unaffected by display scaling (see *DisplayScaleDocuments* and *DisplaySetDocumentSize*).

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplaySetEvent
- Event

8.8.9.29 DisplayBringToFront

Message

DisplayBringToFront *<docname> <objectname>*

Originator

Client

Response

SyntaxError: insufficient parameters to DisplayBringToFront
SyntaxError: invalid parameters to DisplayBringToFront
Error: document *docname* not found by DisplayBringToFront
Error: No object called *objectname* found on display document *docname* by DisplayBringToFront
Info: object brought to front

Response (immediate socket)

Success
Failure

Details

Brings the object to the front of the document.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayAddObject
- DisplaySendToBack

8.8.9.30 DisplaySendToBack

Message

DisplaySendToBack *<docname> <objectname>*

Originator

Client

Response

SyntaxError: insufficient parameters to DisplaySendToBack
SyntaxError: invalid parameters to DisplaySendToBack
Error: document *docname* not found by DisplaySendToBack
Error: No object called *objectname* found on display document *docname* by DisplaySendToBack
Info: object sent to back

Response (immediate socket)

Success
Failure

Details

Sends the object to the back of the document.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Display devices
- DisplayAddObject
- DisplayBringToFront

8.8.9.31 DisplayKeyboardEvents

Message

DisplayKeyboardEvents <*docname*> **up|down|both|none**

Originator

Client

Response

SyntaxError: insufficient parameters to DisplayKeyboardEvents
SyntaxError: invalid parameters to DisplayKeyboardEvents
SyntaxError: second parameter must be up/down/both/none
Error: document *docname* not found by DisplayKeyboardEvents
Info: keyboard events set

Response (immediate socket)

Success
Failure

Details

Enables or disables keyboard events for the document.

If the second parameter is 'down', pressing a key generates an event. If the second parameter is 'up', only releasing a key generates an event. If the second parameter is 'both', both the pressing and releasing of a key generate events. If it is 'none', then no key events are generated.

Keyboard events are only useful for situations in which a *single* subject is being tested using a *single* display device. Thus, keyboard events are only generated if (1) the client has enabled keyboard events for a document; (2) the document is being displayed in a window; (3) the window has the *input focus*. The authors only envisage this being useful for human testing.

When a keyboard event is generated, the following message is sent to the client:

KeyEvent: <*char*> <*updown*> <*docname*>

where

char is the Windows virtual-key code of the key in question;
updown is "down" if the key was depressed and "up" if it was released;
docname is the name of the document that generated the event.

- Keys pressed while the ALT key is held down are ignored.
- Keyboard *repeat* events are specifically ignored (i.e. if you depress a key, Windows will generate a stream of events, but Whisker will prevent all but the first message from getting to the client – the next message that can be sent is the release of the same key). It's an operant control system, not a word processor.
- Certain keys (e.g. ALT) do not generate events. You cannot request events for *specific* keys (mainly because this would be somewhat tedious; if you feel this is a limitation, contact Rudolf Cardinal at rudolf@pobox.com.)
- You cannot ask the server whether a specific key is up or down at a given instant.
- Obviously, keyboard events can be set up for several documents simultaneously, and/or by several clients, but the Windows *focus* can only be in one window (document) at a time, and events are directed to that document's client.
- The **key codes** are shown in the charts on the following pages.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Keyboard code values
- Display devices
- KeyEvent
- DisplayShowDocument

8.8.9.32 Keyboard code values

The Windows 'virtual-key' (VK) codes are reproduced here (from WINUSER.H). They are given in hexadecimal (in which the letters A–F represent the numbers 10–15); so, for example, the left arrow key (?) generates the code VK_LEFT, which is 0x25 (hexadecimal 25), or (2 × 16) + (5 × 1) = 37 decimal. If you enabled key-down events and the left arrow key was pressed whilst a document named *fish* had the focus, Whisker would send the message

```
KeyEvent: 37 1 fish
```

(Some of the virtual keys may not be available on your keyboard, and a few are processed specially by Windows, so check that an esoteric key functions properly before relying on it.)

We're missing a load of virtual keys… VK_PAGE_UP is not part of the C++ documentation, for example – odd (is in complete set, though). Similarly, VK_BACK_SPACE… etc.

```
/*
 * Virtual Keys, Standard Set
*/
#define VK_LBUTTON        0x01
#define VK_RBUTTON        0x02
#define VK_CANCEL         0x03
#define VK_MBUTTON        0x04   /* NOT contiguous with L & RBUTTON */
#define VK_BACK           0x08
```

```
#define VK_TAB            0x09
#define VK_CLEAR          0x0C
#define VK_RETURN         0x0D
#define VK_SHIFT          0x10
#define VK_CONTROL        0x11
#define VK_MENU           0x12
#define VK_PAUSE          0x13
#define VK_CAPITAL        0x14
#define VK_KANA           0x15
#define VK_HANGEUL        0x15  /* old name - should be here for compatibility */
#define VK_HANGUL         0x15
#define VK_JUNJA          0x17
#define VK_FINAL          0x18
#define VK_HANJA          0x19
#define VK_KANJI          0x19
#define VK_ESCAPE         0x1B
#define VK_CONVERT        0x1C
#define VK_NONCONVERT     0x1D
#define VK_ACCEPT         0x1E
#define VK_MODECHANGE     0x1F
#define VK_SPACE          0x20
#define VK_PRIOR          0x21
#define VK_NEXT           0x22
#define VK_END            0x23
#define VK_HOME           0x24
#define VK_LEFT           0x25
#define VK_UP             0x26
#define VK_RIGHT          0x27
#define VK_DOWN           0x28
#define VK_SELECT         0x29
#define VK_PRINT          0x2A
#define VK_EXECUTE        0x2B
#define VK_SNAPSHOT       0x2C
#define VK_INSERT         0x2D
#define VK_DELETE         0x2E
#define VK_HELP           0x2F

/* VK_0 thru VK_9 are the same as ASCII '0' thru '9' (0x30 - 0x39) */
/* VK_A thru VK_Z are the same as ASCII 'A' thru 'Z' (0x41 - 0x5A) */

#define VK_LWIN           0x5B
#define VK_RWIN           0x5C
#define VK_APPS           0x5D
#define VK_NUMPAD0        0x60
#define VK_NUMPAD1        0x61
#define VK_NUMPAD2        0x62
#define VK_NUMPAD3        0x63
#define VK_NUMPAD4        0x64
#define VK_NUMPAD5        0x65
#define VK_NUMPAD6        0x66
#define VK_NUMPAD7        0x67
#define VK_NUMPAD8        0x68
#define VK_NUMPAD9        0x69
#define VK_MULTIPLY       0x6A
#define VK_ADD            0x6B
#define VK_SEPARATOR      0x6C
#define VK_SUBTRACT       0x6D
#define VK_DECIMAL        0x6E
#define VK_DIVIDE         0x6F
#define VK_F1             0x70
#define VK_F2             0x71
#define VK_F3             0x72
#define VK_F4             0x73
#define VK_F5             0x74
```

```
#define VK_F6             0x75
#define VK_F7             0x76
#define VK_F8             0x77
#define VK_F9             0x78
#define VK_F10            0x79
#define VK_F11            0x7A
#define VK_F12            0x7B
#define VK_F13            0x7C
#define VK_F14            0x7D
#define VK_F15            0x7E
#define VK_F16            0x7F
#define VK_F17            0x80
#define VK_F18            0x81
#define VK_F19            0x82
#define VK_F20            0x83
#define VK_F21            0x84
#define VK_F22            0x85
#define VK_F23            0x86
#define VK_F24            0x87
#define VK_NUMLOCK        0x90
#define VK_SCROLL         0x91
```

See also

- DisplayKeyboardEvents
- KeyEvent
- Display devices

8.8.10 Video objects

Whisker version required

Video support requires **WhiskerServer v4.0** (summer 2011).

In brief, Whisker plays videos using DirectShow. This means it supports any video filetype for which a decoder is available and registered with DirectShow (with a very few caveats as below). **WMV9** (Windows Media Video 9) is supported as the default filetype.

Like any other object, video objects are placed in display documents, which may then be shown on display devices.

To play audio tracks from a video file, the display device must additionally be associated with a (claimed) audio device.

Caveats

There are some important caveats, as follows.

- **Video can be CPU-intensive.** Electing to play audio adds to the CPU load.
- **Flexibility/speed tradeoff, configured in advance.** Because the server may have to show multiple synchronized copies of a given video (e.g. the subject's view and the server copy, and/or multiple subject views), and because it must be possible to add and remove views without pausing the video, even briefly, the server uses a system that requires it to know in advance the maximum number of possible audio and/or video "views" of the video file. This must be specified in *Server – Video configuration*. Unused potential connections use some CPU time; choose the minimum possible value. Because of these limits, DisplayAddObject (option: video) and

DisplayShowDocument can now fail, if they would add too many video views.

- **Timing.** Internally, for each video object, the server has a single "source" filtergraph and potentially multiple "render" filtergraphs (whose maximum numbers must be known when the video objects are created). When a renderer is active, the source graph is bridged to the relevant render graphs. Note that (1) the synchronization between the subject's view and the server view is not guaranteed to be perfect; (2) when a video timestamp is requested, it is taken from the *source* graph; (3) the DirectShow and bridging systems do several things to try to ensure accurate synchronization between the notional timestamp and the audio/video renderers, but this is not an absolute guarantee.
- **Video is incompatible (in its current incarnation) with DirectShow backbuffering;** therefore, in *Configure – Display Devices*, do NOT tick "Use DirectDraw back buffer" if you wish to use video. (If you don't, go ahead and tick it, as it improves static video performance.)
- **Audio streams from video devices must use DirectDraw-supported sound devices** (so they won't work on the "Primary Sound Driver" or "fake" audio devices). Prior to Whisker v4.6, they could not be confined to just one audio channel, so could not be used with left/right "split" devices (see *Configure – Audio Devices*); this limitation is removed as of **Whisker v4.6 (Dec 2014)**.
- **Video objects are always "topmost".** Don't try to place anything on top of video objects; they will be forced down (and if you try to overlap two videos, they will flicker as they each try to go topmost).

More on supported video file types

```
  THESE WORK:
    - video codec type Windows Media Video 9 (and audio codec type
Windows Media Audio 9.2)
      ... known as WMV9
      ... uses the "WMVideo Decoder DMO" filter and works
    - things encoded with video codec "wmv2" and audio codec "wmav2" by
ffmpeg (see below)
  SOME FILES WON'T LOAD, and say so. Convert them (see below).
  SOMETIMES AUDIO FAILS:
    - in which case the server log might say e.g. "Couldn't use bridge
to create source graph - DirectX error in CSourceGraph::CSourceGraph:
Cannot play back the audio stream: the audio format is not supported."
    - use the -noaudio flag in this case
  SOME FILES LOAD AND DON'T COMPLAIN BUT VIDEO PLAYS AS A BLANK SCREEN.
    - example: video codec type Windows Media Screen V7 (and audio codec
type Windows Media Audio V8)
      ... fail because the VMR9 renderer insists on a colour space
converter when used
              from an infinite tee, and the "WMV Screen decoder DMO"
filter used by this format
              won't play through a CSC (as tested in GraphBuilder),
              unlike the "WMVideo Decoder DMO" filter
  TO INSPECT WMV VIDEO CODE TYPES:
    - run Windows Media Player; load file; then File > Properties
  GENERAL PROCEDURE TO MAKE SOMETHING WORK:
    - Install ffmpeg.
      On a Ubuntu/Debian Linux box: sudo apt-get install ffmpeg
      On a Windows box: see http://www.ffmpeg.org/
    - ffmpeg -i INPUTVIDEO.XXX -vcodec wmv2 -acodec wmav2 OUTPUT.wmv
    - This also allows conversion from FLV, MPEG, etc.
    - Or with the newer avconv system:
```

```
      avconv -i INPUTVIDEO.XXX -c:v wmv2 -c:a wmav2 OUTPUT.wmv
    - In the case of a video with no audio, or to remove audio, you can
do:
      avconv -i INPUTVIDEO.XXX -c:v wmv2 -an OUTPUT.wmv
```

Video configuration on the server

See *Server – Video configuration*

Video commands

Video objects are controlled using these conventional object-manipulation commands:

- DisplayAddObject (option: video)
- DisplayDeleteObject
- DisplaySetEvent
- DisplayClearEvent
- DisplaySetEventTransparency
- DisplayEventCoords
- DisplayBringToFront
- DisplaySendToBack

... and these video-specific commands:

- DisplaySetAudioDevice
- VideoPlay
- VideoPause
- VideoStop
- VideoSeekAbsolute
- VideoSeekRelative
- VideoGetTime
- VideoGetDuration
- VideoTimestamps
- VideoSetVolume

8.8.10.1 DisplayAddObject: video

Message

DisplayAddObject *<docname>* *<objectname>* **video** *<x>* *<y>* *<filename>* **[-loop | -noloop] [-wait | -playimmediate | -playwhenvisible] [-width** *x***] [-height** *y***] [-audio | -noaudio] [-top | -middle | -bottom] [-left | -centre | -right] [-backcolour** *<red>* *<green>* *<blue>***]**

Originator

Client

Response

General:

SyntaxError: insufficient parameters to DisplayAddObject
SyntaxError: invalid parameters to DisplayAddObject

Error: document *docname* not found by DisplayAddObject
Error: unknown object type

Video:
SyntaxError: insufficient parameters to DisplayAddObject / video
SyntaxError: invalid parameters to DisplayAddObject / video
SyntaxError: Video given invalid size (DisplayAddObject/bitmap)
Error: DisplayAddObject not possible, would create too many video views for *document*, use fewer or reconfigure WhiskerServer
Error: DisplayAddObject not possible, current views for *document* don't support video, reconfigure WhiskerServer
Error: failed to create video object
Error: failed to load video file "*filename*"
Info: added video "*filename*" to display document *docname*

Response (immediate socket)

Success
Failure

Details

Adds a video object.

Defaults: -noloop -wait -width -1 -height -1 -audio -left -top

- *x* = *x*-coordinate of (by default) the top left point; the meaning of these coordinates can be changed by the vertical/horizontal alignment options below
- *y* = *y*-coordinate of (by default) the top left point; the meaning of these coordinates can be changed by the vertical/horizontal alignment options below
- *filename* = video filename, e.g. c:\test.wmv
- **-loop** makes the video restart when it finishes
- **-noloop** lets the video stop when it's finished
- **-wait** loads the video but does not play it
- **-playimmediate** begins playing the video immediately, even if there are no views, or the server is in the process of setting up the view
- **-playwhenvisible** begins playing when the first view (subject view, not server view) is showing the video object
- **-width** *x* = width to force the video to (specify height and width as -1 to use the native size, or omit)
- **-height** *y* = height to force the video to (specify height and width as -1 to use the native size, or omit)
- **-audio** allows audio to be played if DisplaySetAudioDevice has been set up *before* this video is loaded
- **-noaudio** disables audio
- **–top | –middle | –bottom** determines the vertical alignment (default is top). The top, middle (vertical centre), or bottom of the bitmap can be aligned to the *y* coordinate.
- **–left | –centre | –right** determines the horizontal alignment (default is left). The left edge, centre, or right edge of the bitmap can be aligned to the *x* coordinate.
- **-backcolour** sets the background colour of the video area when no video is being played or paused.

Restrictions on looping with very short videos

When a looping video ends, the main (GUI) thread restarts it. For very brief or single-frame videos, this causes excessive load on the GUI thread, so WhiskerServer appears to grind to a halt. The load is arbitrarily high: the video is started, stops immediately, says "restart me...", and so all spare GUI thread time is consumed. **Therefore,** Whisker will disable looping on any video shorter than (arbitrarily) **0.5 s**.

What could you do instead? Well, you could (a) use a still image, if you're trying to use a single-frame video as a still image; (b) edit your video file yourself, repeating it until it's at least at the threshold length, and

then Whisker will loop it happily.

Revision history

Looping restriction in WhiskerServer v4.5.
Implemented in WhiskerServer version 4.0.
Vertical/horizontal alignment options added in v4.2.1.
Background colour option added in v4.6.1.

See also

- Video objects
- Server / video configuration

8.8.10.2 DisplaySetAudioDevice

Message

DisplaySetAudioDevice *<displaydevice> <audiodevice>*

Originator

Client

Response

SyntaxError: insufficient parameters to DisplaySetAudioDevice
SyntaxError: invalid parameters to DisplaySetAudioDevice
SyntaxError: second parameter must be up/down/both/none
Error: Invalid device/line/client number or alias sent to DisplaySetAudioDevice
Error: Audio device *AAA* (*XXX*) not the DirectSound variety, use another
Info: Display device *DDD* using audio device *AAA* (DirectX name *XXX*) for video playback

Response (immediate socket)

Success
Failure

Details

Associates a given display device with a given audio device, so that audio, if chosen, can be played from video files.

Not all audio devices will support this: in particular, the device must be (a) real, not fake (to get round this in a development environment, and to force audio off, do so by configuring the server in Server / Video configuration); (b) a DirectX device (thereby excluding the "Primary Sound Driver" - choose its DirectX equivalent) ; (c) not a left/right-"split" device.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects
- Server / video configuration

8.8.10.3 VideoPlay

Message

VideoPlay *<document> <videoobject>*

Originator

Client

Response

SyntaxError: insufficient parameters to VideoPlay
SyntaxError: invalid parameters to VideoPlay
Error: Document *docname* not found by VideoPlay
Error: No video object called *videoobject* found on display document *docname* by VideoPlay
Info: Video started

Response (immediate socket)

Success
Failure

Details

Starts playing a video.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects

8.8.10.4 VideoPause

Message

VideoPause *<document> <videoobject>*

Originator

Client

Response

SyntaxError: insufficient parameters to VideoPause
SyntaxError: invalid parameters to VideoPause
Error: Document *docname* not found by VideoPause
Error: No video object called *videoobject* found on display document *docname* by VideoPause
Info: Video paused

Response (immediate socket)

Success
Failure

Details

Pauses a video.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects

8.8.10.5 VideoStop

Message

VideoStop <*document*> <*videoobject*>

Originator

Client

Response

SyntaxError: insufficient parameters to VideoStop
SyntaxError: invalid parameters to VideoStop
Error: Document *docname* not found by VideoStop
Error: No video object called *videoobject* found on display document *docname* by VideoStop
Info: Video stopped

Response (immediate socket)

Success
Failure

Details

Stops a video, and rewinds it to the beginning.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects

8.8.10.6 VideoSeekAbsolute

Message

VideoSeekAbsolute <*document*> <*videoobject*> <*time_ms*>

Originator

Client

Response

SyntaxError: insufficient parameters to VideoSeekAbsolute
SyntaxError: invalid parameters to VideoSeekAbsolute
Error: Document *docname* not found by VideoSeekAbsolute
Error: No video object called *videoobject* found on display document *docname* by VideoSeekAbsolute
Info: Video seek performed

Response (immediate socket)

Success
Failure

Details

Seek-positions a video to an absolute time, in milliseconds (without altering its play mode, i.e. if it was playing, it will continue playing from the new point; if it was stopped/paused, it will remain stopped but when you play it, it will resume from the new point). Negative times are converted to 0 (the start); times beyond the end are converted to the end time.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects

8.8.10.7 VideoSeekRelative

Message

VideoSeekRelative *<document> <videoobject> <time_ms>*

Originator

Client

Response

SyntaxError: insufficient parameters to VideoSeekRelative
SyntaxError: invalid parameters to VideoSeekRelative
Error: Document *docname* not found by VideoSeekRelative
Error: No video object called *videoobject* found on display document *docname* by VideoSeekRelative
Info: Video seek performed

Response (immediate socket)

Success
Failure

Details

Seek-positions a video to an relative time, in milliseconds (without altering its play mode, i.e. if it was

playing, it will continue playing from the new point; if it was stopped/paused, it will remain stopped but when you play it, it will resume from the new point).

Negative times seek backwards; positive times seek forwards.

Resulting absolute times are truncated to the start or end of the video.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects

8.8.10.8 VideoGetTime

Message

VideoGetTime *<document> <videoobject>*

Originator

Client

Response

SyntaxError: insufficient parameters to VideoGetTime
SyntaxError: invalid parameters to VideoGetTime
Error: Document *docname* not found by VideoGetTime
Error: No video object called *videoobject* found on display document *docname* by VideoGetTime
Info: Document *docname* video *videoobject* time_position: *time_ms*

Response (immediate socket)

VideoTime *<time_ms>*
Failure

Details

Queries the current source graph playback position, in milliseconds.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects

8.8.10.9 VideoGetDuration

Message

VideoGetDuration <*document*> <*videoobject*>

Originator

Client

Response

SyntaxError: insufficient parameters to VideoGetDuration
SyntaxError: invalid parameters to VideoGetDuration
Error: Document *docname* not found by VideoGetDuration
Error: No video object called *videoobject* found on display document *docname* by VideoGetDuration
Info: Document *docname* video *videoobject* duration: *time_ms*

Response (immediate socket)

Duration <*time_ms*>
Failure

Details

Queries the video duration, in millseconds.

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Video objects

8.8.10.10 VideoTimestamps

Message

VideoTimestamps on|off

Originator

Client

Response

SyntaxError: insufficient parameters to VideoTimestamps
SyntaxError: invalid parameters to VideoTimestamps
Info: VideoTimestamps on|off

Response (immediate socket)

Success
Failure

Details

Turns video timestamps on/off.

Allows you to have the video source graph's clock time, in milliseconds, sent along with events when a video object is touched.

Affects Event messages (q.v.).

Revision history

Implemented in WhiskerServer version 4.0.

See also

- Event
- Video objects

8.8.10.11 VideoSetVolume

Message

VideoSetVolume *<document> <videoobject> <volume>*

Originator

Client

Response

SyntaxError: insufficient parameters to VideoSetVolume
SyntaxError: invalid parameters to VideoSetVolume
Error: Document *docname* not found by VideoSetVolume
Error: No video object called *videoobject* found on display document *docname* by VideoSetVolume
Info: Video volume set

Response (immediate socket)

Success
Failure

Details

Sets the audio volume associated with a video. Volumes range from 100 (full) to 0 (minimum), as for AudioSetSoundVolume.

Revision history

Implemented in WhiskerServer version 4.6.

See also

- Video objects

8.8.11 Server-based device names and device definition files

Clients may claim devices (displays, audio devices, digital I/O lines, etc.) by *number*, and clients may assign useful *aliases* to these devices. Additionally, the server knows about a set of 'standard' *device names* for the equipment it is controlling.

Server-based device names bring the following advantages:

- *Ease of client programming*. Clients usually know that they need (for example) a pellet dispenser and a lever in order to function, but they don't care whether the lever is on line 52 or line 87. Server-based names take this chore away from the client.
- *Client portability*. If a client refers to its lines by name only, then it can run on a computer that has the same devices (device names), but wired to different lines.
- *Control of device groupings.* A typical experimental computer might have six operant chambers, each with a pellet dispenser and a lever. If the client decides it wants to use box 4, and can simply say to the server 'I'd like the following devices… for box 4', it simplifies client programming.
- *Exclusion of other clients using device groupings.* The client can prevent others from taking control of devices in the same chamber. For example, if our client uses the pellet dispenser and the lever, but (unbeknownst to it) the chamber also has a clicker, a different client could control the clicker. This might be useful to you, the experimenter (perhaps you want to superimpose a noncontingent click upon your behavioural task without reprogramming that task) but it might be a nuisance (you've written a program to set all your boxes' clickers going, but you don't want someone to run this program accidentally and disturb your IV self-administration task). To avoid this, Whisker allows you to claim *all* the devices for a particular *device group* (a device group typically represents an operant chamber) – thus temporarily preventing any other client from using devices in that chamber.

The server's device definition file

The device definition file is typically a text (.TXT) file that you edit using a standard text editor. A sample file is provided, and is shown below. Note that the first line must match the sample's *exactly*, as WhiskerServer reads this line as a quick check that it has found the correct sort of file.

```
WhiskerServer v2.0 - DEVICE DEFINITION FILE - DO NOT ALTER THIS LINE
######################################################################
# This file defines device names used by the WhiskerServer program.
# Lines beginning with a hash (#) are comments and are ignored.
#
# Each line takes the following format:
#
#  <device_type> <device_number> <group_name> <device_name>
#
# where <device_type> may be
#  line          = digital I/O line
#  display       = display device (monitor)
#  audio         = audio device (sound card, or half-sound card; see manual)
#
# The <device_number> is the number of the line/display/audio device that you
see
# on the server's console - the number that you would otherwise claim.
#
# The COMBINATION of the <group_name> and <device_name> must be unique.
# If the server encounters non-unique device group/name pairs in this file,
# all but the first will be ignored.
# Neither the <group_name> nor the <device_name> may start with a number.
line            0            box1          LEFTLEVER
```

```
line            0           box1        ALTERNATIVE_NAME
line            1           box1        RIGHTLEVER
line            2           box2        LEFTLEVER
line            3           box2        RIGHTLEVER
line            7           box1        LEFTLIGHT
line            8           box1        RIGHTLIGHT
line            9           box2        LEFTLIGHT
line            10          box2        RIGHTLIGHT
display         0           human       display
audio           0           box1        leftsound
audio           1           box1        rightsound
audio           2           box2        leftsound
audio           3           box3        rightsound
```

Device names may not start with a number. This allows WhiskerServer to distinguish whether a client is talking about a device number or a name.

You may give one device more than one name, as long as the {device group + device name} combination remains unique. (Only one of the names will appear on the server's console displays.) This feature allows you, for example, to run a task that requires a clicker in an operant chamber that doesn't have one, but has a spare light. Simply add the device name that the task expects to exist (e.g. 'CLICKER') as an alternative name for your light. (Beware, though: if the task also uses the light, it may perform strangely!)

When you have a definition file that is suitable for your equipment, tell WhiskerServer where it is (see Set server device definition file).

When the server starts, the following entries appear in its event log:

```
Reading device definition file.
All device names read. Found 13 device names.
```

You may want to create 'fake' lines to support tasks that would not otherwise run on your equipment. For example, if your task requires a device named PUMP, and you don't have a pump but don't mind running the task without one, then install some fake lines and edit your device definition file so that one of the fake lines has the name PUMP in the appropriate device group.

Ways to claim devices using their names

Whenever you claim an individual device using a server-based device name, you must specify it by its *group* and *device name*. The following commands support this:

- **LineClaim**
- **AudioClaim**
- **DisplayClaim**

Additionally, you may claim whole groups, using the **ClaimGroup** command.

See also

- LineClaim
- AudioClaim
- DisplayClaim
- ClaimGroup
- Device names and aliases

- Fake I/O lines

8.8.11.1 ClaimGroup

Message

ClaimGroup <*groupname*> **[–prefix <***prefix***>] [–suffix <***suffix***>]**

Originator

Client

Response

You may receive the following failure messages:

SyntaxError: insufficient parameters to ClaimGroup
Error: no devices found/successfully claimed in group <*groupname*>

or a series of messages of the format

Info: claimed device <*groupname*>/<*devicename*> = line|display|audio <*devicenumber*> (alias <*prefix*><*devicename*><*suffix*>)
Info: claimed device <*groupname*>/<*devicename*> = line|display|audio <*devicenumber*> (ALIAS NOT SET)
Error: could not claim device <*groupname*>/<*devicename*> = line|display|audio <*devicenumber*>

For example, the command

ClaimGroup box1 –prefix mybox_ –suffix _fish

might generate the response

Info: ClaimAccepted: Line 5 For Input As 5 (ALIAS NOT SET)= box1/RIGHTLEVER
Check this, has MRFA altered it?
Info: claimed device = line 1 (ALIAS NOT SET)
Info: claimed device box1/LEFTLIGHT = line 7 (alias mybox_LEFTLIGHT_fish)
Info: claimed device box1/RIGHTLIGHT = line 8 (alias mybox_RIGHTLIGHT_fish)
Error: could not claim device box1/leftsound = audio 0
Info: claimed device box1/rightsound = audio 1 (alias mybox_rightsound_fish)

if audio device 0 was already claimed by another client.

Additional messages may also be generated as the aliases are created.

Response (immediate socket)

Success	*(implies every device was claimed successfully)*
Failure	*(either there were none to claim, or at least one failed to be claimed)*

Details

Attempts to claim every device in the group *groupname*. This will include lines, display devices, and audio devices. Each device will be assigned an alias that consists of the device name, with a user-specified prefix and/or suffix. If you don't like this alias, you are of course free to add others of your liking! If the unprefixed alias would interfere with your program's function, simply add a prefix like 'junk_'. When lines

are claimed, the default reset state (see LineClaim) is 'Leave'.

Note that it is perfectly valid to claim an entire device group with ClaimGroup, ensuring that no other clients can influence that device group, and then to go through a specific subset of devices issuing individual LineClaim, AudioClaim, and DisplayClaim commands to verify that every device you expect to be present, is.

Revision history

Implemented by WhiskerServer version 2.3.

See also

- Device names and aliases

8.8.11.2 Device names are not the same as aliases

Only the *claiming* commands (LineClaim, AudioClaim, DisplayClaim, ClaimGroup) know about server-defined device names. All other commands only know about *aliases* that you have defined.

Why? To prevent changes in the device definition file affecting clients adversely. (Take an example: a client claims output lines by number, claims line 5, which is a pellet dispenser, and gives it the alias REINFORCER. Let's say line 5 is listed in the definition file with the device name PELLET. If someone plugs in an IV infusion pump on line 10 and gives it the name REINFORCER, not knowing that the client uses that alias internally, the client might start to operate the IV infusion pump as well as the pellet dispenser.)

LineClaim, AudioClaim, and **DisplayClaim** do *not* create aliases automatically, so **you should use the `-alias` parameter to these commands** to ensure that you have an alias.

Scenario:

```
ClaimGroup box2
```

ensures that box 2 is proof against outside interference by claiming all devices in group 'box2'

```
LineClaim box2 leftlever -input -alias leftlever
LineClaim box2 rightlever -input -alias rightlever
LineClaim box2 pellet -output -alias pellet
```

specifically ensures that box 2 has two levers and a pellet dispenser by re-claiming them.

See also

- LineClaim
- AudioClaim
- DisplayClaim
- ClaimGroup
- Server-based device names and device definition files

8.8.12 Time stamps

For some behavioural tasks, it is important to know the exact timing of events. While a client has the option of establishing for itself the time when an event (or other message) *arrives* from the server, it will be more accurate to have the server report the time at which it *sent* the message. This can be done by adding a *timestamp* to every message coming from the server. The commands detailed in this section support this feature:

- TimeStamps
- ResetClock

8.8.12.1 TimeStamps

Message

TimeStamps on|off

Originator

Client

Response

Info: Timestamps on [<*timestamp*>]
Info: Timestamps off
SyntaxError: invalid parameters to TimeStamps
SyntaxError: insufficient parameters to TimeStamps

Response (immediate socket)

Success
Failure

Details

When timestamps are switched on, *every* message coming from the server has a timestamp added. Here's what one hypothetical message looks like with and without a timestamp:

Event: Nosepoke
Event: Nosepoke [43827]

Appended to the original message is a space, '[', a number, and ']'. The number is the system time in milliseconds (ms).

The server never sends '[' or ']' except to mark a timestamp – so you may want to avoid using '[' as part of your event names (because you'll have to distinguish between your event and the timestamp).

Example: reaction times

Imagine your subject is trained to respond to a light stimulus by pressing a lever. You want to know the reaction time between turning the light on and the lever being pressed. Consider this (responses from the server are shown in blue, commands from the client in red):

LineSetEvent Lever on lever_pressed
Info: LineEventCreated: 8 on
TimeStamps On

Info: Timestamps on [429458]
LineSetState Light on (*)
Success [429682] (†)
...
Event: lever_pressed [429912]

(*) This command was sent through the immediate socket in order to get a response to the LineSetState command.
(†) This response therefore arrives via the immediate socket.

The reaction time is (429912 ms – 429682 ms) = 230 ms.

This is the most accurate way to measure reaction times, because network latencies do not affect the measurement. (Although a network delay in the LineSetState call will delay the time at which the light goes on, the lever press is accurately measured relative to the moment the light is turned on. Network delays in server-to-client transmission do not matter because time measurement is only being performed on the server.)

Revision history

Implemented in WhiskerServer version 1.

See also

- Time stamps

8.8.12.2 ResetClock

Message

ResetClock

Originator

Client

Response

Info: Clock reset
SyntaxError: invalid parameters to ResetClock

Response (immediate socket)

Success

Details

This command supplements the timestamp feature. If you turn timestamps on, the server stamps messages with the system time – the time (in ms) since the server's computer was started. This will often be a large number (it's a 32-bit unsigned integer) and not very meaningful. If you issue the ResetClock command, the server resets the clock for your client to zero.

Revision history

Implemented in WhiskerServer version 1.

See also

- Time stamps

8.8.13 Status information and niceties

8.8.13.1 Version

Message

Version

Originator

Client

Response

Info: Version: *<version>*

Response (immediate socket)

<version>

Details

The server responds with a version number. If future versions of the server support an extended command set, the Version command can be used to determine the capabilities of the server.

Revision history

Implemented by WhiskerServer version 2.3.

8.8.13.2 ClientNumber

Message

ClientNumber

Originator

Client

Response

Info: You are client number *<number>*

Response (immediate socket)

<number>

Details

The server responds with the client number.

Revision history

Implemented by WhiskerServer version 2.3.

8.8.13.3 WhiskerStatus

Message

WhiskerStatus [-lines] [-clients] [-events] [*<clientnumber(s)>*]

Originator

Client

Details

Causes the server to send a whole bunch of status info.
Default: general server status report, with help info on syntax.
Clients: describes registered clients.
Lines: reports status and ownership of all lines.
Events: reports all pending line/timer events
<*Clientnumber*>: restricts reports to numbered client(s). If *clientnumber* is '*', refers to the calling client.

Thus a program interested in the status of all boxes on a system would issue

```
WhiskerStatus -clients
```

while a client wanting a report on its own lines might issue

```
WhiskerStatus -lines *
```

If you send this request down the immediate socket, the response **Failure** will be given.

This command is not fully implemented. At the moment, it always gives the same general status report.

Revision history

Implemented in WhiskerServer version 1.

See also

- ReportName
- ReportStatus

8.8.13.4 ReportName

Message

ReportName *<text>*

Originator

Client

Response

No reply.

Response (immediate socket)

Success

Details

Tells the server the client's name.

Example

ReportName Pavlovian-to-instrumental transfer, v. 26-Aug-99

Revision history

Implemented in WhiskerServer version 1.

See also

- WhiskerStatus

8.8.13.5 ReportStatus

Message

ReportStatus *<text>*

Originator

Client

Response

No reply.

Response (immediate socket)

Success

Details

Tells the server what the client's up to. *Text* is the status report that will be reported on the server screen and is available to programs like WhiskerStatus.

Example

ReportStatus rat 6 has made 15 responses

Revision history

Implemented in WhiskerServer version 1.

See also

- WhiskerStatus

8.8.13.6 ReportComment

Message

ReportComment *<text>*

Originator

Client

Response

No reply.

Response (immediate socket)

Success

Details

Does nothing, but the message will enter the server's communication log for this client, if active. Thus, this is a debugging facility.

Example

ReportComment About to start reinforcer

Revision history

Implemented in WhiskerServer version 4.6.

8.8.13.7 TestNetworkLatency

Message

TestNetworkLatency

Originator

Client

Details

If the client sends TestNetworkLatency, the server will send Ping. The client should respond with PingAcknowledged as soon as possible. The server will calculate the time difference between sending the Ping and receiving the PingAcknowledged and report it in the following manner:

Info: Network latency is 15 ms

Do not send multiple TestNetworkLatency messages very fast, as the server doesn't keep track of multiple outstanding Pings. You will get an abnormally high value as the server matches the PingAcknowledged with a much earlier Ping, or something like that.

If the client wants to test the network latency itself, it can send a Ping; the server will respond immediately

with a PingAcknowledged. The TestNetworkLatency command just saves the client the effort of calculating the time difference.

The Whisker test client supports Ping/PingAcknowledged, as does the C++ client library and the SDK.

These commands are supported on the immediate socket. That is to say, whichever socket you send the command on, the following replies are received on the same socket:

TestNetworkLatency → Ping

Ping → PingAcknowledged

PingAcknowledged → Info: Network latency is *<latency>* ms [on the main socket]
or PingAcknowledged → *<latency>* [on the immediate socket]

Revision history

Implemented in WhiskerServer version 1.

See also

- Ping

8.8.13.8 RequestTime

Message

RequestTime

Originator

Client

Response

Info: Time: *<time>*

Response (immediate socket)

<time>

Details

Asks for a Time message to be sent giving the system time (on the server) in milliseconds. If you want to use this timing information (e.g. for latency calculation) and your client language can't read the system time sufficiently accurately, you can use this feature of the server.

Comments

This is the actual system time, *not* the clock that is reset by the ResetClock command. Thus, you can have the following exchange with the server:

TimeStamps on
Info: Timestamps on [200952019]
ResetClock

Info: Clock reset [0]
RequestTime
Info: Time: 200965075 [6943]

Revision history

Implemented in WhiskerServer version 1.

8.8.13.9 Echo

Message

Echo *stuff*

Originator

Client

Response

stuff

Response (immediate socket)

stuff

Details

The server replies with the contents sent.

Comments

This is for testing purposes only, to saturate network communications with a sequence of messages.

Revision history

Implemented in WhiskerServer version 4.6.2.

8.8.14 Client-client communications

Clients may send messages to each other, if this feature is enabled on the server and the recipient permits messages to be sent to it. The following commands control this feature:

- PermitClientMessages
- SendToClient
- ClientMessage

8.8.14.1 PermitClientMessages

Message

PermitClientMessages on|off

Originator

Client

Response

Info: permitting messages from other clients
Info: refusing messages from other clients
SyntaxError: insufficient parameters to PermitClientMessages
SyntaxError: invalid parameters to PermitClientMessages

Response (immediate socket)

Success
Failure

Details

Chooses whether your client will accept incoming messages from other clients. (The server must also permit client-client messages for any actually to arrive.)

The default is not to permit client messages.

Revision history

Implemented in WhiskerServer version 2.0.

See also

- Client-client communications

8.8.14.2 SendToClient

Message

SendToClient *<clientnum> <message>*

Originator

Client

Response

Error: client number is invalid
Error: server is not permitting client-client communications
Error: client <*clientnum*> not found
Error: client <*clientnum*> not accepting messages
Error: no message to send
Info: message successfully sent to client <*clientnum*>
SyntaxError: insufficient parameters to SendToClient
Warning: Sending a client message to yourself!
Warning: No listening clients receiving broadcast message!

Response (immediate socket)

Success
Failure

Details

If successful, causes the generation of a **ClientMessage** message to the destination client (q.v.).

For a message to be sent, all of the following conditions must be met:
- The destination client number must be valid. **Use the client number -1 to broadcast to all other clients.**
- The server must allow client–client communications (determined on the server console menu).
- The receiving client must permit incoming client–client messages (determined by the RefuseClientMessages and PermitClientMessages commands)

Revision history

Implemented in WhiskerServer version 2.0.

See also

- Client-client communications

8.8.15 Client authentication

Special editions of WhiskerServer exist that permit only certain clients to use them fully.
For example, there is an edition of Whisker for CANTAB and MonkeyCantab, sold by Campden Instruments Ltd http://www.campden-inst.com/
This edition of the server permits no other clients to use it; users must purchase a full version of the server to use other clients.

These special clients authenticate themselves using the following command protocols:

- **Authenticate**
- **AuthenticateChallenge**
- **AuthenticateResponse**

A non-authenticated client using one of these special editions of the server will see this error message:

SyntaxError: Not permitted - provide authentication first

if they try to manipulate devices.

8.8.15.1 Authenticate

Message

Authenticate *package clientname*

Originator

Client

Response

AuthenticateChallenge: *challengetext*
SyntaxError: insufficient parameters to Authenticate
SyntaxError: invalid parameters to Authenticate

Response (immediate socket)

AuthenticateChallenge: *challengetext*
Failure

Details

Initiates an authentication attempt. **Applicable only to editions of the server that only accept authenticated clients.**
The server responds with a AuthenticateChallenge command.
Authentication is completed when the client responds to the AuthenticateChallenge command with a valid **AuthenticateResponse** command.

Revision history

Implemented in WhiskerServer version 2.6.8.

See also

- Client authentication
- AuthenticateChallenge
- AuthenticateResponse

8.8.15.2 AuthenticateChallenge

Message

AuthenticateChallenge *challengetext*

Originator

Server

Response

The client should respond with an AuthenticateResponse command.

Details

Stage 2 of an authentication attempt. **Applicable only to editions of the server that only accept authenticated clients.**
Authentication is completed when the client responds to the AuthenticateChallenge command with a valid **AuthenticateResponse** command.

Revision history

Implemented in WhiskerServer version 2.6.8.

See also

- Client authentication
- Authenticate
- AuthenticateResponse

8.8.15.3 AuthenticateResponse

Message

AuthenticateResponse *responsetext*

Originator

Client

Response

Authenticated
Error: authentication failed
SyntaxError: insufficient parameters to AuthenticateResponse
SyntaxError: invalid parameters to AuthenticateResponse

Response (immediate socket)

Success
Failure

Details

Stage 3 of an authentication attempt. **Applicable only to editions of the server that only accept authenticated clients.**
If this command succeeds, the client is authenticated by the server.

Revision history

Implemented in WhiskerServer version 2.6.8.

See also

- Client authentication
- Authenticate
- AuthenticateChallenge

8.8.16 Secure server data logs

WhiskerServer allows clients to record events, communications, and/or their own message to disk via Whisker while they are running, and Whisker authenticates these logs cryptographically, appending a digital signature.

See:

- **The logging command set**
- Permanence of data recording
- Digital signing of data logs
- Verifying digitally-signed logs

8.8.16.1 Logging command set

- **LogOpen**
  Opens a log file.
- **LogPause**
  Pauses logging.
- **LogResume**
  Resumes logging.
- **LogSetOptions**
  Chooses what to log.
- **LogWrite**
  Records your own message to the log.
- **LogClose**
  Closes the log. In the Industrial Edition of Whisker, the log will be digitally signed at this point.

8.8.16.1.1 LogOpen

Message

LogOpen *<filename>*

Originator

Client

Response

Info: log opened
SyntaxError: insufficient parameters to LogOpen
SyntaxError: invalid parameters to LogOpen
Error: file exists or cannot open file
Error: log already open, close that one first

Response (immediate socket)

Success
Failure

Details

Opens a text log. Each Whisker client can only have one primary log file open at once. That log can be used to record significant events, client–server communications, and arbitrary status reports from the client. All entries are date- and time-stamped.

The log is started as soon as it is opened.

Logs have the following output format:

```
SystemDate_YMD,SystemTime_HMS,Time_ms,Source,Text
2001-11-05,21:40.48,40927,Log,LOG_OPENED
2001-11-05,21:40.48,40927,Log,Recording options: events off keyevents off
clientclient off comms off signature on
2001-11-05,21:40.48,40927,Server,Info:...
```

```
2001-11-05,21:40.48,40927,Server,Info:...
2001-11-05,21:40.48,40927,Server,ImmPort: 1334
2001-11-05,21:40.48,40927,Server,Code: 0_41
2001-11-05,21:40.48,49046,Client-IMM,ClaimGroup box0
2001-11-05,21:40.48,49047,Server-IMM,Failure
2001-11-05,21:40.48,49049,Client,RelinquishAllLines
2001-11-05,21:40.48,49049,Server,Info:...
...
2001-11-05,21:40.48,600,Client-MSG,Task beginning, score 0.
2001-11-05,21:40.48,10204,Server,Event: Lever_pressed
```

The *SystemDate* and *SystemTime* field are the server's computer clock; the *Time_ms* field is the timestamp clock (see *TimeStamps*); the *Source* field indicates the source of the log entry; the *Text* field contains the entry itself.

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.16.1.2 LogPause

Message

LogPause

Originator

Client

Response

Info: logging paused
Error: no log file open
SyntaxError: invalid parameters to LogPause

Response (immediate socket)

Success
Failure

Details

This command can be used to pause. LogResume will restart the log. The fact of pausing and restarting will be recorded in the log:

```
2001-11-05,21:40.48,10204,Log,LOG_PAUSED
...
2001-11-05,21:40.57,19204,Log,LOG_RESUMED
```

Comments

In an industrial or governmental setting, pausing the log may be inadvisable as it weakens the data trail.

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.16.1.3 LogResume

Message

LogResume

Originator

Client

Response

Info: logging resumed
Error: no log file open
SyntaxError: invalid parameters to LogResume

Response (immediate socket)

Success
Failure

Details

See LogPause.

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.16.1.4 LogSetOptions

Message

LogSetOptions [–events on|off] [–keyevents on|off] [–clientclient on|off] [–comms on|off] [-signature on|off]

Originator

Client

Response

Info: Recording options: events on|off keyevents on|off clientclient on|off comms on|off signature on|off
SyntaxError: insufficient parameters to LogSetOptions
SyntaxError: invalid parameters to LogSetOptions

Response (immediate socket)

Success
Failure

Details

You may choose what class of events are written to the log:

Explicit messages from the client (sent with LogWrite) are always recorded.
–events: Event messages are recorded.

–keyevents: KeyEvent messages are recorded.
–clientclient: Client-to-client (ClientMessage:) communications are recorded.
–comms: All client–server communications are written (including events). **Be warned:** this can create large logs.
–signature: The log is digitally signed when it is closed. **Be warned:** this can pause the system briefly if the log is very long (see below).

At least one option must be specified. The new logging options are written to the log in full. For example:

```
2001-11-05,21:40.48,40927,Log,Recording options: events on keyevents off
clientclient on comms on signature off
```

By default, signatures are on and all other options are off. The settings from LogSetOptions persist if you close the log and open another, or if you issue the command before opening a log.

PERFORMANCE NOTES (technical)

1. If a client logs data at a very high rate, it may impact upon the system's performance. The act of writing to a disk log is conducted by the same thread that processes the same client's network communications. Therefore, if this thread spends a great deal of time writing to disk, it will impact the performance of the client whose log it is before it impacts the performance of the rest of the server (or of other clients) - though this will happen insofar as the whole computer starts to run slowly.
2. When a client closes itself, the client's thread digitally signs its log before the client is removed from the server. This may take a few moments if the log is large. After that, it informs the server that the client should be removed. Removal of the client from the system is performed by the server's main (user interface) thread. **However,** if the user deletes a client manually from the server's console, it will be the server's user interface thread that has to sign the log; consequently, the user interface may pause (and an hourglass be shown) while the server signs the log. This may also pause ongoing processing of all displays (since display processing is also handled by the server's user interface thread). **Motto:** don't delete clients by hand without thinking first.
3. **When a client signs a log, the intensity of processing is such that brief (e.g. 60 ms) pauses may occur in polling.** This is an unfortunate compromise; it is not possible, within one process (WhiskerServer.exe), to separate the thread priorities for log-signing and polling to a great enough extent. The choices are either (a) the signing thread priority is higher than optimal - polling continues perfectly and the computer freezes during signing, to the extent that the mouse doesn't work; (b) the polling priority is briefly lower than optimal - the computer remains perfectly usable and polling suffers slightly during signing. I've (RNC) chosen (b), as people always seem to get upset when their computers freeze for unknown periods of time. **If you cannot afford even this pause, turn the signature off.**

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.16.1.5 LogWrite

Message

LogWrite *<message>*

Originator

Client

Response

No response if successful (even if the log is paused).
Error: no log file open

SyntaxError: insufficient parameters to LogWrite

Response (immediate socket)

Success
Failure

Details

Writes a user message to the log. The *Source* field will be 'Client-MSG'. The *Text* field will be *message*. (See LogOpen for details of the log format.)

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.16.1.6 LogClose

Message

LogClose

Originator

Client

Response

Info: log closed
Error: no log file open
SyntaxError: invalid parameters to LogClose

Response (immediate socket)

Success
Failure

Details

This terminates logging and closes the log. (The LogSetLogging and LogSetOptions states are preserved, should you open another log.)

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.16.2 Permanence of data recording

Industrial research often requires that data be stored onto a read-only medium at the time of collection. Whisker supports writing to any file supported by Windows, but you should note that devices such as CD-ROMs and CD-RW devices may not operate fast enough to cope with high-speed data output from Whisker. Additionally, some such devices may not allow software to write small packets at arbitrary times (e.g. they may want the whole file to be written to CD at once).

If your output device cannot cope with 'live' data output from Whisker, save it to a high-speed disk

using operating-system (Windows) access protection.

This advice also applies to data recording from Whisker clients.

8.8.16.3 Digital signing of data logs

Why use digitally-signed logs in a behavioural research environment?

Digital signatures help to provide a **verifiable data chain.**

Whisker is used to capture data from devices and thus from subjects. Whisker clients interpret the raw events (data) to make inferences about the subject's behaviour — and thus, potentially, to make inferences about the effects of drugs or other manipulations. The US Food and Drug Administration lays down guidelines on the recording of data during studies of new drugs, aimed in part at ensuring that data is not fraudulently manipulated. Specifically, raw data should be preserved in a non-modifiable form at all possible stages. Whisker aids this process by using strong encryption techniques to provide data logs that were provable created by Whisker, and not modified subsequently.

insert FDA guidelines here

These principles apply, of course, to data capture in any environment.

If you also want a guarantee that data has not been fraudulently *deleted*, you should record or copy the Whisker logs onto a read-only medium and secure it physically.

What does digital signing assure?

- Digital signing assures users that a validly-signed log was generated by the Whisker server program, or by someone who knows the Whisker digital signature private key.

The only two people with access to the raw form of the key are Rudolf Cardinal and Mike Aitken, the authors; we give you, the users, the assurance that we make every effort to keep this private key safe.

Unfortunately, the Whisker server software must also 'know' the private key, because it signs the logs. Thus, the private key is embedded in the Whisker server, and is in principle extractable. This is an **inherent design flaw** (see Howard & LeBlanc, 2002, p. 168 and Chapter 7, esp. p189–190). We have hidden the key within the server and we believe it is difficult to retrieve the key from the Whisker software; however, it is not impossible. There is no obvious way to make this system perfect: Whisker must be able to function on an isolated computer (without live Internet access), so all the information required to sign a log must be present on that computer. The best we can do is to make that information ephemeral, encrypted, and heavily disguised. We also guarantee that the key-hiding techniques used in Whisker are not the same as those in other programs of ours (such as Whisker-CANTAB clients), and that Whisker's digital signature key pair is used for no other purpose.

If you are serious about your data security, a trusted member of the research team must sign data records as soon as possible using a private key that is guaranteed not to be available to a malicious user, such as a private key held on a smart-card carried by the trusted user.

The public RSA key is available for everyone to know. It is:

```
30820120300D06092A864886F70D0101010500038201D0030820108028201010009E97F8B6ED97D
3EA3BBB0CBAB96E87C215D026A7904A635D2C0D143B5392CD1C78AE178953416D09F0F0275BFFBA
6CDD5B2B73BE17924B27E0BFA89AD011C693608EBCF2ADCFCD77340804E155617714A722B0A4EB3
60D23B2CBCFB1749A2E3C2C03C05B392444C05C073310D3B75499E5DF4D598CDB3B2CD9F1976A4E
677F86974419C129712631F3A51F79DA9A7CB303C018EFFA7F2108C1707E5E0F5730C6F3D5922D6
AEF6B24F424A867AFB82936C7BE71093624D5965C6E9733076BCA18F6D691E89F9FBAA3661F52AD
BC6B12E12EA74C7E41E017E943F20E4292FEF721CF33B7AE13077F61297893F2E7F6E97247EDB7E
E17A9DBF0A809C317137C7BBF020111
```

As the system has this inherent weakness, we make it harder still to falsify logs. Quasi-private information is encrypted with a quasi-private key and stored in the log. (By quasi-private, we mean that the information and the key are embedded in the Whisker server software but are hidden.) Users are unable to decrypt or read this data. We, the authors, hold the matching RSA key and are able to decrypt this verification information from logs to establish their authenticity once the customer's identity has been established beyond doubt. Falsification of a Whisker log requires that all these information-hiding techniques are broken — and even then, the fraudster would not know if he had succeeded in creating a valid log, as he would not be able to verify this final step.

- The log always includes timestamps. Thus, a validly-signed timestamped log guarantees that Whisker generated the log when the computer's clock was set to that time (or that someone who knows the private key faked it, see above). The timestamps are only as trustworthy as the clock on the computer that Whisker runs on.
- The log (if fully enabled) contains all communications between client and server. The fact that the log is an authentic record of what went on from the server's perspective does not guarantee that the data is an accurate reflection of a real-world experiment. A forger within a drug company might play the role of a test subject (by responding on a touchscreen, for example) and fraudulently create data that purports to be the result of a drug study. Whisker cannot guarantee against this.

The signing and verification process

Signing

1. Whisker opens a data log and writes timestamped data to it. During this time, other applications can read the log but not write to it.
2. When the log is finished, the file is not closed. (We don't want other applications to be able to modify the log before it is signed.)
3. Encrypted private verification information is written to the log.
4. A hash (message digest) is generated from the log with the SHA-1 algorithm.
5. The hash is signed using the RSA algorithm and the private key, to create the digital signature.
6. The signature is appended to the file (it is the last line), and the file is closed.

User verification (VerifyWhiskerLog.exe)

1. The verification application reads the file. All data *except* the last line is used to generate a hash with the SHA-1 algorithm.
2. The last line (the signature) is decrypted with the public key, using the RSA algorithm.
3. The decrypted key is compared to the hash. If they match, the signature is valid.

Author verification

On request, the authors of Whisker are able to provide further confirmation of the likely authenticity of a log, with the following procedure:

1. The customer's identity is established.
2. The digital signature of the log is verified (as above).
3. The extra verification information is checked with the 'author-private' key.

8.8.16.4 Verifying digitally-signed logs

The **VerifyWhiskerLog** tool (in its usual installed location, `C:\Program Files\WhiskerControl\VerifyWhiskerLog\VerifyWhiskerLog.exe`) is a command-line utility can be used to check that a log is a valid Whisker log, signed by WhiskerServer at the time of creation. The syntax is:

```
verifywhiskerlog filename
```

If the log is valid, VerifyWhiskerLog will produce the following output:

```
Whisker Public Log Verification Utility. By Rudolf Cardinal.
Copyright (C) 2002-3 Cambridge University Technical Services Ltd.

Verifying file: thing.txt
Digital signature found; checking its validity.
Digital signature is correct and matches the Whisker public key.
```

If not, the output will end in one of the following ways:

```
Unable to open file for input.
File is NOT a valid Whisker log.

No digital signature found.
File is NOT a valid Whisker log.

No contents found before digital signature.
File is NOT a valid Whisker log.

Digital signature found; checking its validity.
File is NOT a valid Whisker log.
```

For further verification, send the log to the authors. Verification via this process will require you to establish your identity to the authors directly.

For more details of this process, see Digital Signing of Data Logs.

8.8.17 ShutDown

Message

ShutDown

Originator

Client

Response

No reply

Response (immediate socket)

No reply

Details

Kills all line events, kills all timers, releases all lines and closes the network connection(s). There may be a short delay (< 1 s) before the network connection is closed.

This command should be issued when the client has finished. It releases ownership of the lines under the client's control, so another client can use them. If you close the network connection without this, the server will notice and clean up. However, **if the client crashes,** leaving the network connection open, there may be a considerable delay before the server notices.

The problem of crashed clients

Obviously, I could have programmed the system so the server regularly checked that its clients were OK, and killed all links to unresponsive clients. However, I don't agree with this philosophy; the client might be busy doing other things. The solution I went for is this: The server keeps track of the time of the last communication received from the client. The operator (you!) can also ask the server to 'ping' all its clients. The server sends a *Ping* message to the client(s), and a well-written client will respond with an acknowledgement. As this constitutes communication, this will reset the 'time since last communication' clock. This clock is displayed on the server's console (it's in the main client list). In addition, if the server tries to send a message to the client and the network link actually fails (which often happens with a dead client), the message 'ERROR SENDING TO CLIENT' will be recorded indelibly in that client's status information.

So, if you think your client is dead, have a look at its clock. If it hasn't responded for a while, you can ping it using the server's console. If it responds, its clock will be reset to zero. If it doesn't respond, *and you know the client is programmed to respond to Ping events,* or if the network error message has appeared, the client may well be dead. You can then delete it from the server's list and free up its resources. However, it's your job to decide whether that client is really dead, and not just too busy doing something else to respond to ping messages.

If you're living life on the edge and the client is on a different computer to the server, you must also bear in mind the possibility that the network is down (rather than the client).

Revision history

Implemented in WhiskerServer version 1.

8.8.18 Analogue command set

To play a signal to an analogue output device (digital-to-analogue converter; DAC), you must create it in a buffer. Just as for audio devices, analogue output buffers are created for a particular output device, so to play the same pattern on multiple DACs, create a buffer for each one. Unlike audio devices, you may *not* play back more than one buffer to one DAC at once.

- **AnalogueClaim**
  Claims analogue lines. (Analogue lines are also supported by the ClaimGroup command.) As a safety feature, when claiming analogue output lines, you may specify the voltage to which they are set if the client relinquishes control (or have the server leave them as they are at that point).

- **AnalogueSetAlias**
  Sets aliases for analogue lines.
- **AnalogueRelinquishAll**
  Releases control of analogue lines.
- **AnalogueReadConfig**
  Allows the channel's configuration state to be read. This allows the user to find out whether the channel is an input or an output channel, the voltage range for the channel (e.g. -1.25V to +1.25V).
- **AnalogueReadState**
  Reads an instantaneous voltage value from an analogue input channel (analogue-to-digital converter, or ADC).
- **AnalogueSetState**
  Sets a digital-to-analogue converter (DAC) to a specified voltage, and cancels any ongoing waveform playback.
- **AnalogueSampleSignal**
  Samples an input and/or output line repeatedly, capturing input values from the ADC (and logging values sent simultaneously to the DAC if desired). Data may be reported back over the TCP link and/or logged directly to disk. Use Whisker's other data-logging facilities to record other types of event simultaneously.
- **AnalogueCancelSample**
  Cancels channel sampling.
- **AnalogueRequestEvent**
  Sets up event notification on an analogue input or output line. You can request notification when the voltage exceeds or falls below a threshold, or enters a specified range of values.
- **AnalogueKillEvent**
  Kills an event by its event name.
- **AnalogueKillEventByLine**
  Kills an event for a given analogue line.
- **AnalogueKillAllEvents**
  Kills all analogue events.
- **AnalogueCreateBuffer**
  Creates a buffer by specifying all the necessary values.
- **AnalogueLoadBuffer**
  Loads a buffer from a data file.
- **AnaloguePlayBuffer**
  Sends a pattern of voltages to the DAC from a buffer held in memory; optionally, the buffer may be played a set number of times, or cyclically until stopped. **Note** that when a buffer stops playing (when it is not being played cyclically and it finishes, or when it is aborted), the DAC is held at the voltage currently being 'played'. Buffers should therefore finish on a sensible final voltage value (often, but not necessarily, 0 V), and if you abort a buffer prematurely, you should follow the aborting command with an **AnalogueSetState** command to ensure the DAC is producing a sensible voltage. Only one buffer may be played to the DAC at once.
- **AnalogueStopPlayback**
  Stops playback on a given channel.
- **AnalogueDeleteBuffer**
  Deletes a buffer entirely.
- **AnalogueDeleteAllBuffers**
  Deletes all buffers in memory.

8.8.18.1 AnalogueClaim

Message

AnalogueClaim *<linenumber>* **[-input|-output] [-Reset** *<voltage>***|-Leave] [–alias** *<alias>***]**
AnalogueClaim *<devicegroup> <devicename>* **[-input|-output] [-Reset** *<voltage>***|-Leave] [–alias** *<alias>***]**

Originator

Client

Response

ClaimAccepted: *<linenumber>*
ClaimRejected: *<linenumber> <reason>*
SyntaxError: insufficient parameters to AnalogueClaim
SyntaxError: invalid parameters to AnalogueClaim

For example,

ClaimRejected: 112 is a non-existent line
ClaimRejected: 5 is already claimed
ClaimRejected: line 7 is not an input line

If you set an alias, you will also get messages concerning the success or failure of the alias command. You may also get a message of the form

ClaimAccepted: 5 (alias not set)

if the claim succeeded but the alias was improperly formed.

If you request a reset voltage, you may also get an additional message:

SyntaxError: invalid reset voltage (must be number with V suffix)
Error: requested reset voltage is out of range

Response (immediate socket)

Success
Failure

If you request an alias, you will only get Success if the alias command also succeeds.

Details

Claims analogue line *linenumber* for input or output. **Lines are numbered from 0.**

By default, output lines are set to 0 V. If a 'reset' voltage is specified, output lines are set to that voltage. or if 'reset' is specified, output lines are turned off. If 'leave' is specified, the output line is left in its current state.

Input | output. Whether a line is an input or an output line is *not* determined by this parameter, but by the configuration of the server (which you can set, on the server, depending on how you wire up the apparatus). This parameter is merely a statement of your intent. You do not have to specify 'input' or 'output' when you claim the line. However, it is advisable. If you claim an input line and subsequently try to set its state, you will get an error message. It is better to state when you claim the line that you think it is an output line; if you are wrong, then the error message comes immediately and you know things aren't safe to proceed.

Reset | Leave. If you specify a 'Reset' voltage and the line is an output line, it will be set to this voltage

immediately, *and when the client gives up the line*. If you specify 'Leave', it will be left in its current state, whatever that is. **The default is a reset voltage of 0 V.** (Note that it is only possible to set the reset state when you claim a line, and not later; this is deliberate, as it is meant as a safety feature and not to be fiddled around with willy-nilly.)

The *voltage* parameter must be specified in volts (not mV) and followed by a 'V' (with no preceding space). It may be positive or negative. If it is not possible to set the line to the requested voltage, an error will be generated and the closest possible voltage used.

Server device names. If you use the second form of the command, the line specified by the combination of the *devicegroup* and *devicename* parameters is claimed.

Example

AnalogueClaim 3 -output -reset -0.150V -alias echemProbe

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.2 AnalogueSetAlias

Message

AnalogueSetAlias *<line_number>*|*<existing_alias>* *<new_alias>*

Originator

Client

Response

Info: analogue line *<devicenumber>* is using the alias *<alias>*
SyntaxError: insufficient parameters to AnalogueSetAlias
SyntaxError: invalid parameters to AnalogueSetAlias
Error: analogue line number is not a valid number or pre-existing alias
Error: AliasRefused: You do not own AnalogueLine *<line_number>*
Error: alias is blank
Error: can't have spaces in an alias
Error: aliases can't begin with a digit

Response (immediate socket)

Success
Failure

Details

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.3 AnalogueRelinquishAll

Message

AnalogueRelinquishAll

Originator

Client

Response

Info: All analogue lines relinquished
SyntaxError: invalid parameters to AnalogueRelinquishAll

Response (immediate socket)

Success
Failure

Details

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.4 AnalogueReadConfig

Message

AnalogueReadConfig *<linenumber>|<alias>*

Originator

Client

Response

Info: *<linenumber>* (*<alias>*) input|output *<minVoltage> <maxVoltage>*
Warning: AnalogueReadConfig called with multiline alias *<alias>*: only one config returned!
[a warning message sent on the main socket if such a command was received on the immediate socket]
Error: no such channel

Response (immediate socket)

input|output *<minVoltage> <maxVoltage>*
Failure

Details

Returns information on

- whether the channel is input (ADC) or output (DAC)
- the range of voltages that the channel can read/produce

Examples

Info: 1 (ECG_Lead_II) input –1.25V 1.25V
Info: 2 (RectalThermometer) input 0V 10V
Info: 3 (DobutaminePump) output –10V 10V
Info: 4 (RadioVolume) output 0V 10V

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.5 AnalogueReadState

Message

AnalogueReadState *<linenumber>*|*<alias>*

Originator

Client

Response

Info: State: *<linenumber>* (*<alias>*) *<voltage>*
SyntaxError: invalid parameters to AnalogueReadState
SyntaxError: insufficient parameters to AnalogueReadState
Error: no such line or alias
Warning: AnalogueReadState called with multiline alias *<alias>*: only one state returned!
[a warning message sent on the main socket if such a command was received on the immediate socket]
Error: AnalogueReadState cannot read line *<linenumber>*

Response (immediate socket)

<voltage>
error

Details

Reads an instantaneous voltage value from an analogue input channel (analogue-to-digital converter, or ADC) or an analogue output channel (DAC) that you have previously set.

Results are given in absolute voltage units. Voltage results always have a capital V appended with no space (e.g. –0.25V). (Voltage specification is preferred to raw ADC/DAC values as there is little room for confusion.)

Examples

Info: State: 5 (ECG_Lead_I) +0.3760V
Info: State: 7 (signalgenerator) –5.200V

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.6 AnalogueSetState

Message

AnalogueSetState *<linenumber>|<alias> <voltage>*

Originator

Client

Response

A successful call generates no response. Otherwise an error message will be generated:

Error: can't set line *lineNumber*
SyntaxError: insufficient parameters to AnalogueSetState
SyntaxError: invalid parameters to AnalogueSetState
SyntaxError: invalid line to AnalogueSetState

Response (immediate socket)

Success
Failure

Details

Sets the voltage on an analogue output line (DAC).

The voltage may have a 'V' appended (but in lower or upper case, but with no space between the number and the V).

If you try to set a voltage that is out of range, the server will quietly alter your voltage to the maximum or minimum permitted by the hardware configuration.

The AnalogueSetState command cancels any ongoing waveform playback.

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.7 AnalogueSetEvent

Message

AnalogueSetEvent *<linenumber>|<alias>* **above|below|range** [*<parameters>*] *<eventtext>*

Originator

Client

Response

Info: analogue event created: *<linenumber> <eventclass>*
Error: analogue event refused: *<reason for refusal>*
SyntaxError: insufficient parameters to AnalogueSetEvent
SyntaxError: invalid parameters to AnalogueSetEvent

Response (immediate socket)

Success
Failure

Details

The following event classes may be requested from analogue input or output lines:

- **Above** <*voltage*>. Produces an event when voltage exceeds specified value. No further events are produced while the voltage remains above the threshold; the next such event will occur once the voltage falls below and then rises above the threshold again.
- **Below** <*voltage*>. Produces an event when voltage falls below specified value.
- **Range** <*minVoltage*> <*maxVoltage*>. Produces an event when voltage enters the specified range.

Only one event of each class may be set.

Examples

AnalogueSetEvent echemProbe above 0.5V BurningBrain
AnalogueSetEvent echemProbe below –0.5V BurningBrain
AnalogueSetEvent echemProbe range –0.005V 0.005V NiceAndChilled

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.8 AnalogueClearEvent

Message

AnalogueClearEvent <*eventname*>

Originator

Client

Response

Info: killed all analogue events called <*eventname*>
SyntaxError: insufficient parameters to AnalogueClearEvent
SyntaxError: invalid parameters to AnalogueClearEvent

Response (immediate socket)

Success
Failure

Details

Clears an analogue event, or events, by the message associated with them. May clear multiple events.

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.9 AnalogueClearEventByLine

Message

AnalogueClearEventByLine *<linenumber>|<alias>* **above|below|range|all**

Originator

Client

Response

SyntaxError: insufficient parameters to AnalogueClearEventByLine
SyntaxError: invalid parameters to AnalogueClearEventByLine
Error: invalid line *<linenumber>*
Error: you do not own line *<linenumber>*
Info: cleared all events on line *<linenumber>*

Response (immediate socket)

Success
Failure

Details

Clearsanalogue events on the specified line. The default is to clear **all** events, but you can selectively clear events of a particular class (**above|below|range;** see AnalogueSetEvent).

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.10 AnalogueClearAllEvents

Message

AnalogueClearAllEvents

Originator

Client

Response

Info: All line events cleared
SyntaxError: invalid parameters to AnalogueClearAllEvents

Response (immediate socket)

Success
Failure

Details

Clears all analogue events.

Revision history

Implemented in WhiskerServer version 2.6.8.

8.8.18.11 SetOutputDirectory

Development status

NOT IMPLEMENTED YET.

Message

SetOutputDirectory *<directory>*

Originator

Client

Response

Info: output directory set to *<directory>*
SyntaxError: insufficient parameters to SetOutputDirectory
SyntaxError: invalid parameters to SetOutputDirectory

Response (immediate socket)

Success
Failure

Details

Sets the directory that the server will try to use to store output files (e.g. containing captured analogue data).

If the server is asked to create or open an output file called file.txt, it searches for it in the following order:

1. In the media directory specified by the client using SetMediaDirectory;
2. In the server's default output directory (configured on the server console);
3. As a raw filename. (If a full path is given, e.g. c:\file.bmp, then the absolute path is used; if a simple filename, e.g. file.bmp, is given, then the server searches in the current working directory for its process, typically the directory that the server's .EXE file was started from.)

Note that the path is relative to the *server*, even if you are running a client on a different computer. (It is possible to store files on a different computer. For example, if a machine called client wants a server called server to store data on the client's hard disk, it could send a filename as \ \client\client_disk\directory\file.txt. However, this is not recommended practice as it depends heavily on network performance and reliability; in particular, when high-speed analogue sampling is being used, this is unwise practice.)

8.8.18.12 AnalogueOpenOutputFile

Development status

NOT IMPLEMENTED YET.

Message

AnalogueOpenOutputFile *<filehandle> <filename>* **[-append]**

Originator

Client

Response

Info: file opened
SyntaxError: insufficient parameters to AnalogueOpenOutputFile
SyntaxError: invalid parameters to AnalogueOpenOutputFile
Error: file exists, -append not specified
Error: cannot open file

Response (immediate socket)

Success
Failure

Details

Opens the file entitled *filename*, and gives it a handle *filehandle*. The handle is used by other Whisker commands (e.g. AnalogueSample) to refer to the file in future.

If the file exists and the *–append* switch is used, Whisker will try to open the file and append new data to the end. If the file exists and*–append* is not specified, Whisker will generate an error. To safeguard your data, Whisker will not let you overwrite an existing file.

8.8.18.13 AnalogueCloseOutputFile

Development status

NOT IMPLEMENTED YET.

Message

AnalogueCloseOutputFile *<filehandle>*

Originator

Client

Response

Info: file closed
SyntaxError: insufficient parameters to AnalogueCloseOutputFile
SyntaxError: invalid parameters to AnalogueCloseOutputFile
Error: no such file handle
Error: file in use, stop sampling first

Response (immediate socket)

Success
Failure

Details

Closes the file with the handle *filehandle*. You cannot close a file if you are currently storing sampled data to it.

Comments

Files are also closed when clients disconnect.

8.8.18.14 AnalogueSampleSignal

Development status

NOT IMPLEMENTED YET.

Message

AnalogueSampleSignal <*Channel*> <*ChannelLabel*> **[–TimeToSample** <*TimeToSample*>**] [–Rate** <*rate*>**] [–OutputTCP] [-OutputFile** <*filehandle*>**] [–MaxTimeToHoard** <*MaxTimeToHoard*>**|–MaxSamplesToHoard** <*MaxSamplesToHoard*>**]**

Originator

Client

Response

Info: Sampling channel <*channel*> as <*ChannelLabel*>
SyntaxError: insufficient parameters to AnalogueSampleSignal
SyntaxError: invalid parameters to AnalogueSampleSignal
Error: no such file handle open

Response (immediate socket)

Success
Failure

Details

Channel = the name or number of the channel to sample
ChannelLabel = the label used to tag data returning from this channel

–TimeToSample = the time (in ms) for which to sample the channel (e.g. 60000 for a 1-minute sample, 3600000 for a 1-hour sample). A sample time of '0' will cause Whisker to sample until an *AnalogueCancelSample* command is received; this is the default.

–Rate = the frequency, in Hz, with which to sample the channel. The default is 1 Hz. An analogue input card such as the Amplicon PCI230 can sample at up to 312 kHz (–rate 312000). A typical slow-technique electrochemistry experiment might sample once every 20 s (–rate 0.05).

–OutputTCP causes data to be returned down the TCP socket directly to the client. The *main* socket is used, and never the immediate socket, as data can turn up unpredictably (just like digital events). Data returned over the TCP socket is prefixed with AnalogueData: (see AnalogueData for the data format). The *–OutputTCP* command can be used in conjunction with *–OutputFile.*

–OutputFile causes data to be logged to the file with the specified handle (see AnalogueOpenOutputFile and AnalogueCloseOutputFile). The file format is described under AnalogueData. This command can be used in conjunction with *–OutputTCP.*

–MaxTimeToHoard and ***–MaxSamplesToHoard*** determine the number of samples (or the time in milliseconds) that the server will 'hoard' before sending them to the client. You may specify this either as a time or as a number of samples (but not both). If you are sampling at 250 Hz, for example, you might want the server to tell you the sampled value as soon as each sample comes in (*–MaxTimeToHoard 4* or *–MaxSamplesToHoard 1*), or you might want to be kept up to date only every second (*–MaxTimeToHoard 1000* or *–MaxSamplesToHoard 250*). **Note 1:** the server reserves the right to send data out *more* frequently than you specify (it may do this if its buffers are becoming full and it wants to offload some data). **Note 2:** if you specify a *MaxTimeToHoard* that exceeds the sampling period (e.g. if you sample at 0.05 Hz or every 20 s and ask the server to hoard samples for a maximum of 1 s before sending them to you) the server will not send you data until it has some, but it will send data when that time comes.

Samples are always time-stamped with the system 24h clock and the internal system clock, to millisecond accuracy.

Examples

AnalogueSampleSignal 2 Bed2_ECG_LeadII –Rate 250 –OutputTCP
AnalogueSampleSignal 3 Bed2_PCWP –Rate 100 –OutputTCP

Comments

One AnalogueSampleSignal command sets up sampling on one channel only. Similarly, one AnalogueData: message contains data for one channel only. If multiple channels are being sampled simultaneously, multiple messages will be sent.

Output lines can be sampled. Why? So you can correlate input voltages from the ADCs precisely with output values that are being played back from a buffer to a DAC — see below — and have both input and output voltages accurately time-stamped. **Note:** At high data rates, Whisker delivers a sequence of output values to the DAC and relies on the DAC to time them correctly. Similarly, at high data rates, Whisker requests a whole series of data values from the ADC (and relies on the ADC to time them correctly). Therefore, as a **tip:** if the DAC values must be *precisely* correlated to ADC input values, you should wire a copy of the DAC output to a spare ADC directly.

If you issue a second AnalogueSampleSignal command on a channel you were already sampling, the new settings will override the old, effective as soon as possible.

See also

- AnalogueData
- File format for analogue data written directly to disk

8.8.18.15 AnalogueData

Development status

NOT IMPLEMENTED YET.

Message

AnalogueData: *<ChannelLabel> <time>,<voltage> <time>,<voltage> <time>,<voltage>…*

Originator

Server

Details

This command returns data requested with "AnalogueSampleSignal –OutputTCP". See AnalogueSampleSignal.

ChannelLabel = the label you specified when issuing the AnalogueSampleSignal command. It identifies the channel being sampled.

time = the system time in ms (floating-point to allow for microsecond accuracy), according to Whisker's internal clock (see *TimeStamps* and *ResetClock*).

voltage = the sampled value, expressed in the usual way as a voltage (i.e. with a 'V' appended).

Examples

AnalogueData: Bed2_ECG_LeadII 400.0,+0.045V 404.0,+0.045V 408.0,+0.048V 412.0,+0.057V
AnalogueData: Bed2_ICP 400.0,-0.23V

See also

- AnalogueSampleSignal

8.8.18.16 Format for analogue data logged directly to disk

Development status

NOT IMPLEMENTED YET.

Format for data logged directly to disk (from AnalogueSampleSignal –OutputFile)

Data logged directly to disk is formatted so that it can be interpreted easily by a wide range of software, and can be imported directly into a relational database. The data is in comma-delimited text format, with the first line containing the field names. The following example illustrates a file that has been used for multiple simultaneous AnalogueSampleSignal commands (three independent channels):

```
SystemDate_YMD,SystemTime_HMS,Time_ms,ChannelLabel,Value_V
2001-11-19,15:24.56,400.0,Bed2_ECG_LeadII,0.045
2001-11-19,15:24.56,400.0,Bed2_ICP,-0.23
2001-11-19,15:24.56,400.0,Bed2_PCWP,0.18
2001-11-19,15:24.56,404.0,Bed2_ECG_LeadII,0.045
2001-11-19,15:24.56,408.0,Bed2_ECG_LeadII,0.048
2001-11-19,15:24.56,410.0,Bed2_PCWP,0.49
2001-11-19,15:24.56,412.0,Bed2_ECG_LeadII,0.057
```

(For the medically oriented, note that these values aren't representative of real ECG leads and presuppose some amplification has occurred before sampling! Furthermore, Whisker only reports absolute ADC voltage values back; calibration and conversion to real-world pre-amplified voltages/pressures/concentrations/etc. is the client's job.)

Note that the Time_ms field is not the time *within* the second specified by SystemTime. The SystemDate/SystemTime fields should be considered somewhat independent of the Time_ms field; the system date/time are imprecise while the Time_ms field is a high-precision field. However, having both can be useful.

Note that this field layout generates about 50 bytes of output per sample, depending on the length of the channel label. (Even a more economical single-channel file with just the millisecond timer and the value would occupy ~10 bytes per sample, and a human-unreadable time/value pair wouldn't be much smaller at 6–8 bytes per sample depending on precision.) Similar considerations apply to the TCP output. **It is therefore UNWISE to attempt continuous sampling with logging to disk for long periods at very high**

sample rates, as the disk or network subsystem may be saturated. For example, continuous sampling at 312 kHz would generate 2.5–15 Mbytes·s^{-1} with any of these data formats.

See also

- AnalogueSampleSignal

8.8.18.17 AnalogueCancelSample

Development status

NOT IMPLEMENTED YET.

Message

AnalogueCancelSample -channel <*Channel*>**|-ChannelLabel** <*ChannelLabel*>

Originator

Client

Response

Info: Sampling cancelled
Info: Sampling cancelled (not sampling anyway!)
SyntaxError: insufficient parameters to AnalogueCancelSample
SyntaxError: invalid parameters to AnalogueCancelSample

Response (immediate socket)

Success
Failure

Details

Cancels sampling of an analogue channel. You may specify the channel by its number/alias, or by the label you associated its data with in the AnalogueSampleSignal command.

Examples

AnalogueCancelSample -channel 2
AnalogueCancelSample -ChannelLabel Bed2_PCWP

8.8.18.18 AnalogueCreateBuffer

Development status

NOT IMPLEMENTED YET.

Message

AnalogueCreateBuffer <*linenumber*>|<*alias*> <*buffername*> <*buffersize*> <*datum*>,<*datum*>,<*datum*>,…

Originator

Client

Response

SyntaxError: insufficient parameters to AnalogueCreateBuffer
SyntaxError: invalid parameters to AnalogueCreateBuffer
Info: created analogue output buffer <*buffername*>
Error: no such analogue output channel

Response (immediate socket)

Success
Failure

Details

Creates a buffer of length *buffersize* for use with channel *linenumber* filled with the data points listed.

Example

AnalogueCreateBuffer flipflop 2 -0.5V 0.5V

Comments

This is a highly tedious way to create a buffer.

8.8.18.19 AnalogueLoadBuffer

Development status

NOT IMPLEMENTED YET.

Message

AnalogueLoadBuffer <*linenumber*>|<*alias*> <*buffername*> <*filename*>

Originator

Client

Response

SyntaxError: insufficient parameters to AnalogueLoadBuffer
SyntaxError: invalid parameters to AnalogueLoadBuffer
Error: can't find data file
Error: invalid analogue data file
Error: no such analogue output channel
Info: loaded data as buffer <*buffername*>

Response (immediate socket)

Success
Failure

Details

Creates a buffer for use with channel *linenumber* by loading a file from disk. The file should contain data

with one value per line; lines beginning with '#' are treated as comments and ignored. For example:

```
# Voltage waveform for fast cyclic voltammetry sweep.
# Intended to be swept at 300 V/s, so 30 kHz (10 mV step).
0.0V
0.010V
0.020V
0.030V
0.040V
...
```

8.8.18.20 AnaloguePlayBuffer

Development status

NOT IMPLEMENTED YET.

Message

AnaloguePlayBuffer <*linenumber*>|<*alias*> <*buffername*> **[-rate** <*rate*>**] [-reload** <*reloadcount*>**]**

Originator

Client

Response

Info: Playing buffer <*buffername*> on channel <*linenumber*> at <*rate*> Hz (reload count <*reloadcount*>)
SyntaxError: insufficient parameters to AnaloguePlayBuffer
SyntaxError: invalid parameters to AnaloguePlayBuffer
Error: no such analogue output channel
Error: no such buffer

Response (immediate socket)

Success
Failure

Details

–Rate = the frequency, in Hz, with which to sample the channel. The default is 1 Hz. An analogue input card such as the Amplicon PCI230 can sample at up to 312 kHz (–rate 312000). A typical slow-technique electrochemistry experiment might sample once every 20 s (–rate 0.05).

–Reload determines whether the buffer should be played several times. A parameter of 0, the default, causes the buffer to be played once. A parameter of 1 causes the buffer to be replayed once (for a total of two playings), and so on; a parameter of –1 causes the buffer to be played endlessly (until an AnalogueStopPlayback command is received).

If the channel was already playing a buffer, playback of the previous buffer will be stopped and the new buffer will be played.

8.8.18.21 AnalogueStopPlayback

Development status

NOT IMPLEMENTED YET.

Message

AnalogueStopPlayback *<linenumber>|<alias>*

Originator

Client

Response

Info: playback stopped on channel *<linenumber>*
SyntaxError: insufficient parameters to AnalogueStopPlayback
SyntaxError: invalid parameters to AnalogueStopPlayback
Error: no such analogue output channel

Response (immediate socket)

Success
Failure

Details

Stops any buffers being played on the specified channel. (The channel's voltage will remain at the last output value; if you want to specify a new value, issue an AnalogueSetState command).

8.8.18.22 AnalogueDeleteBuffer

Development status

NOT IMPLEMENTED YET.

Message

AnalogueDeleteBuffer *<linenumber>|<alias> <buffername>*

Originator

Client

Response

Info: buffer deleted
SyntaxError: insufficient parameters to AnalogueDeleteBuffer
SyntaxError: invalid parameters to AnalogueDeleteBuffer
Error: no such buffer
Error: no such channel

Response (immediate socket)

Success
Failure

Details

Deletes the specified buffer (stopping its playback if it was being played).

8.8.18.23 AnalogueDeleteAllBuffers

Development status

NOT IMPLEMENTED YET.

Message

AnalogueDeleteAllBuffers *<linenumber>|<alias>*

Originator

Client

Response

Info: buffers deleted
SyntaxError: insufficient parameters to AnalogueDeleteAllBuffers
SyntaxError: invalid parameters to AnalogueDeleteAllBuffers
Error: no such channel

Response (immediate socket)

Success
Failure

Details

Deletes all buffers for the specified channel.

8.9 Writing Whisker clients: general principles

I'm first going to discuss the choice of programming language, discuss the principles of behavioural task programming, and then walk you through some examples – a Visual Basic client, a brief digression into Perl, and then some C++ clients together with the library designed to simplify client programming in C++.

8.9.1 Programming tasks: design principles

This seems like a good point to summarize the three main ways of designing a multi-chamber control system.

1. General-purpose language with hardware-specific extensions

Example: Arachnid.

Pros:

- Programming experience (in the case of Arachnid, with BBC BASIC) applies to

completely different applications

Cons:

- Independent control of multiple boxes is hard to achieve (with danger of programmer error)
- Very hard to run two different tasks simultaneously, as only a single user program runs at once
- Need to learn specific language

2. Custom language designed for operant chamber control; software runs user tasks 'inside' the server

Example: Med-PC®.

Pros:

- Safe, independent control of multiple operant chambers
- Simple programming language (or even graphical task design)

Cons:

- Programming language has restricted capabilities
- Programming experience not useful for completely different applications

3. Server and clients are independent programs

Example: Whisker.

Pros:

- Safe, independent control of multiple operant chambers
- User free to choose programming language for client task (unless a simplifying "super-client" is used)
- Programming experience generalizes to other applications (unless a simplifying "super-client" is used)

Cons:

- Need to learn at least one programming language (unless a simplifying "super-client" is used)
- Need to communicate with server (unless a simplifying "super-client" is used)

Comparison

Options 1 and 3 may entail a **choice of programming language** (though some examples, such as Arachnid, limit you to just one language). In this situation, there are three considerations: how easy the language is to learn, the merits of the language itself, (including whether it is *interpreted* or *compiled*), and how useful the language is applications outside operant chamber control.

One of the main dangers of using an interpreted language like BBC BASIC for operant chamber control is that typing and syntax errors may not be discovered until the task crashes with a rat in the box. **Compiled languages** find all those kinds of bugs before you ever run the program. I strongly recommend the use of compiled languages for all but the most trivial programming tasks when it is very important that the program does not crash.

(My [RNC] favourite is **C++,** the language the Whisker server was written in. C++ is one of the

most popular languages world-wide, and for good reason. If you choose to learn and use C++, a fringe benefit is that you will become skilled in a general-purpose programming language, and computers everywhere will become your slave.)

The principle 'con' of Whisker's approach is that the user must develop some minimal programming ability in the chosen language. I have attempted to reduce the programming burden by providing *libraries* of code for different programming languages, so writing a task becomes a more a matter of filling in blanks.

8.9.2 Choosing your programming language

The only requirement of a client language is that it can communicate using TCP/IP sockets. This does rule out some languages.

Beginners are probably best off with Microsoft **Visual Basic.**

Intermediate programmers might like to consider **Visual Basic or C++.**

C++ is harder to learn but gives you full control over the system; it's also enormously powerful. The programming skills you develop will be useful in many different situations. C++ is the closest to the world standard for 'real' programming; knowing C++ and Windows programming will make you sought after. I quite like Microsoft's C++ development environment, but it is something of a mixed blessing; you can do a lot very quickly, but it does hide some of the details from you (which, depending on your perspective, is a convenient or a confusing thing). It's also cheap, with an academic licensing system.

One other possibility is **Perl,** a language originally designed for scripting and text processing. It looks pretty obscure at first glance, but half an hour with a book like *Learning Perl* (Schwartz & Christiansen) should give you all you need to write effective programs. The Perl compiler/ interpreter for Windows is free.

Expert programmers don't need to be told. For your information, the server and test client were written using Microsoft Visual C++ 6.0 using Winsock. The server runs off a single timer that polls at 1 ms (or as close to that as the system can manage); when the server receives this message, it polls its digital I/O boards and checks if any timer events need servicing. All other events are triggered through the Windows or sockets communications system.

8.9.3 Programming models for behavioural tasks

The technique you adopt can make task programming easy or difficult. Think through your task carefully before beginning.

Types of task and appropriate programming models

Most behavioural tasks that I've come across use one or more of the following styles of programming.

Free-operant tasks

Simple reinforcement schedules fall into this category. Take the example of a ratio schedule: whenever a lever is pressed, a counter is advanced, and whenever the counter reaches a certain number (the ratio requirement), reinforcement is delivered and the counter is reset.

In Whisker, where everything is driven by events, this is phenomenally easy to code. First, you ask the server to send you a message when a lever is pressed – perhaps *BattleshipPotemkin*. Part of your program will receive messages coming from the server. If a message begins with *Event:*, the rest of the message should be checked. If you had received *Event: BattleshipPotemkin*, then the rat has pressed a lever. Increment your counter variable and if appropriate, send commands to deliver reinforcement.

State models

An extension of this model is a *state*-based system. To be frank, every behavioural task can be programmed using this model, but it's worth considering explicitly. The program maintains a representation of what *state* the task is in. When it receives information about the subject's actions, its response is based on (1) what the subject did, but also (2) what state the task is in.

As a very simple example, and an extension of the ratio schedule, suppose you want to ignore lever-presses while reinforcement is being delivered. You would implicitly have defined two possible States of the Box: *hanging around waiting* and *delivering reinforcement*. When the lever-press message comes in and the box is in the *delivering reinforcement* state, nothing is done about it; if the box is in the *hanging around* state instead, the schedule counter is incremented. In this example, you would also set up timers to inform you when enough reinforcer had been delivered, and this would trigger the state to change from *delivering reinforcement* to *hanging around*. (The transition in the opposite direction obviously occurs when the rat fulfils the ratio.)

I have found that many novice programmers use state-based programming techniques without really realizing it. I think it is well worth representing the state of the box *explicitly* in your code. (There are often more states than you think!)

State models in complex tasks

Sometimes, of course, your task may be too complicated to represent with a single state chain. Perhaps you are training animals to respond under a discriminative stimulus. Different stimuli are presented and lever-pressing is measured during each stimulus. The stimuli are controlled on one progression (e.g. interstimulus interval → S1 → ISI → S2…), and you want to ignore lever-presses while your pump is running, so you implement *waiting* and *delivering reinforcement* states as well. It would be easiest to represent these states in two variables (maybe called StimulusSequenceState and LeverState). That way the transition from one state to the other is clearly defined for each variable (if a single variable was used for both, you'd have to work out which stimulus state to go back into when the reinforcer has been delivered, and so on). The response to a lever-press would then depend on the conjoint state of both variables – reinforcement would be delivered when the discriminative S+ was on and the pump was off.

Furthermore, states may be controlled on the basis of time (and therefore, in Whisker, timers). A random interval schedule sets up a 'ticker'; every second, reinforcement is or is not 'set up'. When the rat presses a lever, reinforcement is only delivered if it has been set up by the timer-based schedule. This is another example easily implemented if you think about the states involved.

Most complicated tasks are very easily represented using state models. For example, the notorious five-choice serial reaction time task has various states: when a stimulus is presented, for example (the *Stimulus On* state?), several things can happen: the rat can respond correctly or incorrectly, or it can fail to respond within a time limit. There are three states that the system can move into: a timer will move it into the *Omission Timeout* state, but before this occurs, a response from the rat

can move the system into a *Delivering Reward* or a *Incorrect Response Timeout* state. (Maybe in this example the two timeouts do not need to be distinguished.)

8.9.4 Programming benefits of Whisker's design

Keeping it simple

The main operant control system used by our lab at the time Whisker was first written was Arachnid (Paul Fray Ltd / Cenes, Cambridge). This is an extension of BBC BASIC. In designing Whisker, I [RNC] was particularly keen to avoid the sources of programming error that were common in Arachnid programs. Among the most common problems are typing errors, misunderstandings (often due to illegible code), inappropriate use of global variables, failure to design tasks carefully, and cross-talk between boxes.

Some of these problems are due to the choice of language. Replacing an interpreted language like BBC BASIC with a compiled, strongly type-checked language like C++ vastly reduces the number of typing errors that find their way into programs – the compiler picks them up, so you can guarantee that a program that compiles successfully is at least syntactically correct. Logical errors can still occur, but C++ encourages you to write small, easily testable chunks of code that hide information from each other, so many logical problems (including over-reliance on global variables) disappear.

On the other hand, some programming problems we experienced with Arachnid were due to weaknesses in the design of the system. When a program must control six operant boxes, the potential for errors increases; it is the programmer's responsibility to make sure that every switch and timer event is tagged to the correct box, to create a system whereby different boxes can run asynchronously, and so on.

Whisker follows the Unix philosophy instead: rather than one huge program, create lots of small ones. If one program only controls one box, errors of cross-talk are simply not possible.

Most of the programming examples I have supplied only control a single box. The exception is the SecondOrder task, which can control other boxes for the purpose of yoking. In this case, all the potential for cross-talk exists. The program calls its events things like *Yoke_4_Active_Lever*, and passes the *Active_Lever* part of this message on to a data structure representing the fourth yoked box. (SecondOrder follows the C++ principle of representing each operant chamber as an independent object that stores its own data.) Theoretically, it would have been possible to implement yoking by running two kinds of clients simultaneously, one controlling all the boxes using Whisker's line-grouping facility (discussed earlier, in Part V) and the other reading inputs from the yoked boxes, but then there's a problem of synchronizing the two programs; it would really have been impractical.

Naming your events

You could name your events after Russian films, but that would be silly. More useful would be to define events like NOSEPOKE_ON, which is what I tend to do. But there's no reason why you can't change the event name during the task to make it simpler to process events – for example, you could set things up so that the nosepoke detector generates INITIATE_TRIAL sometimes and REWARD_COLLECTED at others.

Events: a comparison to Arachnid

For those of you that are familiar with Arachnid, a direct comparison may be helpful. In Arachnid, you execute `PROCpipe_switch()` and `PROCpipe_timer()` calls, passing the name of a BASIC function of your own. When the event occurs, Arachnid calls your function.

Whisker gives you control at one level higher up. You can receive all the messages coming from the server, and decide what to do with them – which might involve calling other functions, just as Arachnid would have handled it, but it doesn't *have* to. It's a more powerful system and it makes for simpler code in some situations.

A word about ClearEvent commands: these commands stop any more messages being sent from the Server – but it doesn't send back any that are 'on the way'. This is slightly different to Arachnid, when a event could never be detected after it had been 'Killed'. This is especially important when we have to deal with touchscreens, which might send a lot of messages quite quickly, or when we're doing slow processing in the client. If you have to only respond to whichever of two messages comes in first, make sure the code can never respond to both: don't just assume that 'it'll never happen'.

Technical note

Of course, it is nice at times to be able to set up a complex sequence of events and have it happen without further interference; the Whisker C++ client library provides this kind of functionality. One can, for example, ask for a message to be sent to the server after a specified delay, and the library can do this for you. (Internally, it asks the server for a timer of its own, whose name begins with _sys, and it remembers what you wanted done when that timer elapses. The point is that this detail is hidden from you.) Visual Basic provides similar functionality by supporting re-entrant functions via the DoEvents command.

8.10 C++ and WhiskerClientLib

To simplify the process of writing tasks in C++, I have written a library of routines that simplify the process of communicating with the server and provide some useful behavioural functions. This library forms the basis of the **Software Development Kit**, which extends its function into several other languages.

Full source code is supplied in the *WhiskerClientLib* directory (by default, *\Program Files\WhiskerControl\Whisker\WhiskerClientLib*). The compiled libraries are to be found in the *Debug* and *Release* subdirectories. (Which of these two versions you will use will depend on your compilation settings, discussed below.)

8.10.1 About C++ and WhiskerClientLib

Learning C++

If you would like to learn C++, I [RNC] suggest you get a good compiler. The compiler is the program that converts the source code (in textual format) into an executable program, finding your typing mistakes along the way. There are some pretty good free C++ compilers; one is DJGPP, available at *http://www.delorie.com/djgpp/*. Mind you, if you're in an academic environment you can get a very cheap version of Microsoft Visual C++ 6.0 Professional, which is what I use.

In addition, you'll want a book or an electronic tutorial. Bear in mind that most books on learning

Visual C++ (Microsoft's product) are concerned more with Windows programming than on the language itself, but you should start with a plain C++ tutorial. I'm not sure what the best one for beginners currently is. The definitive reference is Stroustrup (1997), *The C++ Programming Language (3rd Edition)*, though the book is oriented towards programmers with some experience. Stroustrup created C++ (see also *http://www.research.att.com/~bs/*). There are additional literature references at the end of this Guide.

Building programs in C++

The actual process depends on the compiler. If you have loaded a workspace into Visual C++, you can press F7 to compile the application, and Ctrl-F5 to run it.

An introduction to WhiskerClientLib

Several of the C++ clients supplied with Whisker (including SimpleCPPClient, WhiskerStatus and SecondOrder) use the WhiskerClientLib **library.** A library is a collection of code to perform certain functions; its purpose is to save you reinventing the wheel. In the case of WhiskerClientLib, the library contains a set of routines that automate connecting to and communicating with the server and also assist in the creation of behavioural tasks. More detail is given later.

If you want to change and rebuild SimpleCPPClient, see also the notice regarding Debug and Release modes.

Comments on the suitability of C++

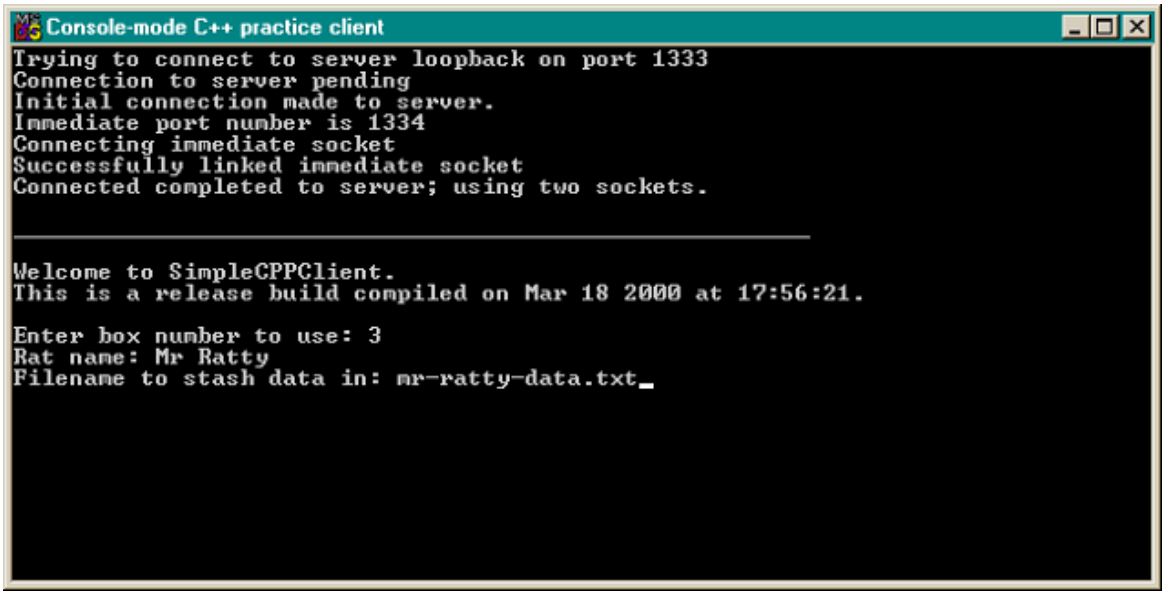
C++ is a general-purpose programming language; as such, you can do pretty much anything in it. If it's not obvious from the rest of this guide, it's my [RNC] favourite programming language.

8.10.2 SimpleCPPClient: a C++ console-mode example

About SimpleCPPClient

This is a task so simple as to be useless behaviourally – it extends a lever, records up to ten lever-presses, times out after a minute and stores the response times. Nevertheless, it illustrates communication with the server, controlling outputs, responding to timers and input devices, and data output in a comma-delimited format suitable for importing directly into a database. Furthermore, it is one of the easiest kinds of client to write in C++, because it doesn't really need any Windows graphical user interface programming. You can run the task from the Start menu.

When running, it looks like this:



The source code is provided in the SimpleCPPClient directory. If you want to open projects ('workspaces') in Visual C++, double-click on the *.dsw files (in this case, SimpleCPPClient.dsw). The process of writing such an application is documented in the following files:

- Building this program – quickly!.txt
- Building this program – from scratch, in detail.txt

It would be easy to copy this application and modify it to perform a more useful task. As this is meant to be a programming exercise, I'll leave the rest to you.

8.10.3 Classes to help you write clients

A key concept in C++ is code *re-usability*. Since I've written all the code needed to create a client task, you shouldn't have to. All you need to do is to use the **classes** that I created. A class represents an **object** in a very general sense; in this case, a behavioural task. The main class designed to help you is called **CWhiskerTask**, and here are the functions that it offers you:

This list is incomplete. Examine WHISKERTASK.H for the full catalogue.

```
class CWhiskerTask
{
       typedef void (CWhiskerTask::*WhiskerFuncPtr)(const CString&,  long); //
the type of a callback function
public:
     CWhiskerTask();
     virtual ~CWhiskerTask();

     // ******************************************
     // Event Callbacks
     // ******************************************
     // The client is expected to override some or all of these

     virtual void ServerConnected(); // main connection has just been made
     virtual void ServerFullyConnected(); // main and immediate sockets are both
connected
```

```
     virtual void ServerDisconnected(); // Called when communication is lost

     virtual void IncomingInfo(CString msg, long lTime); // an Info message has
arrived
     virtual void IncomingError(CString strError, long lTime); // an Error
message has arrived
     virtual void IncomingEvent(CString strEvent, long lTime); // an Event
message has arrived
     virtual void IncomingClientMessage(int iSourceClient, CString strMessage,
long lTime); // a message from another client has arrived
     virtual void IncomingOther(CString strMessage, long lTime); // some other
kind of message has arrived
     virtual void IncomingWarning(CString strMessage, long lTime);
     virtual void IncomingSyntaxError(CString strMessage, long lTime);
     virtual void IncomingKeyEvent(int iKey, bool bDown, CString strDocument,
long lTime);

     // Callbacks Without timestamps.
     // Override these if you don't want to handle both
     // (these will only be called if the stamped versions are not overriden)

     virtual void IncomingInfo(CString msg); // an Info message has arrived
     virtual void IncomingError(CString strError); // an Error message has
arrived
     virtual void IncomingEvent(CString strEvent); // an Event message has
arrived
     virtual void IncomingClientMessage(int iSourceClient, CString
strMessage); // a message from another client has arrived
     virtual void IncomingOther(CString strMessage); // some other kind of
message has arrived
     virtual void IncomingWarning(CString strMessage);
     virtual void IncomingSyntaxError(CString strMessage);
     virtual void IncomingKeyEvent(int iKey, bool bDown, CString strDocument);

     virtual bool PollPipedEvents(const CString& msg, long lTime); // override to
'pipe' your own events: return true to prevent them from getting to IncomingEvent
()
     virtual void StatusMessage(const CString& msg); // Override to handle status
messages from this library


     // ******************************************
     // Whisker Command Wrapper methods
     // ******************************************

     // Connecting & Disconnecting
     bool Initialise(const CString& strServerName, int iPort, const CString&
strBox = "");
     void DisconnectFromServer(); // closes all sockets
     void ShutDown(); // releases timers/events/lines and closes
     bool IsConnected();
     bool IsImmediateConnected();

     // Controlling Devices
     bool ClaimGroup(const CString& strDeviceGroup, const CString& strPrefix =
"", const CString& strSuffix = "");
```

```
     // Timer devices
     bool TimerSetEvent(const CString& strEvent, unsigned long timeInMs, int
iRepetitions = 0); // request a timer event from the server
     void TimerClearEvent(const CString& strTimer);
     void TimerClearAllEvents();

     // Audio devices
     bool AudioClaim(int iDevice, const CString& strAlias = "");
     …

     // Line Devices
     bool LineSetState(const CString& strLine, bool bState);
     bool LineReadState(const CString& strLine, bool& bState);
     ...

     // Display Devices
     bool DisplayClaim(int iDevice, const CString& strAlias = "");
     ...

     // Other multimedia
     bool SetMediaDirectory(const CString& strDirectory);

     // Client-client comms
     bool PermitClientMessages(bool bPermit = true);
     bool SendToClient(int iClientNum, CString strMsg);

     // Other Whisker Commands
     void ReportName(const CString& strName);
     void ReportStatus(const CString& strName);
     bool TimeStamps(bool bTimeStamps);
     void ResetClock();
     long RequestTime();
     CString ServerVersion();


     // ****************************************
     // Whisker Helper methods (not part of the Whisker Command set)
     // ****************************************
     CString ThisComputerName(); // returns the IP name of this host
     int ClientNumber(); // returns the client number if connected, -1 if not.

     // alternative command names...
     bool SwitchOn(const CString& strLine){return LineSetState(strLine,true);}
     bool SwitchOff(const CString& strLine){return LineSetState(strLine,false);}

     void ReplyToEvent(const CString& strEvent, const CString& strReply, bool
bTransparent = false);
     void KillReply(const CString&, const CString&);

     // Calculation functions
     static DWORD Minutes_ms(int minutes) {return minutes*60*1000;}; // converts
minutes to ms
     static DWORD Minutes_ms(float minutes) {return minutes*60*1000;}; //
converts minutes to ms
```

```
     static DWORD Seconds_ms(int seconds) {return seconds*1000;}; // converts
seconds to ms
     static DWORD Seconds_ms(float seconds) {return seconds*1000;}; // converts
seconds to ms

     // Flashing & Pulsing Lines
     void FlashLine_Pulses(const CString& strLine, int iCount, DWORD on_time,
DWORD off_time, bool bOnAtRest = false); // flashes the line iCount times
     void FlashLine_Duration(const CString& strLine, DWORD duration, DWORD
on_time, DWORD off_time, bool bOnAtRest = false); // flashes the line for
"duration" ms
     CString StartFlashLine(const CString& strLine, DWORD on_time, DWORD
off_time, bool bOnAtRest = false); // starts the line going; returns its event
label
     void StopFlashLine(const CString& strLine, const CString& strEventLabel,
bool bOnAtRest = false); // stops the flashing

     // Timing
     bool SendAfterDelay(const CString& strMsg, DWORD delay, const CString&
strEventLabel = ""); // send a message after a given delay
     DWORD CurrentTime(); // return system time (ms)
     int TestNetLatency(); // return ms delay for Server->Client->Server
messageloop

     // Useful string manipulation functions
     static bool GetPrefix(const CString& msg, CString& prefix, CString&
remainder); // IN: msg from server. OUT: message prefix and remainder
     static int GetIntegerParam(CString &strSentence); // chops first word out of
sentence and converts it to a number
     static void GetNextWord(CString &strWord, CString &strSentence); // chops
first word out of strSentence, stores in in strWord
     static CString StringWithQuotes(const CString &string){return "\"" +
string + "\"";}; //puts quotes round the string
     CString UniqueString(); // returns a unique string

     // Probability functions
     float RandomProbability(); // returns random float, 0-1
     int RandomInt(int max); // returns a random integer, 0-max
     int RandomInt(int min, int max); // returns a random integer, min-max
     bool Chance(float p); // returns true with probability p
     void SeedRandomNumberGenerator(); // seeds random number generator with
system clock

// Advanced Communications:

     // To Server:
     // The client does not need these, for any of the current Whisker Command
set, but
     // can be used if the user wants to talk directly to server.

     bool Send(const CString &msg); // sends msg to server
     void ImmediateVoid(const CString &msg); // sends msg to immediate socket,
ignores reply
     bool ImmediateBoolean(const CString& question); // sends question on
immsocket, returns success/failure
     bool ImmediateSocketQuery(const CString& out, CString& in); // sends "out"
```

```
on immsocket, puts reply in "in"
     CString ImmediateSocketQuery(const CString& question); // sends "question"
on immsocket, returns reply
     CString ImmediateSocketQuery(const CString& question, long& Timestamp); //
sends "question" on immsocket, returns reply with timing info
     bool RemoveTimestamp(CString& msg, long& Timestamp); // removes timestamp
(if any) from reply & rets value

     // From Server
     // Any override must call the baseclass version in order to use the
     // library correctly (Immediate Socket connection & callbacks & library
event scheduling)

     virtual void IncomingMessage(CString strRecvd); // handles raw incoming data
from socket
     virtual void Parse(CString& msg); // handles CR/semicolon-stripped messages;
also deals with immsocket connection and pinging

     //conversions to string
     static CString OnOffString(bool bOn);
     static CString DisplayPenOptionsToString(LOGPEN* pOptions);
     static CString DisplayBrushOptionsToString(WHISKERBRUSHOPTIONS* pOptions);
```

I'm not going to go through all of these functions in detail. You have the library source code, which is commented, and you have several examples of clients that use the library (SimpleCPPClient, WhiskerStatus). If you know C++, that should be enough to get you going. Remember, the *virtual* functions are the ones you might override; the others are functions you call actively.

Note that CWhiskerTask provides a whole host of useful functions. Try looking there before you write a routine yourself.

Technical note: callback function parameters

Several of the library functions use **callback** functions. If I was writing 'pure' code, I'd probably have made them of type CObject* (since the library is written using MFC) and use run-time type information (RTTI) to establish what parameters were actually coming into a particular callback function. This would allow a pointer to any CObject-derived class to be passed.

Instead, I made the parameter of type **const CString&** (that is, a reference to a constant [immutable] CString). This necessitates using CString::Format to pack parameters into a string, and there are some custom routines in the library to extract them.

It's not as neat; why do it? Basically, I felt it was too optimistic to expect users to manage explicit casts and RTTI checking, even though it makes for poorer code. Everyone understands strings.

Technical note: implementing your own callback functions

For technical reasons, you have to do a *little* bit of work as the client here. These instructions assume your client task (CMyTask) is derived from CWhiskerTask.

You should include this snippet in your class definition:

```
class CMyTask : public CWhiskerTask
{
    // ...
    CPipedEventList<CMyTask> m_PipedEvents;
    bool PollPipedEvents(CString msg) { return m_PipedEvents.Poll(); }
    // ...
}
```

Then, to use the code, your callback function must take a single **const CString&** parameter, and return `void`. Then you can do this (the example is to deliver a single pellet):

```
void CMyTask::PelletOn()
{
    Send("LineSetState PELLET on");
    Send("TimerSetEvent 40 0 _pellet_dispensed");
    m_PipedEvents.AddEvent("_pellet_dispensed", this, &CMyTask::PelletOff,
"hey there", PIPED_EVENT_NUMBER, 1);
}

void CMyTask::PelletOff(const CString& message)
{
    // message will contain "hey there",
    // but it could contain something useful!
    Send("LineSetState PELLET off");
}
```

This code is unnecessary as you could just have used *SendAfterDelay("LineSetState PELLET off", 40)*, but it illustrates the principles.

8.10.4 Creating C++ clients for Whisker

Creating console-mode C++ applications for Whisker

The discussion of SimpleCPPClient covers this. In particular, the text file *Building this program – from scratch, in detail.txt* gives you step-by-step instructions.

Creating Windows-based C++ applications for Whisker

In Visual C++, to create the skeleton of a Windows simple application you just follow these steps:

1. *File* → *New* → *Project* → *MFC AppWizard (exe)* → give it a name and a directory
2. Choose type (SDI, MDI, dialogue). I'm going to use dialogues for a while because they're easier.
3. Include support for Windows Sockets.
4. Accept all the other defaults.

Creating a behavioural task using the library

Include the library files in the project

Regardless of the type of application you create, you will next want to include the WhiskerClientLib library in your project. Choose *Project* → *Add To Project* → *Files*, look for

files of type *Library files (*.lib)* and choose either

WhiskerClientLib\Debug\WhiskerClientLib_Debug.lib

or

WhiskerClientLib\Release\WhiskerClientLib_Release.lib

Choose the version of the library that matches your compilation settings; see below.

Derive your task's class from CWhiskerTask

The core of the behavioural task is a class derived from CWhiskerTask.

Insert a new class (*Insert → New Class*) and name it (within the SecondOrder program, it was called CSecondOrderTask). Make it a generic class, and tell the compiler that it's derived from CWhiskerTask with public inheritance.

Include the library header file

Finally, so your code knows what's in the library, add the following line to your new class's header file:

```
#include "../WhiskerClientLib/WhiskerTask.h"
```

Depending on the directory you are developing your project in, you'll need to edit the path in that statement so that it does actually reach WhiskerTask.h!

Debug and Release mode in Visual C++, and the Whisker client library

One important thing. Visual C++ applications can be compiled in two modes, Debug and Release. Compiling a project in **Debug** mode builds information into the resulting program that can be used for debugging; if the application crashes, you can step through the source code to find out where, and so on. **Release** mode is used for finished products: the resulting file is smaller, but it doesn't contain debug information. You can also let the compiler optimize the code for speed or size when you use Release mode (though I have avoided this since I found some bugs in the VC++ 6.0 optimizer, i.e. it screws the code up sometimes).

Given a choice, I would have distributed SimpleCPPClient in Debug mode – after all, it's a programming example and not a useful behavioural task. However, I'm not allowed to distribute the libraries that you need to run Debug mode applications.

If you own Visual C++ and want to switch SimpleCPPClient back into Debug mode, open its project by double-clicking on *SimpleCPPClient.dsw* and choose *Build → Set Active Configuration*. Change the setting to *Win32 Debug*. When you next build the project (just press F7) it will be in debug mode.

I'm afraid there's one other thing to do first. You can't mix debug and release libraries, so I've had to create two versions of the WhiskerClientLib library, which this application uses. Rather than hide it away, I've tried to make it very obvious: if you switch the left-hand view to *FileView*, you should see *WhiskerClientLib_Release.lib* in the file list. You should delete this, then choose *Project → Add To Project → Files*, look at files of type *Library Files (*.lib)* and double-click on *WhiskerClientLib_Debug.lib*, which lives in the *WhiskerClientLib\Debug* directory.

This applies to every application built using WhiskerClientLib; if you want to compile in Debug mode, use the debug version of the library; for release builds, switch to the release library.

Warning. If you mix debug and release-mode libraries (i.e. by building your program in debug mode and linking in the release mode of WhiskerClientLib, or vice versa), you will get a few warnings as you build the program. The compiler will let you run the code, but it may CRASH at some unpredictable moment. Don't ignore the warnings! Use the proper library. The errors you get look like this:

```
LINK : warning LNK4098: defaultlib "mfc42.lib" conflicts with use of other libs;
                            use /NODEFAULTLIB:library
```

8.10.5 A few last suggestions

Debugging

The Server Is Your Friend. Use its debugging facilities and monitor what your program is doing.

Front ends

A text-based (console mode) front end offers little opportunity for the user to wreak mischief. However, Windows-based front ends typically offer a host of buttons to press or menu items to choose; you must ensure that the user can't do anything inappropriate (like disconnect from the server without saving the data, or trying to connect twice). The usual way of achieving this in Windows is to disable ('grey out') the relevant buttons so they can't be selected when their use is unsuitable.

Controlling multiple boxes

Think twice before doing this. One of the strengths of Whisker is that you often don't need to; you can run a second copy of a simple one-box program. If you do need to control several boxes from the same program, remember the following:

- For yoking, you can always give two output lines the same name and they'll be controlled as one.
- You must name your inputs so you can distinguish them (e.g. *Box_1_LeftLever*, *Box_2_LeftLever*).
- Be extremely careful not to introduce 'cross-talk' errors, when you misdirect data or commands to or from the wrong box.

8.11 Data collection principles

My philosophy of data collection can be summarized in three rules:]

1. Never type in data;
2. Collect all the data you could possibly want, from the word go, in a structured format;
3. Worry about what to do with it later.

8.11.1 Relational databases

I have found the most useful way to store data is in a **relational database,** often called a relational database management system (RDBMS). A relational database stores data in **tables,** which are made up of *fields* and *records:*

A table:			*five fields:*		
	Date	Rat	NumResponses	NumStimuli	NumReinforcements
one record:	17/2/00 12:29:00	M4	56	5	1
another:	17/2/00 14:37:06	M5	437	43	8
… and so on	17/2/00 12:54:00	M4	263	26	5

The driving principle behind a relational database is this: **never duplicate data.** Let's say our rats came from two groups, Sham and Lesion. If we wanted to record this in the database, so we could analyse data by group, we could store it like this:

Table *BigData*

Date	Rat	Group	NumResponses	NumStimuli	NumReinforcements
17/2/00 12:29:00	M4	sham	56	5	1
17/2/00 14:37:06	M5	lesion	437	43	8
17/2/00 12:54:00	M4	sham	263	26	5

However, this introduces two problems. Firstly, it generates very large tables. Secondly, and more importantly, it is unclear what to do if the data is inconsistent – let's say the underlined 'sham' was changed to 'lesion' by mistake. The database would then not know whether rat M4 was in the Sham or Lesion group – there would be entries for both. The solution to both problems is to create two tables, *linked* on the smallest possible unit of information (in this example, the rat name):

Table *Responses*

Date	**Rat**	NumResponses	NumStimuli	NumReinforcements
17/2/00 12:29:00	**M4**	56	5	1
17/2/00 14:37:06	**M5**	437	43	8
17/2/00 12:54:00	**M4**	263	26	5

Table *Groups*

Rat	Group
M4	sham
M5	lesion

By using the rat name as a **key** (also known as a *foreign* key), the database can link the two tables together whenever we want to know how many responses the two groups made on average.

When we want to find out that sort of information, we **query** the database, specifying how we want to see the data. We could, for example, obtain the following (ignoring a glaring scientific error!):

Query *AverageByGroups*

Group	**N**umberOfSubjects	MeanNumResponses	MeanNumStimuli	MeanNumReinforcements
sham	2	159.5	15.5	3

lesion	1	437	43	8

Summary of database principles

So relational databases split up the data (which should be entered in well-designed tables without any duplication of information) from queries that look at the data in an infinite variety of ways.

A concrete example: Microsoft Access 97

Microsoft Access 97 is a commonly-used relational database for PCs. It isn't perfect, by a long shot, but I've found it good enough. It supports **structured query language (SQL)** for designing queries; this is a powerful quasi-English language. For example, the query shown above would be written in SQL like this:

```
SELECT group,
    count(*) as NumberOfRats,
    avg(NumResponses) as MeanNumResponses,
    avg(NumStimuli) as MeanNumStimuli,
    avg(NumReinforcements) as MeanNumReinforcements
FROM responses, groups
WHERE responses.rat = groups.rat
GROUP BY group
;
```

If you find all this a bit cryptic, Access also provides a graphical interface for designing queries.

Getting data out of a database

Given a well-designed database, you should be able to get the data out in any conceivable way. The size of this manual doesn't permit a detailed look at relational database design or queries, but there are abundant sources. If you use Microsoft Access, there's the help system, but I also recommend Viescas JL (1997), *Running Microsoft Access 97*, Microsoft Press. Beyond that there is a whole field of database design.

Tip

I operate on the principle that any view of the data is achievable. If the graphical query design can't do it, you can use SQL. If SQL can't do it alone, you can use Visual Basic to augment it. If all that fails (and it hasn't failed me yet) you can always re-export the data and use a general-purpose programming language to analyse it. If the data's there, you can get at it.

One thing is worth noting: modern statistical packages (e.g. SPSS, *http://www.spss.com/*) are starting to support the ODBC standard for exchanging information with databases. You can set up database queries to create views of the data that your stats packages can use, then set up sequences of ODBC capture, analysis and graphical presentation in your stats package. Then whenever you import new data, you can run the entire analysis in a matter of seconds. If you handle large volumes of data, it easily repays the initial effort.

8.11.2 Getting data into a database

This topic provides some notes that you may find helpful when importing data into relational databases such as Microsoft Access 97.

The direct method

It is possible for behavioural tasks to store data in a database *directly*. Many of the behavioural clients written for Whisker do just this (e.g. SecondOrder, SeekTake, ImpulsiveChoice, FiveChoice, etc.). They ask you which database you'd like to use, and then store data in tables in the correct format. They're very easy to use, and not that hard to program, either, if you use Visual C++ (which can set up classes for you to export data to a given database).

The indirect method

A method that needs less programming skill, and can apply to a great variety of languages and computers, is to export your data in an ASCII text file. The trick is to save the data in a format that is easily importable into a database. Regrettably, there is no easy way to save data destined for *multiple* database tables into a *single* text file. You could save data into a single file and process this file later, splitting it into several files for importing into a database. Alternatively, you could save one text file *per table*. In this case, the files should be saved in **comma-delimited** format, with the **field names on the first row.** Our example data might have come from two files looking like this:

response-Feb2000.txt

```
Date,Rat,NumResponses,NumStimuli,NumReinforcements
2/17/2000 12:29:00,M4,56,5,1
2/17/2000 14:37:06,M5,437,43,8
3/17/2000 12:54:00,M4,263,26,5
```

groups.txt

```
Rat,Group
M4,sham
M5,lesion
```

It is very easy to import data in this format into Access (choose *File* → *Get External Data* → *Import*, choose your text file, ensure that you choose *Delimited* format and that *First Row Contains Field Names* is selected; then simply choose the table to put the data in).

Amalgamating multiple files

A problem I had with this method is that it did take a long time to import every single file. So I wrote a small utility called **amalgam.pl** (written in the language Perl) to amalgamate lots of similar files that were all destined for the same database table, in such a way that the header row (with the field names) only appeared once.

This tool is supplied with Whisker (by default, somewhere within *\Program Files\WhiskerControl*). To use it, you will need to obtain Perl (from *http://www.ActiveState.com/*) and install it. (I'm afraid that their distribution terms prevent me from supplying you with a copy of Perl together with Whisker.)

You can either copy `amalgam.pl` to a directory that is on your path, or you can add the directory containing `amalgam.pl` to the path. To do the latter, choose *Control Panel* → *System*

→ *Environment*. Click on the Path variable and edit the *Value* field to append the directory that Amalgam is in, using semicolons to separate consecutive path entries (e.g. *;d:\program files\rudolf cardinal\whisker\amalgam;*). Then click *Set* and *OK*. If you change the System path, you'll affect all users of the system; if you change the User path, only your own user. Only administrators can change the system path.

Once you have that set up, simply fire up a command prompt and change to the directory containing your data. Let's say you have a whole host of response data files, called `response-Jan2000.txt`, `response-Feb2000.txt`, `response-Mar2000.txt`, and so on. You can issue this command:

```
amalgam.pl big-response-file.txt response*.txt
```

and it will create a single file, `big-response-file.txt`, which you can import into Access in one step. If you want to import into five tables, you are aiming to generate five files. (Amalgam will not let you merge files that have different headers.)

A note on dates

Access's text import facility will only recognize dates as such if they are in a very particular format. For the UK, it is this:

17/03/2000 15:30:05

and this depends on telling Windows that you are in the UK in *Control Panel → Regional Settings*. (Specifically, Access recognises the 'short date' and 'short time' pictures set up in Control Panel.) It is so useful to have date/time-stamping of your data (and so awkward to configure the Access's import facility differently) that I strongly suggest you configure your regional settings and write your behavioural tasks to save data to disk in this format.

8.11.3 Recovering data from old applications

Data in ASCII format

Of course, you may not have the luxury of redesigning your task to store data in a machine-friendly format. If you're lucky enough to have electronic output in ASCII form, it may not be in neat comma-delimited files but look more like this:

```
d62-990519.txt
Box 6: 62 - ; Wed,19 May 1999.12:41:50, 200 min session
RIGHT Lever active: Overall FR 1, FR 1 for CS
Reinforcements per spell: 1
CS duration (csec): 100
Reinforcement delay (csec): 100
Lever retraction time (csec): 2000
Reinforcement duration (csec): 728

Active: 59     Inactive: 13     Stimuli: 44     Reinforcements: 44

...
```

This is part of an output file from an old second-order schedule of reinforcement written in Arachnid. All great for a human to read, but a pain for a computer. It can be handy to have a

human-readable file like this, but this should only be in addition to a computer-readable format.

My usual solution to this problem is to write a mini-program in **Perl,** a language especially designed for searching for patterns in text files and extracting data from them (see Getting data into a database for a discussion of using Perl tools). For the example above, I have a Perl script that scans through, extracts the data and creates a set of comma-delimited files, which can be amalgamated and imported into a database. (Perl is free and quite easy to learn; see the bibliography at the end of this guide for details.) My new Whisker-based SecondOrder task stores data directly into a database (its computer-readable format), and keeps a human-readable summary file that looks like this:

```
m4-03Mar2000-1020-summary.txt
SECOND-ORDER I.V. SELF-ADMINISTRATION – SUMMARY FILE
...
Rat: m4
Session: 24
Date/time code: 03-Mar-2000(1019)
...
Session time limit (min): 120
Locomotor time bin size (min): 20
...
Reinforcement device: PUMP
Max #reinforcements: 10
Pump infusion duration (s): 5.83
...
Number of stimuli: 41
Number of reinforcements: 4
Finished at: Fri 03-Mar-2000, 12:39
Active lever responses: 454
Inactive lever responses: 87
```

which is one of the easiest formats to read into Perl if you needed to (each file follows exactly the same format; there's one piece of information per line; there are a few sections with comma-delimited output that can be copied directly into a file for database import).

Data not in ASCII format

If the numbers that make up your data set are stored in some obscure format, it's still possible to extract them, but you may end up having to write a program in a general-purpose language like C or C++ to extract them. This is beyond the scope of this guide.

8.11.4 Data integrity in relational databases

To ensure data integrity, relational databases offer you two facilities.

Field definitions

When tables are created, the *data type* of each field must be defined – whether it's a number, a string, or a date, for example. In addition, many databases (including Access) offer a wide range of further data-checking facilities; for example, you can require that the field be entered (not left blank), specify minimum and maximum values, specify the maximum length of a string, constrain a string to certain values, and so on.

Indices

You create an index for two reasons. One is to speed up searches – creating an index like this has no effect on the data itself. The other is to enforce data integrity rules.

Indices can be used to ensure that *duplicates* cannot be entered (such an index is called a **unique index,** because the items it indexes must all be unique). Indices may be created on several fields; for example, a unique index on {Rat, Session, Trial} fields would ensure that the combination of those three fields would have to yield a unique value.

Indices are often thought of as being involved in *relationships* between tables, particularly when such relationships are one-to-one or many-to-one. If you have one table that relates rats to their experimental groups, and another that contains the session-by-session data, the two tables might be linked using the *Rat* field (as in the example we used earlier). It would be usual for the *Group* table to have a unique index for the *Rat* field. As this is also the field or *key* used to look up individual records, it is often called the **primary key.** Nevertheless, the index is only involved in the relationship to ensure the uniqueness of this key and to speed up the process of searching data; it has nothing to do with the end results of looking up the data.

8.12 Networking in Perl

Obtaining and installing Perl

Perl for Win32 is available free, from *http://www.ActiveState.com/*. Follow the installation instructions that accompany the software.

Writing code for Perl and running Perl programs

Essentially, you use a text editor to create a Perl script, usually called `something.pl`. Then you can simply type `something.pl` from the command line, passing it further arguments as necessary.

Example Perl scripts for Whisker

A few examples follow, of which **whisker_perl_demo.pl** is the clearest.

8.12.1 whisker_perl_demo.pl

```
#!/usr/bin/perl -w
# whisker_perl_demo.pl
# 7 Feb 2010
# ---------------------------
# Syntax:     whisker_perl_demo.pl [server [port]]
#
# Note that Perl comments are preceded by a hash (#)

require 5.002; # Require this version of Perl.
use strict; # Enable strict syntax checking.
use Socket; # Use the Socket module for TCP/IP communications.
use IO::Handle; # IO::Handle provides the autoflush command.
use DBI; # Database access functions.
use DBI qw(:sql_types); # Use SQL keywords.
# use DBI qw(:get_info); # for DBI::SQL_DBMS_NAME; not yet supported; see http://
```

```
www.mail-archive.com/dbi-dev@perl.org/msg00624.html

my $server = shift || 'localhost'; # Get server name or supply default.
my $mainport = shift || 3233; # Get (main) port number or supply default.
my ($immport, $line, $event, $reply); # Declare other variables.
my $DEBUGNETWORK = 1; # Controls verbose printing of network communications
my $DEBUGDATABASE = 1; # Controls verbose printing of database communications

print "Whisker demo in Perl\n";
print "--------------------\n";

# Declare handles
our ($OUTFILE, $MAINSOCK, $IMMSOCK);

# Open text file.
print "--- Enter text-based results file details.\n";
my $textfile = AskUser("Text results file", "whiskertemp.txt");
open($OUTFILE, ">$textfile");

# Open database connection.
print "--- Enter database details.\n";
my $driver = AskUser("Database driver for DBI (e.g. mysql, ODBC) (NB case-
sensitive)", "ODBC");
my $database = AskUser("Name of database", "testdb");
my $username = AskUser("Database username", "root");
my $password = AskUserPassword("Database password");
my $dbh = ConnectToDatabase($driver, $database, $username, $password);

# Open network connection.
$immport = ConnectBothPorts($server, $mainport);

# Initial commands to server: claiming lines and setting up the task.
Send("ReportName Whisker Perl demo program");
Send("ReportStatus Absolutely fine.");
Send("WhiskerStatus");
$reply = SendImmediate("TimerSetEvent 1000 9 TimerFired");
$reply = SendImmediate("TimerSetEvent 12000 0 EndOfTask");
Send("TestNetworkLatency");

# Enter a loop to listen to messages from the server.
while (chomp($line = <$MAINSOCK>)) {
       if ($DEBUGNETWORK) { print "SERVER: $line\n"; } # For info only.
       if ($line =~ /^Ping/) {
              # If the server has sent us a Ping, acknowledge it.
              Send("PingAcknowledged");
       }
       if ($line =~ /^Event: (.+)/) {
              # The server has sent us an event.
              $event = $1;
              if ($DEBUGNETWORK) { print "EVENT RECEIVED: $event\n"; } # For info
only.

              # Event handling for the behavioural task is dealt with here.
              if ($event =~ /^EndOfTask$/) { last; } # Exit the while loop.
       }
}
```

```
# Create some sample data.
# For arrays of arrays, see http://perldoc.perl.org/perldsc.html#ARRAYS-OF-ARRAYS
# For passing array references to functions, see http://www.cs.cf.ac.uk/Dave/PERL/
node61.html
my $table = "table1";
my @fields = ("field1","field2","field3");
my @values = (
              [ "data1","data2","data3" ],
              [ "data4","data5","data6" ],
              [ "data7","data8","data9" ],
       );

# Write data to disk/database. Do these separately, disk first, in case there are
database problems.
foreach my $i (0..$#values) {
       my @record = @{$values[$i]}; # Long-winded for illustration
       WriteRecordToCSVFile($OUTFILE, ($i == 0), \@fields, \@record);
}
foreach my $i (0..$#values) {
       SQLInsertRecord($dbh, $table, \@fields, \@{$values[$i]});
}

LogOut();
print "----------------\n";
print "Finished.\n";
exit;

# ------------------------------------------------------------
# --------------------------- Networking routines.
# ------------------------------------------------------------

sub ConnectBothPorts {
       my $server = shift; # Fetch first parameter
       my $mainport = shift; # Fetch next parameter

       my ($immport, $code, $line); # Declare other local variables.
       ConnectMain($server, $mainport); # Log in to the server.
       # Listen to the server until we can connect the immediate socket.
       while (chomp($line = <$MAINSOCK>)) { # chomp removes the trailing newline
              # The server has sent us a message via the main socket.
              if ($DEBUGNETWORK) { print "SERVER: $line\n"; } # Print the message,
for info only.
              if ($line =~ /^ImmPort: (\d+)/) { $immport = $1; }
              if ($line =~ /^Code: (\w+)/) {
                     $code = $1;
                     ConnectImmediate($server, $immport, $code);
                     last; # Exit the while loop.
              }
       }

       return $immport;
}

sub ConnectMain {
       my $server = shift; # Fetch first parameter.
       my $port = shift; # Fetch next parameter.
```

```
        print "Connecting main port to server.\n";
        if ($port =~ /\D/) { $port = getservbyname($port, 'tcp'); }
        die "No port" unless $port;
        my $iaddr = inet_aton($server) || die "No host: $server";
        my $paddr = sockaddr_in($port, $iaddr);
        my $proto = getprotobyname('tcp');
        socket($MAINSOCK, PF_INET, SOCK_STREAM, $proto) || die "Can't make socket:
$!";
        connect($MAINSOCK, $paddr) || die "Can't connect: $!";
        print "Connected to main port $port on " . inet_ntoa($iaddr) . "\n";
        use Socket qw(IPPROTO_TCP TCP_NODELAY);
        setsockopt($MAINSOCK, IPPROTO_TCP, TCP_NODELAY, 1); # Disable the Nagle
algorithm.
        $MAINSOCK->autoflush(1); # Ensure that output to the socket gets sent
immediately.
}

sub ConnectImmediate {
        my $server = shift;
        my $port = shift;
        my $code = shift;

        print "Connecting immediate port to server.\n";
        if ($port =~ /\D/) { $port = getservbyname($port, 'tcp'); }
        die "No port" unless $port;
        my $iaddr = inet_aton($server) || die "No host: $server";
        my $paddr = sockaddr_in($port, $iaddr);
        my $proto = getprotobyname('tcp');
        socket($IMMSOCK, PF_INET, SOCK_STREAM, $proto) || die "Can't make socket:
$!";
        connect($IMMSOCK, $paddr) || die "Can't connect: $!";
        print "Connected to immediate port $port on " . inet_ntoa($iaddr) . "\n";
        use Socket qw(IPPROTO_TCP TCP_NODELAY);
        setsockopt($IMMSOCK, IPPROTO_TCP, TCP_NODELAY, 1);
        $IMMSOCK->autoflush(1);
        SendImmediate("Link $code");
}

sub LogOut {
        close ($MAINSOCK) || die "Error closing main socket: $!";
        close ($IMMSOCK) || die "Error closing immediate socket: $!";
}

sub Send {
        # Send something to the server on the main socket, with a trailing newline.
        if ($DEBUGNETWORK) { print STDOUT "Main socket command: @_\n"; } # For info
only.
        print $MAINSOCK "@_\n";
}

sub SendImmediate {
        # Send a command to the server on the immediate socket, and retrieve its
reply.
        if ($DEBUGNETWORK) { print STDOUT "Immediate socket command: @_\n"; } # For
```

```
info only.
	print $IMMSOCK "@_\n";
	chomp(my $reply = <$IMMSOCK>);
	if ($DEBUGNETWORK) { print STDOUT "Immediate socket reply: $reply\n"; } #
For info only.
	return $reply;
}

# ------------------------------------------------------------
# -------------------------- User interface routines.
# ------------------------------------------------------------

sub AskUser {
	print "$_[0] [$_[1]]: ";
	my $rc = <>;
	chomp $rc;
	if($rc eq "") { $rc = $_[1]; }
	return $rc;
}

sub AskUserPassword {
	# http://www.perlmonks.org/?node_id=352298
	$|=1; #Turn on AutoFlush
	use Term::ReadKey;
	ReadMode('noecho');
	ReadMode('raw');
	my $pass = '';
	print "$_[0]: ";
	while (1) {
		my $c;
		1 until defined($c = ReadKey(-1));
		last if (($c eq "\n") || ($c eq "\r")); # Windows machines: Enter
gives \r, not \n!
		print "*";
		$pass .= $c;
	}
	ReadMode('restore');
	print "\n";

	return $pass;
}

# ------------------------------------------------------------
# -------------------------- Database routines, using ODBC.
# ------------------------------------------------------------

sub ConnectToDatabase {
	my $driver = shift;
	my $database = shift;
	my $username = shift;
	my $password = shift;

	print "Connecting to database.\n";
	my $dbh = DBI->connect('DBI:'.$driver.':'.$database,$username,$password) or
die "Failed to connect to database: $database\n"; # *** weaken "die"
```

```
	my $dbtype = $dbh->get_info(17); # 17 is DBI::SQL_DBMS_NAME
	print "Connected to database: $driver:$database. Database type is:
$dbtype\n";

	return $dbh;
}

sub SQLInsertRecord {
	my $dbh = shift;
	my $table = shift;
	my $ref_fields = shift; my @fields = @{$ref_fields};
	my $ref_values = shift; my @values = @{$ref_values};

	my $fields_size = @fields;
	my $values_size = @values;
	if ($fields_size != $values_size) { die "SQLInsertRecord failed: size of
fields array differs from size of values array\n"; }
	# SQL ultimately e.g. INSERT INTO table (field1, field2, field3) VALUES
(value1, value2, value3);
	# but we start with INSERT INTO table (field1, field2, field3) VALUES (?, ?,
?);
	my $sqlstring = "INSERT INTO $table (";
	foreach my $i (0..$#fields) {
		if ($i > 0) { $sqlstring .= ", "; }
		$sqlstring .= $fields[$i];
	}
	$sqlstring .= ") VALUES (";
	foreach my $i (0..$#values) {
		if ($i > 0) { $sqlstring .= ", "; }
		$sqlstring .= "?";
	}
	# if ($DEBUGDATABASE) { print "SQL so far: $sqlstring\n"; }
	$sqlstring .= ");";
	my $insert_sql = $dbh->prepare($sqlstring);
	foreach my $i (0..$#values) {
		$insert_sql->bind_param($i+1, $values[$i]); # refers to the i'th ? in
the query; bind_param indexed from 1, not 0
	}
    $insert_sql->execute(); # excute the SQL statement
}

# ------------------------------------------------------------
# --------------------------- Textfile results storage.
# ------------------------------------------------------------

sub WriteRecordToCSVFile {
	my $OUTFILE = shift;
	my $firstrecord = shift;
	my $ref_fields = shift; my @fields = @{$ref_fields};
	my $ref_values = shift; my @values = @{$ref_values};

	my $fields_size = @fields;
	my $values_size = @values;
	if ($fields_size != $values_size) { die "WriteRecordToCSVFile failed: size
of fields array differs from size of values array\n"; }
```

```
        if ($firstrecord) { # For the first record, print the field names
               foreach my $i (0..$#fields) {
                      if ($i > 0) { print $OUTFILE ","; }
                      print $OUTFILE $fields[$i];
               }
               print $OUTFILE "\n";
        }
        # Print the values in comma-separated format
        # could use (0..($values-1)) or (0..$#values) to iterate through
        foreach my $i (0..$#values) {
               if ($i > 0) { print $OUTFILE ","; }
               print $OUTFILE $values[$i];
        }
        print $OUTFILE "\n";
}
```

8.12.2 whiskerstat.pl

A simple TCP client: implementing a status program in Perl

Here's an example of network communications using Perl. You can run it as

```
whiskerstat.pl somewhere.psychol.cam.ac.uk
```

```
#!/usr/bin/perl –w
# PIT.pl
# --------------------------- Call this file "WhiskerStat.pl".
# Note that Perl comments are preceded by a hash (#)
require 5.002;
use Socket;

# --------------------------- Say hello

print "\n\nWhiskerStat.pl\n\n";

&LogIn(@ARGV);

&Send("ReportName WhiskerStat status program");
&Send("ReportStatus Absolutely fine.");
&Send("WhiskerStatus");
&Send("TestNetLatency");

while ($line = <SOCK>) {
    # The server has sent us a message. Deal with it.
    if ($line =~ /Ping/) { &Send("PingAcknowledged"); }
    print $line;
}

&LogOut();
print "All done.\n";
exit;

# ------------------------------------------------------------
# --------------------------- Networking routines.
# ------------------------------------------------------------

sub LogIn {
    # Get parameters from the command line, or acceptable defaults
```

```
        my ($remote,$port, $iaddr, $paddr, $proto, $line);
        $remote  = shift || 'localhost';
        $port    = shift || 1333;  # default port
        if ($port =~ /\D/) { $port = getservbyname($port, 'tcp') }
        die "No port" unless $port;

        # Connect to the server

        $iaddr   = inet_aton($remote)               || die "no host: $remote";
        $paddr   = sockaddr_in($port, $iaddr);
        $proto   = getprotobyname('tcp');
        socket(SOCK, PF_INET, SOCK_STREAM, $proto)  || die "can't make socket: $!";
        connect(SOCK, $paddr)    || die "can't connect: $!";
        print "Connected to " . inet_ntoa($iaddr) . "\n\n";

        select(SOCK); $| = 1;
        # This is vital to ensure that output to the socket
        # gets sent immediately. Otherwise it hangs around for ever
        # (literally) in some circumstances.

        select(STDOUT);
        # Output goes to the local console once more.
    }

    sub LogOut
    {
        close (SOCK)            || die "close: $!";
    }

    sub Send {
        # This sends something to the server, with a trailing newline.
        # Use it like Send("Hello ","I am " . " a ", "fish");
        print SOCK @_;
        print SOCK "\n";
    }
```

8.12.3 berlindummyserver.pl

```
#!/usr/bin/perl -w
# ---------------------------
# Syntax:     berlindummyserver.pl [server [port]]
# ---------------------------
# 16 May 2010
#
# Note that Perl comments are preceded by a hash (#)

require 5.002; # Require this version of Perl.
use strict; # Enable strict syntax checking.
use Socket; # Use the Socket module for TCP/IP communications.
use IO::Handle; # IO::Handle provides the autoflush command.
use IO::Socket::INET;

my $numinputs = 16;
my $numoutputs = 8;
my (@inputstate, @outputstate);
for (my $i = 0; $i < $numinputs; ++$i) {
       $inputstate[$i] = 0;
}
for (my $i = 0; $i < $numoutputs; ++$i) {
```

```
	$outputstate[$i] = 0;
}
# Special
$inputstate[2]=1;
$inputstate[5]=1;
$outputstate[2]=1;

my $port = shift || 5002; # Get (main) port number or supply default (default as
per York Winter's email of 17/5/10).
my ($client, $command, $reply, $channel, $duration, $state, $hostinfo); # Declare
other variables.
my $DEBUGNETWORK = 1; # Controls verbose printing of network communications
my $delimiter = "\n\r"; # CR (\n), LF (\r)

print "Berlin dummy server in Perl\n";
print "---------------------------\n";

# Open network connection.
# http://www.osix.net/modules/article/?id=101
# Listen = number of pending connections; SOMAXCON is a special symbol for the
system maximum
# Reuse = restart the server manually
my $server = IO::Socket::INET->new( Proto => 'tcp', LocalPort => $port, Listen =>
SOMAXCONN, Reuse => 1);
die "can't setup server" unless $server;
print "[Server $0 is running]\n";

# Now wait for a connection. (Can telnet to this server as a test.)
while ($client = $server->accept()) {
	$client->autoflush(1);
	my ($name, $aliases, $addrtype, $length, @addrs);
	($name, $aliases, $addrtype, $length, @addrs) = gethostbyaddr($client-
>peeraddr, AF_INET);
	printf "[Connect from %s]\n", $name;

	# Enter a loop to listen to messages from the server.
	while ($command = <$client>) {
		$command =~ s/\s+$//; # remove trailing rubbish
		$command =~ s/^\s+//; # remove leading rubbish
		# don't use chomp (which only removes \n); instead, remove CR, LF,
any other trailing whitespace
		# http://www.wellho.net/forum/Perl-Programming/New-line-characters-
beware.html
		if ($DEBUGNETWORK) {
			print "CLIENT: $command\n"; # For info only.
			# print "String length = " . length($command) . "\n";
		}
		if ($command =~ /^GetNumberOfInputChannels/) {
			Send("NumberOfInputChannels 16");
		}
		if ($command =~ /^GetNumberOfOutputChannels/) {
			Send("NumberOfOutputChannels 8");
		}
		if ($command =~ /^SetChannelOn (.+)/) {
			$channel = $1;
			if ($channel >= 0 && $channel < $numinputs) {
```

```
                        $outputstate[$channel] = 1;
                        print "Output channel " . $channel . " set ON\n";
                    }
                    # Now a silly debug bit
                    if ($channel == 5 || $channel == 6) {
                        $inputstate[4] = !$inputstate[4];
                        Send("SensorState 4 " . $inputstate[4]);
                    }
                    # No response
                }
                if ($command =~ /^SetChannelOnPulse (.+) (.+)/) {
                    $channel = $1;
                    $duration = $2;
                    if ($channel >= 0 && $channel < $numinputs) {
                        $outputstate[$channel] = 1;
                        print "Output channel " . $channel . " set ON for pulse
of " . $duration . " (NOT REALLY IMPLEMENTED)\n";
                        # should set it off later, but never mind ***
                    }
                    # No response
                }
                if ($command =~ /^SetChannelOff (.+)/) {
                    $channel = $1;
                    if ($channel >= 0 && $channel < $numinputs) {
                        $outputstate[$channel] = 0;
                        print "Output channel " . $channel . " set OFF\n";
                    }
                    # No response
                }
                if ($command =~ /^GetSensorState (.+)/) {
                    $channel = $1;
                    if ($channel >= 0 && $channel < $numinputs) {
                        Send("SensorState " . $channel . " " . $inputstate
[$channel]);
                    }
                }
                # Can also send SensorState messages spontaneously ***
            }
            close ($client) || die "Error closing socket: $!";
            print "[Disconnected.]\n";
        }
        # use CTRL-C to shut down the server
        exit;

        sub Send {
            # Send something to the server on the main socket, with a trailing newline.
            if ($DEBUGNETWORK) { print STDOUT "Message to client: @_\n"; } # For info
only.
            print $client "@_" . $delimiter;
        }

        # -------------------------------------------------------------
        # --------------------------- User interface routines.
        # -------------------------------------------------------------
```

```
sub AskUser {
       print "$_[0] [$_[1]]: ";
       my $rc = <>;
       chomp $rc;
       if($rc eq "") { $rc = $_[1]; }
       return $rc;
}
```

8.13 Python as a Whisker client

Python is obtainable from http://www.python.org/.

We suggest collecting Whisker-specific Python routines into one file, **whisker.py**, and then linking to this from task scripts, as in the following examples.

8.13.1 whisker.py

```
# whisker.py
# 7 Feb 2010

# ------------------------------------------------------------
# --------------------------- Networking routines.
# ------------------------------------------------------------

import re
def connect_both_ports(server, mainport):
    """Connect the main and immediate ports to the server."""
    connect_main(server, mainport) # Log in to the server.
    # Listen to the server until we can connect the immediate socket.
    for line in getlines_mainsock():
        # The server has sent us a message via the main socket.
        if verbosenetwork:
            print "SERVER: " + line # Print the message, for info only.
        m = re.search(r"^ImmPort: (\d+)", line)
        if m:
            immport = m.group(1)
        m = re.search(r"^Code: (\w+)", line)
        if m:
            code = m.group(1)
            connect_immediate(server, immport, code)
            break

import socket
def connect_main(server, portstring):
    """Connect the main port to the server."""
    print "Connecting main port to server."
    m = re.match(r"\D", portstring) # search for \D = non-digit characters
    if m:
        port = socket.getservbyname(portstring, "tcp")
    else:
        port = int(portstring)
    proto = socket.getprotobyname("tcp")
    global mainsock
    try:
        mainsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, proto)
```

```
        mainsock.connect( (server, port) )
    except socket.error, msg:
        mainsock.close()
        mainsock = None
        print "ERROR creating/connecting main socket: "+msg
    print "Connected to main port " + str(port) + " on server " + server
    mainsock.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1) # Disable the
Nagle algorithm.

import time
def connect_immediate(server, portstring, code):
    m = re.match(r"\D", portstring) # search for \D = non-digit characters
    if m:
        port = socket.getservbyname(portstring, "tcp")
    else:
        port = int(portstring)
    proto = socket.getprotobyname("tcp")
    global immsock
    try:
        immsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, proto)
        immsock.connect( (server, port) )
    except socket.error, msg:
        immsock.close()
        immsock = None
        print "ERROR creating/connecting immediate socket: "+msg
    print "Connected to immediate port " + str(port) + " on server " + server
    immsock.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1) # Disable the
Nagle algorithm.
    immsock.setblocking(True)
    send_immediate("Link "+code)
    sleeptime = 0.1
    print "Sleeping for "+str(sleeptime)+" seconds as the Nagle-disabling feature
of Python isn't working properly..."
    time.sleep(sleeptime) # the Nagle business isn't working; the Link packet is
getting amalgamated with anything the main calling program starts to send. So
pause.
    print "... continuing. Immediate socket should now be correctly linked."

def log_out():
    try:
        mainsock.close()
    except socket.error, msg:
        print "Error closing main socket: "+msg
    try:
        immsock.close()
    except socket.error, msg:
        print "Error closing immediate socket: "+msg

def send(s):
    # Send something to the server on the main socket, with a trailing newline.
    if verbosenetwork:
        print "Main socket command: " + s # For info only.
    mainsock.send(s + "\n")

def send_immediate(s):
    # Send a command to the server on the immediate socket, and retrieve its
reply.
    if verbosenetwork:
        print "Immediate socket command: " + s # For info only.
    immsock.sendall(s + "\n")
```

```
    reply = getlines_immsock().next()
    if verbosenetwork:
        print "Immediate socket reply:", reply # For info only.
    return reply

# http://stackoverflow.com/questions/822001/python-sockets-buffering
def getlines_immsock():
    # Yield a set of lines from the socket
    buffer = immsock.recv(4096)
    done = False
    while not done:
        if "\n" in buffer:
            (line, buffer) = buffer.split("\n", 1)
            yield line
        else:
            more = immsock.recv(4096)
            if not more:
                done = True
            else:
                buffer = buffer+more
    if buffer:
        yield buffer

def getlines_mainsock():
    # Yield a set of lines from the socket
    buffer = mainsock.recv(4096)
    done = False
    while not done:
        if "\n" in buffer:
            (line, buffer) = buffer.split("\n", 1)
            yield line
        else:
            more = mainsock.recv(4096)
            if not more:
                done = True
            else:
                buffer = buffer+more
    if buffer:
        yield buffer

# --------------------------------------------------------------
# ------------------------- User interface routines.
# --------------------------------------------------------------

def ask_user(prompt, default):
    """Prompts the user, with a default. Returns a string."""
    result = raw_input(prompt + " [" + default + "]: ")
    return result if len(result)>0 else default

def ask_user_password(prompt):
    """Read a password from the console."""
    import getpass
    return getpass.getpass(prompt + ": ")

# --------------------------------------------------------------
# ------------------------- Database routines, using ODBC.
# --------------------------------------------------------------

# Open database connection.
```

```
# There's no direct Python equivalent of DBI (which can talk to e.g. ODBC and
MySQL).
# So we'll use ODBC. On Windows, that comes by default.
# For IODBC: sudo apt-get install iodbc libiodbc2-dev. Then see http://www.iodbc.
org/dataspace/iodbc/wiki/iODBC/IODBCPythonHOWTO . This uses ~/.odbc.ini .
# Or, for MySQL, sudo apt-get install iodbc libmyodbc. Then see https://help.
ubuntu.com/community/ODBC . This uses /etc/odbc.ini .
# Or UnixODBC: sudo apt-get install unixodbc unixodbc-dev. Then see http://ubuntu-
virginia.ubuntuforums.org/showthread.php?p=5846508 . WE'RE GOING THIS WAY.
# The Python interface to ODBC is pyodbc: http://code.google.com/p/pyodbc/wiki/
GettingStarted
# To install pyodbc (see in part http://www.easysoft.com/developer/languages/
python/pyodbc.html):
#   * sudo apt-get install python-all-dev (to get development headers)
#   * download e.g. pyodbc-2.1.6.zip from http://code.google.com/p/pyodbc/
downloads/list
#   * unzip pyodbc-2.1.6.zip
#   * cd pyodbc-2.1.6
#   * amend setup.py: FOR IODBC: change "libraries.append('odbc')" to "libraries.
append('iodbc')"...
#   * amend setup.py: FOR LIBMYODBC: not yet worked out
#   * amend setup.py: FOR UNIXODBC: works as is
#   * sudo python setup.py install
# Now, for unixodbc, set it up:
#   * edit /etc/odbcinst.ini to be e.g.:
#       [myodbc]
#       Description = MySQL ODBC 3.51 Driver (this can be an arbitrary name)
#       Driver = /usr/lib/odbc/libmyodbc.so
#       Setup = /usr/lib/odbc/libodbcmyS.so
#       FileUsage       = 1
#   * edit /etc/odbc.ini to be e.g.
#       [mysql-testdb]
#       Driver       = myodbc
#       Description  = mysql_egret_testdb NEEDS SSH TUNNEL
#       SERVER       = 127.0.0.1 # do not use "localhost" or the driver will look
in /var/run/mysqld/mysqld.sock, instead of looking at PORT
#       PORT         = 3306
#       Database     = testdb
#       OPTION       = 3
# Now test:
#   * isql mysql-testdb USER PASSWORD
#   * python tests/dbapitests.py python tests/dbapitests.py "DSN=mysql-testdb;
UID=xxx;PWD=xxx"

import pyodbc
def connect_to_database():
    print "--- Enter database details."
    dsn = ask_user("ODBC data source name (DSN)", "mysql-testdb")
    username = ask_user("Database username", "root")
    password = ask_user_password("Database password")
    print "Connecting to database."
    dbh = pyodbc.connect("DSN={0};UID={1};PWD={2}".format(dsn, username,
password))
    # XXX print error message if it fails
    # XXX print success message it it succeeds
    # ... at present, it crashes with the error message.
    return dbh

def sql_insert_record(dbh, table, fields, values):
    if len(fields) != len(values):
```

```
        print "Field/value mismatch to sql_insert_record()"
        return
    # SQL e.g. INSERT INTO table (field1, field2, field3) VALUES (value1, value2,
value3);
    # but we start with INSERT INTO table (field1, field2, field3) VALUES
(?, ?, ?);
    sql = "INSERT INTO " + table + " ("
    for i in range(len(fields)):
        if i>0:
            sql += ", "
        sql += fields[i]
    sql += ") VALUES ("
    for i in range(len(values)):
        if i>0:
            sql += ", "
        sql += "?"
    sql += ");"
    if verbosedatabase:
        print "SQL so far (? will be bound to values later): " + sql

    cursor = dbh.cursor()
    cursor.execute(sql, values) # binds the ? to values in the process
    dbh.commit()

def sql_insert_multiple_records(dbh, table, fields, records):
    for record in records:
        sql_insert_record(dbh, table, fields, record)

# -----------------------------------------------------------
# --------------------------- Textfile results storage.
# -----------------------------------------------------------

# Trivial, but...
def produce_csv_output(filehandle, fields, values):
    """Produce CSV output, without using csv.writer, so the log can be used for
lots of things."""
    output_csv(filehandle, fields)
    for row in values:
        output_csv(filehandle, row)

def output_csv(filehandle, values):
    line = ""
    for i in range(len(values)):
        if i>0:
            line += ","
        line += values[i]
    filehandle.write(line+"\n")
```

8.13.2 whisker_python_demo.py

```
#!/usr/bin/python
# whisker_python_demo.py
# 7 Feb 2010
# --------------------------
# Syntax:    whisker_perl_demo.pl [server [port]]
#
# Note that Python comments are preceded by a hash (#)
# Python quick guides and reference:
```

```
#    http://www.intuit-symbiosis.org/computing/?uid=9
#    http://docs.python.org/tutorial/
# Standard conventions:
# - 4 spaces, not tabs
# - CamelCase for classes, lower_case_underscore for functions

import whisker # read in whisker.py from current directory or PYTHONPATH
# If you don't have it, fetch it from http://www.whiskercontrol.com/examples/
whisker.py

# ------------------------------------------------------------
# Main program
# ------------------------------------------------------------

# Fetch command-line options.
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-s", "--server",
                  action="store", type="string", dest="server",
default="localhost",
                  help="connect to Whisker server SERVER", metavar="SERVER")
parser.add_option("-p", "--port",
                  action="store", type="string", dest="port", default="3233",
                  help="use TCP/IP port PORT", metavar="PORT")
parser.add_option("-v", "--verbosenetwork",
                  action="store_true", dest="verbosenetwork", default=True,
                  help="show verbose network messages")
parser.add_option("-q", "--quietnetwork",
                  action="store_false", dest="verbosenetwork",
                  help="suppress network messages")
parser.add_option("-d", "--verbosedatabase",
                  action="store_true", dest="verbosedatabase", default=True,
                  help="show verbose database messages")
parser.add_option("-x", "--quietdatabase",
                  action="store_false", dest="verbosedatabase",
                  help="suppress database messages")
(options, args) = parser.parse_args()
# We now have what we want in (e.g.) options.server and options.port
# Pass info to the whisker namespace.
whisker.verbosenetwork = options.verbosenetwork
whisker.verbosedatabase = options.verbosedatabase

print "Whisker demo in Python"
print "----------------------"

# Open log file.
print "\n--- Let's open a log file."
logfilename = whisker.ask_user("Text results file", "whiskertemp.txt")
logfile = open(logfilename, "w")

# Open database connection.
print "\n--- Let's open a database connection."
dbh = whisker.connect_to_database()

# Open network connection.
print "\n--- Let's connect to Whisker."
whisker.connect_both_ports(options.server, options.port)

# Initial commands to server: claiming lines and setting up the task.
```

```
print "\n--- We're connected. Let's set up a task."
whisker.send("ReportName Whisker python demo program")
whisker.send("ReportStatus Absolutely fine.")
whisker.send("WhiskerStatus")
reply = whisker.send_immediate("TimerSetEvent 1000 9 TimerFired")
reply = whisker.send_immediate("TimerSetEvent 12000 0 EndOfTask")
whisker.send("TestNetworkLatency")

# Enter a loop to listen to messages from the server.
print "\n--- Let's listen to the server and process its events."
for line in whisker.getlines_mainsock():
    if whisker.verbosenetwork:
        print "SERVER: " + line # For info only.
    if line == "Ping":
        # If the server has sent us a Ping, acknowledge it.
        whisker.send("PingAcknowledged")
    if line[:7] == "Event: ":
        # The server has sent us an event.
        event = line[7:]
        if whisker.verbosenetwork:
            print "EVENT RECEIVED: " + event # For info only.
        # Event handling for the behavioural task is dealt with here.
        if event == "EndOfTask":
            break # Exit the while loop.

# Create some sample data.
# For arrays of arrays, see http://perldoc.perl.org/perldsc.html#ARRAYS-OF-ARRAYS
# For passing array references to functions, see http://www.cs.cf.ac.uk/Dave/PERL/
node61.html
table = "table1"
fields = ["field1","field2","field3"]
values = [
          [ "data1","data2","data3" ],
          [ "data4","data5","data6" ],
          [ "data7","data8","data9" ],
         ]

# Write data to disk/database. Do these separately, disk first, in case there are
database problems.
print "\n--- Let's write to the log file."
whisker.produce_csv_output(logfile, fields, values)

print "\n--- Let's write to the database."
whisker.sql_insert_multiple_records(dbh, table, fields, values)

print "\n--- Let's log out from the Whisker server."
whisker.log_out()
print "----------------"
print "Finished."
```

Part IX

SDK User's Guide

9 SDK User's Guide

A more detailed documentation for the SDK will be provided in the next update of the Whisker Documentation

The Whisker SDK provides a simple tool for accessing the functions of Whisker.

The SDK is based around a tool (ActiveX control) for use with VisualBasic, and compatible development environments. This tool greatly simplifies the Whisker command set, and also provides some extra functionality.

Details of setting up the control in Visual Basic can be found in the accompanying Tutorials.

9.1 Visual Basic and the SDK

Whisker comes with a **Software Development Kit (SDK)** to help you to write clients in Visual Basic and other languages.

9.1.1 Comments on the use of Visual Basic

VB is a very easy language to learn, and makes it very easy to interface to a Windows display (when you change a variable, the display on the screen is updated automatically). VisualBasic offers lots of short cuts (you don't have to declare variables as certain types, etc.) - which I advise you not to use: data type safety (making sure the values correspond to the right 'kind of thing') helps you avoid lots of errors.

VB has a lovely function called *DoEvents*, which enables the program to wait for a certain time, but to have other events being processed at the same time (using re-entrant functions). This completely removes the need for a great many timer events. There's an example in the example task; while one part of the program is waiting for the pellet dispenser, the same program can also respond to lever-presses. You do have to be a little careful with re-entrant functions (in particular, not to fiddle with global variables in certain ways, and not to allow too many re-entrant calls; these issues are discussed in the VB on-line manual).

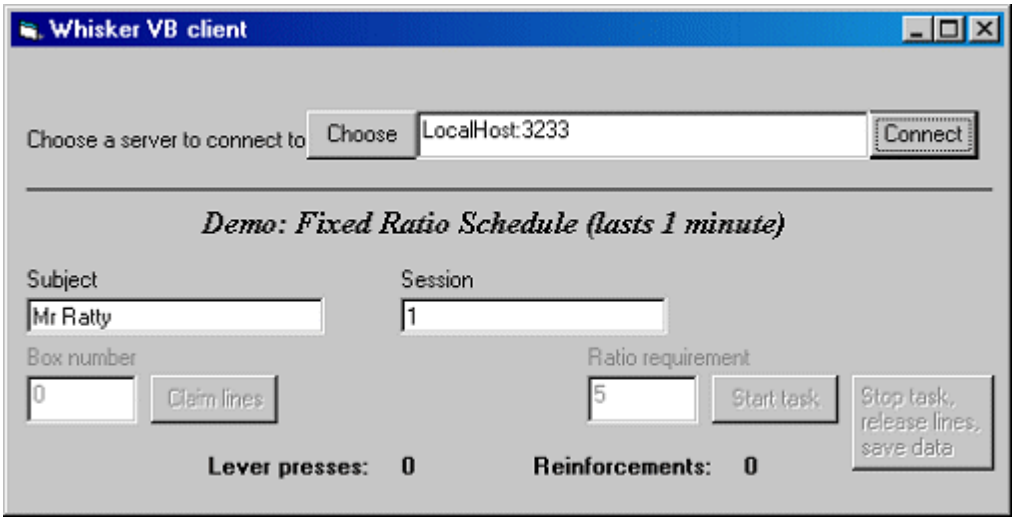
With the SDK Control, most of the potential problems with VB as a language are removed (the language has no native support for TCP/IP sockets, and the socket Control that is packaged with VB is hopeless). I hope that the SDK makes VB programming simple - it is the language that most users of WhiskerControl will probably use!

If you have any comments or suggestions on improving the SDK for VB programming, please contact the author.

9.1.2 VBRatioClient - a simple Visual Basic client

VBRatioClient is an extremely simple Visual Basic client that implements a ratio schedule. You can set the ratio requirement. It also implements a 1-minute timer; when this timer expires, the program ends and stores its data in a comma-delimited text file.

I hope it will be easy for you to develop into a task of your own, should you decide that Visual Basic is the language you want to use.



I suggest you make a copy of the whole project and explore the code – it's quite short. You'll probably find it useful to keep those parts of the code that setup the Server etc., but you'll want to change the functions that deal with the task itself (particularly *LineClaims*, *StartRatioSchedule*, *Whisker_Event*, *StopTask* and *TaskFinished*). The other functions are largely cosmetic.

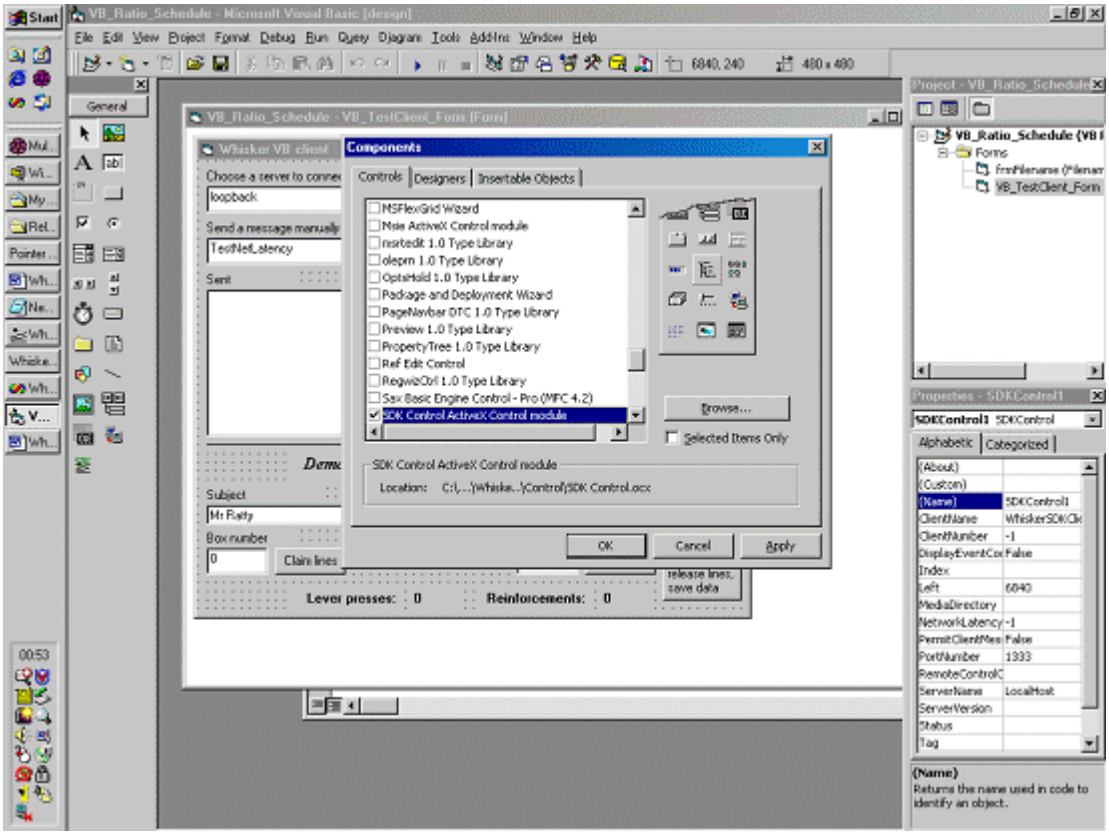
Note the use of Select Case …. End Select in Whisker_Event. This is a very useful way of choosing which piece of code to run for each Event. Those who are used to Arachnid might find it easiest to label the event with the name of a procedure, then call that procedure from within the Select Case block.

Technical note: source

This example used Visual Basic 6.0. It should work with any versions of Visual Basic above 4.0, although it has not been tested.

9.1.3 Writing a simple TestClient with the Visual Basic SDK

- Create a new project of type *Standard EXE*. Change the caption of Form1 to 'My TCP client' or something you prefer.
- Add an SDK control to your form. If you can't see the control on the toolbar, use *Project* → *Components*, tick *Whisker SDK ActiveX Control Module* and click *OK*. See illustration below:



- This will add the control to the toolbar so you can use it. Change the name of the SDK Control control (in the properties box) from 'SDKControl1' to 'Mytask' (or something).
- Add three TextBox controls to the form. Name them *txtSend*, *txtInfo* and *txtEvent*.
- Add a CommandButton control and name it *cmdConnect*. Change its caption to 'Connect'.
- Add another CommandButton called *cmdSend*, with the caption 'Send'.
- Choose *View* → *Code* to see the form's code. Add the following code to the form (just paste it in; you don't have to type the comments, which are lines beginning with an apostrophe):

```
Private Sub cmdConnect_Click()
     MyTask.ConnectToServer
End Sub

Private Sub cmdSend_Click()
     MyTask.SendToServer txtSend.Text
     ' When you click the "Send" command button, this sends
     ' the contents of the "txtSend" text box to WhiskerServer
End Sub

Private Sub Mytask_Info(ByVal InfoMessage As String, ByVal Time As Long)
     txtInfo.Text = InfoMessage
End Sub

Private Sub Mytask_Event(ByVal EventMessage As String, ByVal Time As Long)
     txtEvent.Text = EventMessage
End Sub
```

- You can change the IP name/address and port number of your WhiskerServer in the properties page of the SDK control. Use 'loopback' if the server is running on the same machine as the client. Port 3233 is the default. You can change these settings when the program is running (but

before the client is connected) by using the .ChangeSettings command. Try putting a command button on to do this (there's an example in the FR ratio project).

- **Save the project.**
- Start the server and run the project (click the  button). You should be able to connect, see messages coming from the server, and issue commands.
- When you want to compile your project to an **executable** (.EXE) file, choose *File → Make Project1.exe* (this may show a different name if you have renamed the project). Choose a filename and click OK.

Technical note: executable

The resultant executable file will run directly on any computer with the SDK installed. Note that whenever you use a control from a .OCX or .DLL file in an EXE, that control must be installed (registered) on the machine you wish to run the program on. See the VB documentation for details about this. The SDK Setup program will register the control for you.

- You might like to experiment with changing the txtOutput text box to a list box, so you can see several recent messages from the server. But that's just cosmetic.

9.2 The SDK command set

The control defines functions by means of a Type Library – this means that VisualBasic 'knows' the format of all the commands, and the constant values used by the Control. To browse this library from within VisualBasic, press F2 to show the Object Browser, when the control is loaded into the toolbox. This will show all of the commands in the library – the meaning of these commands should be fairly self-explanatory.

Most of these commands send the corresponding command to the WhiskerServer. Details of the Whisker Command set can be found in the Programmer's Guide which accompanies the WhiskerServer software. The next section explains the extra functions which are provided by the SDK control in addition to the functions provided by WhiskerServer.

For some commands, there are some slight differences between the structure of the Whisker and the SDK command. These are listed here.

9.2.1 Differences between SDK and Whisker Commands

The Whisker SDK Type library contains procedures that map onto the Whisker Command set. There are some slight differences, which are outlined below (syntax examples assume VisualBasic, with an SDK control called 'Whisker'). All Events, Methods, Properties and Constants, along with a description of their use, can be viewed in the Object Browser within Visual Basic.

- **Server Messages** will produce Events in the control's container (e.g. Visual Basic Form) .

  For example, in Visual Basic, if a SDK control (called Whisker) receives an Event: message, for example, it will call the Whisker_Event() subroutine of the form which contains it.

- **Constants** have been added to the type library to represent certain values which are given in

the Command set as strings. For example, wsLineOn and wsLineOff represent the two line states (On = 1 and Off = 0). *See the sections on LineReadState and LineSetState in the Programmer's reference, for important considerations using these values with VisualBasic.*

- Certain Whisker Commands (ReportName, ReportStatus, Version, ClientNumber, TestNetLatency, TimeStamps, SetMediaDirectory, DisplayEventCoords, and PermitClientMessages) have been implemented as properties. These can be read and, where appropriate, changed like any other object properties. View the SDKControlLib in the Object Browser for more details. Examples:

```
Whisker.Status = "I'm OK"
Whisker.ClientName = "Example Client"
myNumber = Whisker.ClientNumber
strServerVersion = Whisker.ServerVersion
netLatency = Whisker.TestNetLatency
```

- **Drawing commands** are treated slightly differently, so that default pen, brush and text settings can be configured by the user. See the topic Drawing with the SDK for more details.

- DisplayGetSize is replaced by:

```
DisplayGetSize(Device As String, Width As Long, Height As Long)
```

  If this call succeeds, the variables X and Y (which must be variables of type Long in VisualBasic) will contain the size of the display device named. For example:

```
Dim X As Long, Y As Long.
Whisker.DisplayGetSize "Screen", x, y
```

- LineSetState and LineReadState can be replaced by the shorthand

```
Whisker.LineState(LineName As String)
```

  For example, a line called "toplight" could be set using

```
Whisker.LineState(toplight) = wsLineOn
```

  Similarly, the current state could be read using

```
CurrentState = Whisker.LineState(toplight)
```

- AudioPlaySound does not have a -loop option in the SDK. Rather, a second command AudioLoopSound is provided.
- AudioLoadTone does not have a duration option in the SDK. Rather, a second command AudioLoadToneX is provided.

9.2.2 Drawing with the SDK

The SDK provides several functions to simplify drawing with Whisker.

Adding objects

The SDK provides serveral functions to allow simple addition of drawing objects to documents:

```
Whisker.DisplayAddText
Whisker.DisplayAddRectangle
Whisker.DisplayAddEllipse
Whisker.DisplayAddLine
```

as well as the plain vanilla

```
Whisker.DisplayAddObject
```

for the more esoteric drawing objects (see the Programmer's Guide for more details).

Drawing Styles

The server uses option strings to specify drawing styles to DisplayAddObject. The SDK simplifies the handling of drawing objects, with three functions:

```
Whisker.DisplaySetBrushOptions
Whisker.DisplaySetTextOptions
Whisker.DisplaySetPenOptions
```

These functions have a double use:

1. They set the 'default' drawing, pen and text options to the parameters specified.
2. They return a String value which represents the options which can be passed to the Server to specify these settings.

So to draw a line in a certain colour and width, all you need to do is to call

```
Whisker.DisplaySetPenOptions width, r, g, b, wsPenSolid
Whisker.DisplayAddLine docname, linename, x , y, x2, y2, ""
```

However, if you wanted to draw lines in two different styles, repeatedly, you could set up two different pens like this:

```
Dim Redln As String
Dim BlueLn As String
...
RedLn = Whisker.DisplaySetPenOptions(1, 255, 0, 0, wsPenSolid)
BlueLn = Whisker.DisplaySetBrushOptions(1, 0, 255, 0,wsPenSolid)
...
Whisker.DisplayAddLine docname, linename, x , y, x2, y2, RedLn
Whisker.DisplayAddLine docname, linename, a , b, a2, b2, BlueLn
```

9.2.3 Additional SDK commands

The SDK provides some additional commands that are not part of the Whisker Command set.

- Commands for killing events
- Commands for implementing simple schedules
- Commands for communicating with the user, and controlling the connection to the Server.

9.2.4 Cancelling events

The WhiskerServer sends an Event: message to the control whenever an event occurs of which the client has requested notification with one of the 'SetEvent' commands. Usually, this event is then simply passed on, along with the event's time stamp (the time, in ms, from the client's connection or the last call to ResetClock), to the _Event() subroutine of the control.

However, there are two occasions when such an event is not simply passed on:

1. When the client has instructed the SDK to **ignore** events of this name (using KillEvent).
2. When a **schedule** has been setup between this Event and another Event, and a cycle of this schedule is complete.

Consider the following extract from a VB program:

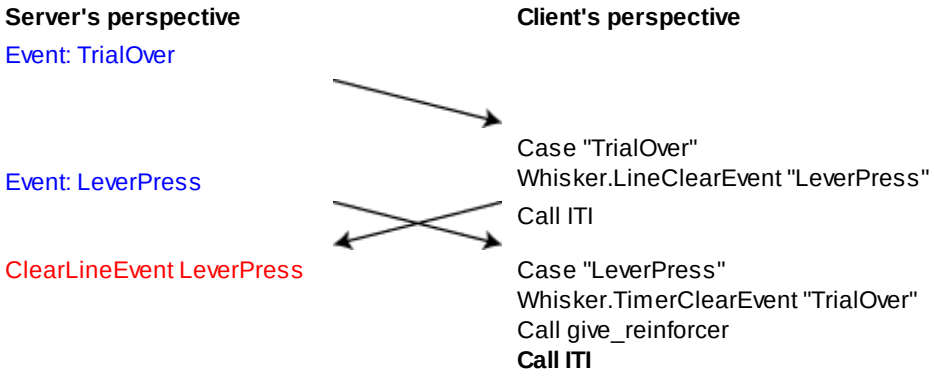
```
Private Sub Start()
     Whisker.TimerSetEvent "TrialOver", 1000, 0
     Whisker.LineSetEvent "Lever", "LeverPress", wsLineToOn
End Sub

Private Sub Whisker_Event(ByVal Message As String, ByVal Time As Long)
     Select Case Message
     Case "TrialOver"
            Whisker.LineClearEvent "LeverPress"
            Call ITI
     Case "LeverPress"
            Whisker.TimerClearEvent "TrialOver"
            Call give_reinforcer
            Call ITI
            End Select
End Sub
```

This code seems clear enough, I hope. Whatever the event that occurs first, the client cancels other outstanding events, and moves to the ITI (giving a reinforcer if the rat had pressed) . No problems, it seems.

But what if the TrialOver timer finishes just before the rat presses a lever?

The following sequence might happen:



Thus, because it takes some time for the client's message to reach the Server, there is always the possibility that two events that seem 'mutually exclusive' may actually both be generated – and the ITI routine gets called twice. This could have all sorts of nasty consequences – possibly trying to

run two trials at the same time!

This may be a rare event, as the messages pass so quickly, but that doesn't mean we can ignore the problem – and with devices that may generate lots of events quickly (like a touchscreen) it could happen frequently!

In general: ClearEvent commands cannot block events that are 'on the way' from the Server.

This may be inconvenient for some clients, especially for those people who are used to languages such as !Arachnid. Whisker SDK therefore adds an extra command to make this situation easy to avoid.

Instead of (or as well as) telling the Server to 'not send any more Events', we can instruct the SDK to ignore any future events of a certain name (Note: we cannot kill just 'line events' of this name – the event name is all the client will receive!).

The SDK command

```
Whisker.KillEvent <eventname>
```

tells the SDK to ignore any events (i.e. not to call the _Event subroutine) with the name *eventname*. This can be used to make sure that we do not respond to events in the kind of situation shown above. The effect of KillEvent can be reversed by

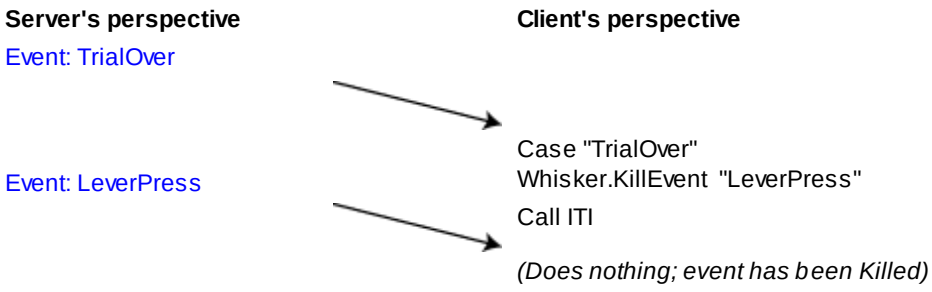
```
Whisker.ReviveEvent <eventname>
```

or

```
Whisker.ReviveEvent ""
```

to undo all previous `KillEvent` commands (and respond as normal to all events).

So if we replace '`ClearLineEvent`' with '`KillEvent`', we have solved the problem:



Note that, in this case, the server will continue to send LeverPress messages, until we Call `ClearEvent`. So we can just `ReviveEvent` on the next trial when we care about them again.

In summary:

`SetEvent` and `ClearEvent` are used to control whether the Server sends a message. You **cannot** be certain that an event is not 'on the way' when you clear it.

`KillEvent` and `ReviveEvent` are used to control whether or not the SDK calls your _Event () procedure when the event arrives. You can be certain that an Event will **not** be responded to after it has been Killed.

9.2.5 Implementing schedules of reinforcement

The Whisker SDK provides a simple tool for implementing commonly used schedules of reinforcement between triggering events (such as lever-press responses) and outcomes (such as reinforcement).

For simplicity, the events that trigger a schedule will be referred to as responses for the time being (even though they don't have to be; see *Higher-order schedules*, below.)

Fixed Ratio (FR) *N*: Every *N*th response causes an outcome to occur immediately.

Fixed Interval (FI) *T*: The first response to occur after an interval *T* from the last outcome causes the outcome to occur immediately.

Variable Ratio (VR) *N*: Each response has a probablility of (1/*N*) of causing an outcome to occur immediately. Thus, on average the outcome occurs every *N* responses. (Note: there are several ways of implementing VR schedules, of which this is one.)

Variable Interval (VI) *NT*: The first response to occur after a variable interval of mean length *NT* causes an outcome to occur immediately. (This is implemented as follows: during the interval, at the end of every sub-interval of duration *T*, there is a probablility of 1/*N* that the interval will end and the outcome will be made delivered when the next response occurs.)

Variable Time (VT) *NT:* The outcome will occur immediately after a variable interval of mean length *NT*. (This is implemented as follows: during the interval, at the end of every sub-interval of duration *T*, there is a probablility of 1/*N* that the interval will end and the outcome will be delivered immediately. There is no response contingency in this schedule; it is *noncontingent*.)

Note: there is no 'fixed time (FT)' schedule – this is provided by the TimerSetEvent function.

Implementing schedules

The SDK implements schedules by producing an Event message (i.e. calling the _Event subroutine) when the outcome is to be delivered. This Event message will appear exactly like a message from the server (and will include a time stamp in ms). The triggering response (for ratio or interval schedules) will **also** appear as an Event message.

*Note: Outcome event messages will be received by the _Event procedure immediately **before** the event that triggers them, and will contain identical time stamp information.*

Schedules are added with the AddSchedule command, and removed with the RemoveSchedule command.

Adding schedules

```
Whisker.AddSchedule <Type> <parameter> <triggerevent> <outcome>
```

The parameter will be *N* for FR,VR,VI and VT schedules. For FI schedules it is *T* (in ms). For VI and VT schedules the *T* parameter is set (in ms) via the SDK's **.ScheduleTimebase** property; for example:

```
Whisker.ScheduleTimebase = 1000
Whisker.AddSchedule wsVariableInterval, 30, "a", "b"
```

means that "a" will cause "b" on a VI-30-second schedule (with a 1000-ms granularity).

The VBRatio client, provided as a demonstration client, shows how a simple FR schedule can be implemented using the SDK.

Removing schedules

```
Whisker.RemoveSchedule <Type> <triggerevent> <outcome>
```

This command will remove any current schedule that matches the supplied criteria. A parameter **`wsAnySchedule`** is provided to match *any* schedule, and empty strings should be used to match any trigger event or outcome.

Thus

```
Whisker.RemoveSchedule wsAnySchedule, "a", ""
```

will stop any events scheduled to be caused by "a".

```
Whisker.RemoveSchedule wsFixedInterval, "", ""
```

will stop all FI schedules.

Removing schedules is not based upon messages to the server, and therefore has immediate effect, like `KillEvent()`.

Higher order schedules

'Outcome' or 'consequence' events resulting from simple schedules can in turn be scheduled by a second call to AddSchedule, providing a simple mechanism to generate higher-order schedules. For example,

```
Whisker.AddSchedule wsFixedInterval, 30000,  "LeverPress", "CS"
Whisker.AddSchedule wsVariableRatio, 4, "CS", "Reinforcer"
```

Following this, LeverPress events will cause CS events on a FI-30-s schedule, and CS events will cause reinforcer events on a VR-4 schedule. A second order design can thus be implemented by responding correctly to "CS" and "Reinforcer" events in the _Event() routine.

Note: If you schedule an event to cause itself (either directly or indirectly, through a higher order schedule) on a FR-1 or similar schedule, your client will "hang" as it constantly sends itself messages. You have been warned....

9.2.6 SDK extra commands

Communicating with the user

- **AlertOperator** Command

```
Sub AlertOperator(Message As String, LocalMsgBox As Boolean, AlertListeners
As Boolean)
```

Example

```
Whisker.AlertOperator "Finished Stage 1", False, True
```

The AlertOperator is used to display message boxes from the client to the user, without the client waiting for a response from the user. If 'LocalMsgBox' is True, then a messagebox is displayed by the client (on the machine running the program). If 'AlertListeners' is True, the SDK sends a 'broadcast' client message to all clients of the form "Alert: <message>". Both the WhiskerStatus and WebStatus clients will display a message box if they receive such a message.

In this way, a connected StatusClient, or web browser, can alert a user who is not working at the machine running the task that the task has finished.

- **AboutBox** Command

This displays an about box showing the version number of the SDK Control.

- **ChangeSettings** Command

This command displays a property page dialog, allowing the user to view or set properties of the control, such as the task name, status, etc. This dialog can be used to set the Server and PortNumber properties of a non-connected SDK control.

Connecting to the Server

- **ConnectToServer** Command

```
Function ConnectToServer() As Boolean
```

Example

```
If Not Whisker.ConnectToServer Then Whisker.AlertOperator("Could not
connect",True,False)
```

Connects to the Server (using the Server name and Port number given by the respective properties). Returns True if connection is made, False if not.

This method will connect both command (immediate) and event (main) sockets There is no need to worry about the Link: command, as this will be handled automatically.

- **Disconnect** Command

This command will close all connections with the Server. If this is called by a connected client,

a Disconnected Event will occur.

Development Note:
The Type library defines a property called 'RemoteControlCode'. This currently does nothing, and is included for future expansion.

9.3 Tutorials with Visual Basic 6.0

Tutorials include:

- Tutorial 1: Getting started writing Whisker tasks in Visual Basic
- Tutorial 2: Converting a simple !Arachnid task to a Whisker Client
- Tutorial 3: Programming tips

9.3.1 Tutorial 1 - Getting started writing Whisker tasks in Visual Basic

Source code for this project is supplied in the Tutorials folder (by default, within \Program Files\WhiskerControl).

Requirements

- A Windows machine with Visual Basic 6.0 and Whisker SDK installed.
- A Windows machine running WhiskerServer (any edition).

Typically, these will be the same machine, running the Programmer's edition of Whisker.

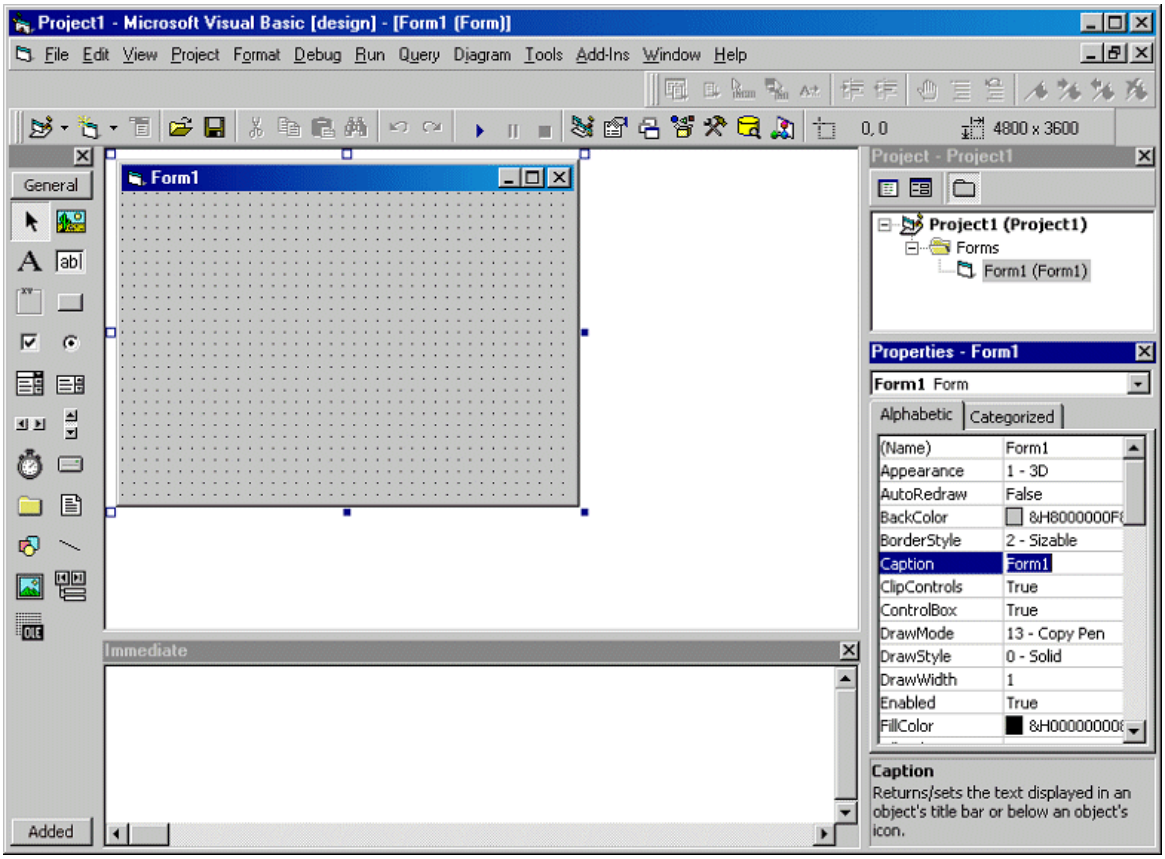
Setting up a new VB project

Run Visual Basic.

Select a 'Standard Exe' New Project.

You will see something like the screen below. This is the main window, the toolbox (left), the project explorer (top right) and the properties panel (bottom right), and immediate window (bottom). There may be some other windows showing, such as 'window position', which you can close.

*Note: These different windows can be closed, and reopened from the **View** menu. Save yourself time later, by getingt used to closing, opening and positioning these windows – otherwise it can be very confusing when one suddenly disappears!*



A new project in Visual Basic 6.0.

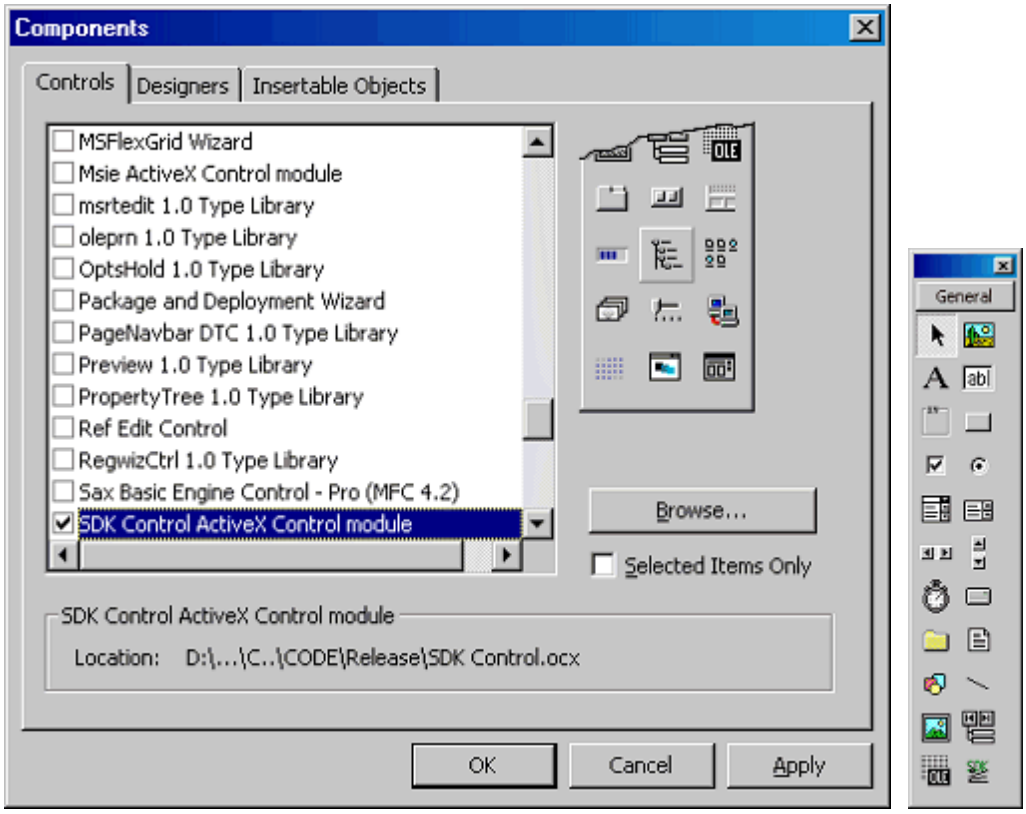
Visual Basic has given you a form (a simple kind of 'window' template) to play with. When the VB application is running, that form will be displayed to the user, and (most importantly) that form will receive messages ("EVENTS") from Windows.

Usually in VB these events will be things like mouse clicks – in our application, they are going to be the Whisker Events (licks, lever presses, etc) that we care about.

Making the form capable of receiving these events is simply a matter of placing an object on the form. This object is the Whisker SDK Control (A 'Control' is the name of the things that Visual Basic can put on forms). The panel on the left (with the General button) is the control toolbox – these are all the things we can add to the form. At the moment, our SDK Control isn't in there, so we have to tell VB to add it to our toolbox.

Adding the Whisker SDK control to the toolbox

Right click on the toolbox , and select *Components…* from the menu.



Components (left) and the Toolbox (right)

This will bring up a dialog like the one shown Find the Whisker SDK Control ActiveX Control module, and select it as shown above (Note ActiveX is Microsoft's name for the kind of technology used to write controls.

[If you cant see anything like 'Whisker SDK Control' in the list of controls, then you can look for it manually by selecting browse. It exists as a file called "Whisker SDK.ocx" in the system directory (if you have installed the SDK). Click 'Browse' and type Whisker SDK.ocx in the dialog that appears.]

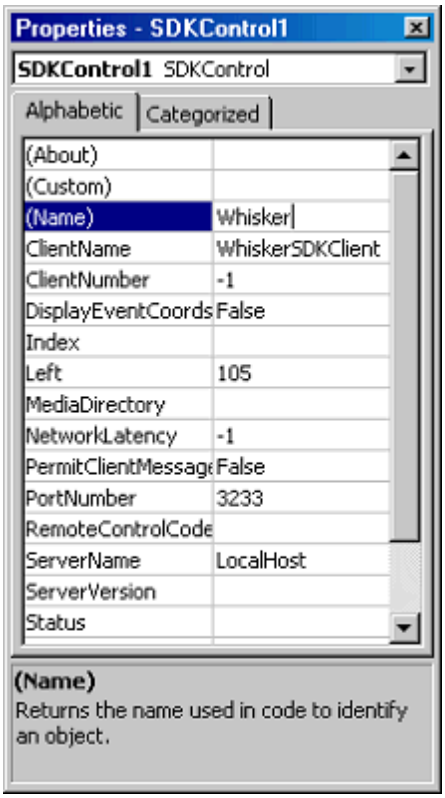
Click OK, and the Control should now be shown in the toolbox.

Adding the Whisker SDK control to the form

Now, if you select the Whisker SDK control in the toolbox (by clicking it), and then click-drag an area on the form, a control will appear on the form. The control is a fixed size (unlike most VB controls, so you can drag it to be any size, but it will always snap back). You can put the control where you like on the form, as it wont be visible on the screen when your task is running.

*Note: many aspects of Visual Basic programming require us to distinguish between how things behave whilst we are programming (**design time**) and whilst the application is running (**run time**). So to use the standard parlance, we would say that the Whisker SDK Control is "invisible at runtime". You will become more familiar with this terminology as you go on.*

Setting the Control's properties



Control properties.

When the Control is selected, the **properties panel** will show a set of properties associated with the control object. Some of these properties govern how the control will behave (in terms of which Server it tries to talk to, etc.), and some govern how the form (i.e. VB) will deal with the control.

If you click on a property, the pane at the bottom of the property panel gives a brief introduction to what each property does. At the top of the list are three special lines: (Name), (About) and (Custom).

The most important line to begin with is **(Name)**. This shows and sets the name that VB will use internally to refer to the control. By default this will be "WhiskerSDK1" which we can change to something else. Change it to "Whisker".

This name 'Whisker' can now be used within your program to access all of the Whisker functions from within the control.

The **(About)** line contains a button in the right hand field. Clicking it shows an 'about box' – which shows you the copyright notice, the version number of the control, and (of course) my name in lights!

The **(Custom)** line shows a **property page** for the control. Many of the other properties relating to your task (the name it will use to identify itself to the server, which server to connect to, where

to look for media files, etc) can be accessed from this property page.

Note: This page (unlike the main properties pane) can be viewed at run-time – that is, the user of your program can be shown this page (by means of a control function "ChangeSettings") in case they want to change these settings. This means that you don't have to write the same code in each of your programs to allow the user to choose the server, for example.

Take a moment to look at the properties. Visual Basic initialises them with sensible defaults – server name 'localhost' is a IP networking shorthand for 'this machine'. With the servername set to this, the application will try to talk to a server running on the same machine. That's (nearly always) what we'll want to do! Use both the properties panel, and the (Custom) button to play with the settings. Change the name of the task – call it something like "Harry's first whisker task" if you want….

Adding code to the form

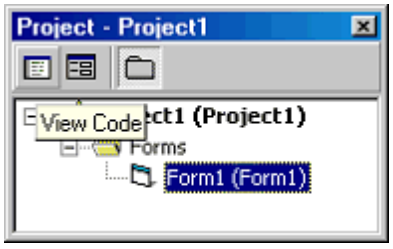
In VB, code can live in 3 main places.

1. In a form
2. In a code module
3. In a class module

For now, we're only going to scratch the surface of VB programming, and put all of our code in the form. The code in a form is used to manipulate the data represented by that form (remember – the form is a kind of window which can be presented to the user), and also to respond to events that windows sends to the form.

Note: Using code modules and class modules allows you to code in a much more concise & logical way than just plonking all your code in a form. As you get more used to VB programming, it is worth learning about 'object oriented' approaches. This will allow you to write programs quickly, and produce code that is much easier to understand and to debug!

If you double click on the form, or click the **view code button**, the main window will switch from the 'form designer' to the 'code editor'.



The 'View Code' button

The code editor is where you type in the program. The code for a VB application will **all be inside subroutines** (similar to PROCedures in BBC BASIC).

Why? Why doesn't it just start at the beginning?

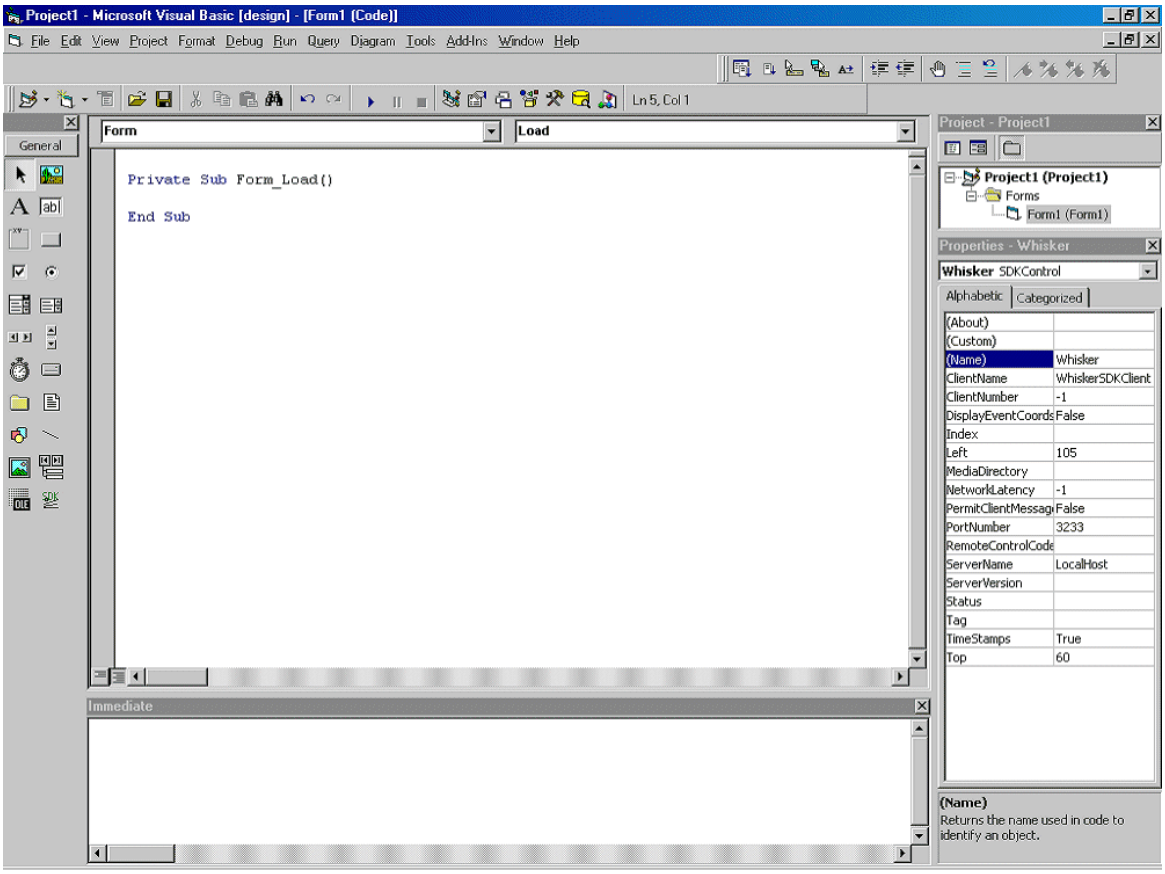
The idea of a program needing a 'first line' is something people who have learned BBC BASIC, or similar languages, are used to. But it doesn't make much difference to the logic where in the code the 'entry point' actually is – for example we could take a BBC BASIC program and put all of the stuff that isn't inside PROCedures, into procedure called 'start', and replace the first line with PROCstart. It wouldn't make any difference to how the program worked.

VB will call your code when it needs to – in response to an event, or at certain times, such as when it has started the program, or when the user has clicked the 'close' button on the form. It is possible to setup a VB project so that it starts with one procedure (sub main) as the 'beginning of the application', rather than loading a form. But you can learn how to do that later!

The Form_Load subroutine

If you double-clicked on the form to get to the code editor, you will now see the beginning and end of a subroutine marked by VB. This is the routine that VB will call as it starts the application, so we can regard it as the 'start' of our application.

If you clicked the View Code button to start the editor, you can insert the Form_Load subroutine by using the two pull-down menus at the top of the editor window. Select 'Form' in the left hand box, and 'Load' in the right hand box.



Form → Load

Note: The Form_Load subroutine is an Event-handling routine. That is, VB calls this routine when the 'being loaded' event happens to the form. [If our application just shows this form, this event happens as soon as the application starts]. All event routines have the same naming practise "objectname_eventname".

Calling Whisker functions

Let's say that we want to connect to the whisker server as soon as the form is loaded. We will need to tell VB to connect the control to the server – this is done by calling one of the control's methods "ConnectToServer".

Under the line

```
Private Sub Form_Load()
```

Type the following

```
Whisker.ConnectToServer
```

Did something funny happen when you were typing? You should have seen a box pop up as soon as you typed the period (full stop) after whisker. This is a handy feature of VB – it knows all of the methods (functions) and properties that can be accessed through the Whisker SDK control object – so you don't have to remember them all!

Now we can test the program. Run WhiskerServer (on the same machine), and once it has started, run the VB program (Run->Start; press F5; or click the blue arrow on the toolbar). You should see the form appear. On the Server Window's left hand panel, click on the plus next to the server, and then on the plus next to Clients. You should see your client's name.

Well done – you've written your first Whisker client.

Extending the client: The VB Status Client Deluxe

Although we have written a client, even in our flush of pride at our new creation, we will have to admit one thing. It's a bit rubbish. So – let's make it into something useful.

A useful client could be one that monitored what was happening on the server. There is a simple client that does just this - the Whisker StatusClient which comes with the server. The status client connects to the server, and asks for information about all connected clients, which it then displays. The display does not update unless the user clicks a button. For our first 'useful' client, we can make an enhanced version of a status client which asks Whisker for a Timer, so that it can refresh its information every couple of seconds.

All the code we need is below – paste or type it into the code window. You can get VB to insert the Whisker subs by using the two menus at the top of the code window.

```
Private Sub Form_Load()
   Whisker.ConnectToServer
```

```
   Whisker.TimerSetEvent "refresh", 2000, -1
End Sub
'called by VB when we start the application.
'It connects to the server, and then asks Whisker for a timer.
'The Timer will send the EVENT (a string 'refresh') after 2000 ms, and repeat
indefinitely.

Private Sub Whisker_Event(ByVal EventMessage As String, ByVal Time As Long)
   Cls
   Whisker.SendToServer "WhiskerStatus"
   Whisker.Status = "Connected for " + Str(Time / 1000) + " seconds"
End Sub
'Called by VB whenever an event occurs.
'Clears the form background, then sends a command message to the server
'Updates the client's reported status to reflect how long we've been connected

Private Sub Whisker_Info(ByVal InfoMessage As String, ByVal Time As Long)
   Print InfoMessage
End Sub
'Called by VB whenever the server sends information.
'Prints the information onto the form
```

Now run the program. If you copied the code right, you now have a truly useful program in 6 lines of hand-written code (count them). That's why VB is called a Rapid Application Development language!

Note: in response to receiving the command "WhiskerStatus" the server sends a set of information about all connected clients. See the programmer's guide for details.

Making it truly useful

Although this client works, it is not actually that useful. Why not? Well we wouldn't want to use a Whisker Timer to update the display, because we might well want this client to be running on a different machine to the Server. Generally, it's not a good idea to allow clients from other machines to control devices on the Server across the network (although you can set the server up that way if you want). Because a Timer counts as a device – a remote monitor client will not be able to do this.

We can make it more useful by asking Windows to time the interval for us using the Timer control instead of the Whisker Timer. This is a simple modification to the code above, and is left as an exercise. You will have to look at the online help for VB and find out about Timer controls [Don't worry about updating the client's status with the connection time in your new version – that is only included to show that the client is updating!]

If you can't work out how to do it, then the answer can be found, along with the source code, in the Tutorial 1 Files folder – but try to figure it out for yourself first!

When you've got the program working, you can turn it into an .EXE file which runs without VisualBasic (under *File→Make Exe*). There's your first completed Whisker Client!

A word of warning: any client that uses the SDK Control can only be run on machines with the SDK installed. However, your licence allows you to install the SDK on as many machines as

you like (although you can only install one copy of the real Editions of WhiskerServer per licence).

A note on oversimplifications

The client we knocked up above has some problems.

First, it uses the simple (but not that useful) Form.Print and Form.Cls functions. Don't use these for more complex tasks – as they are not very helpful!

Second, the Whisker_Event() procedure doesn't check the contents of the event message to find out what it was – this is no good at all for most clients, as they need to respond to many different events. See the tutorial example on converting an !Arachnid client to Whisker to see how this is done.

Pointers for future programming

Some of the remaining Tutorials wont make sense yet to entirely novice programmers. However, glance through to get a feel of where things are going. Then have a look at any of the VB tutorials available on the web, to get more confident with the basics, practice writing some VB code, and refer back there!

9.3.2 Tutorial 2 - Converting Arachnid tasks to Whisker

Source code for this project is supplied in the Tutorials folder (by default, within \Program Files\WhiskerControl).

Requirements

Computers:
- Acorn RiscOS machine.
- Windows PC, with Microsoft Word, Visual Basic and the Whisker SDK installed.

Know-how:
- A basic familiarity of opening and saving files in both Windows and RiscOS (opening files in Applications, etc.).
- Basics of VB, including using the SDK Control (see Tutorial 1).

Overview

A plan to show how one can go about transferring a BASIC file in !Arachnid to a Whisker task in VB, by example. The first two steps will be required for anyone wishing to convert their programs.

The subsequent steps show how to approach the problem for a specific task, which should be helpful to those wishing to convert their own software.

Step 1: Save the !Arachnid program to a DOS 1.44MB floppy disk in TEXT format.

Most disks are already in DOS format – any disk that the Windows PC reads will do.

Using the Acorn computer
If the file is a BASIC file, open it in !Edit (shift-double-click). Set the file type to TEXT (middle button to get the menu, misc→file type).
Save the file to the floppy disk (as e.g. 0:mytask).

Step 2: Open the text file on the Windows machine

Copy the file from the floppy to a sensible place on your computer, e.g. a folder called 'My Whisker Tasks'.

Run Microsoft Word. Select Open (Ctrl-O), and select the "mytask" file.

Microsoft Word should show a listing, with line breaks in the right place, etc.

Note: I recommend Word because it will deal with the Acorn line-break symbols. Other text editors might complain, or try to put the whole program on one line of text.

Save this file. You could print the file to look at while you work, to stop jumping between windows.

For this example, we will use a simple task "Licker" (used by Angela Roberts's lab in Cambridge). The code for this can be seen in full in the appendix of this document.

It is not without its flaws as a BASIC program; many people would object to the use of GOTOs; the structure is poorly commented, and some bits have been REMed out, and other bits seem to be a hang-over from a previous version.

This program was chosen for the tutorial partly because it was not perfect to begin with (and, to be honest, because it is short, and I was converting it anyway!). A tutorial which translated a 'perfect' BASIC program would not be that useful however - most of the programs that need converting will themselves have some idiosyncracies.

Step 3: Examine code structure

Assuming that the !Arachnid task has been written fairly well, you should be able to look at the code and divide it into bits: general structure will usually be a main section of the code - concerned with initialisation, getting settings, etc., then starting the task, PROCwait-ing, and then dealing with the data, some subroutines, and service functions. It's best to deal with each bit separately – some bits will translate very simply into VB, and others may require substantial reworking.

Note: I would **not** recommend 'cutting and pasting' !Arachnid code into VB, and then trying to remove the problems. The chances of obtaining a good Whisker Task at the end are almost zero!

What follows is my overview of the licker task, divided into bits.

Lines 10 – 40 Initialisation (freeing lines, zeroing variables)

```
   10 REM LICKER 2
   12 PROCinit:PROCkill_all:MODE0: BREAK=0: SESSION%=0
   15 ON ERROR IF ERR=17 RUN ELSE REPORT: PRINT" at line"; ERL:
PROCbreak: END
   30 @%=&20107
   35 PROCfree_switch(0,E%):PROCfree_switch(6,E%)
   40 FOR Z%=1 TO 5:PROCgovn_switch(Z%,E%):NEXT
```

Lines 42 – 49: Getting Session Settings

```
   42 PRINT TAB(2,2)"LICKING";
   43 PRINT TAB(2,4)"MONKEY NAME";:INPUT TAB(36,4),MONKEY$
   44 PRINT TAB(2,6)"SESSION LENGTH";:INPUT TAB(36,6),A%:A%=A%*6000
   45 IF A%=0 GOTO 44
   46 PRINT TAB(2,8)"FREE JUICE";:INPUT TAB(36,8),B$
   47 IF B$="Y" OR B$="N" GOTO 48 ELSE 46
   48 PRINT TAB(2,10)"FRONT (0) OR BACK (6)"::INPUT TAB(36,10),LOC%
   49 IF LOC%=6 PUMP%=4 ELSE PUMP%=2
```

Lines 60 – 96 Starting the program

```
   60 PROCpipe_timer(6,A%,0,"FNend(",0,E%)
   61 CLS:PRINT TAB(2,3)"NOS OF LICKS";
   62 PROCswitch_on(PUMP%,E%)
   65 PROCpipe_switch(LOC%,On,1,"FNfree(",0,E%)
   90 PROCwait(E%):*AE
   92 IF C$="R" RUN
   94 IF C$="M" CHAIN ":0.MENU"
   96 END
```

Service functions

```
  100 DEFFNfree(P%,R%)
  105  IF R%=0 =0
  110  IF B$="Y":PROCswitch_on(PUMP%,E%):X=X+1:PRINT TAB(15,3)X;
  115  IF B$="N" GOTO 300
  120 =0
  130 DEFFNend(P%,R%)
  140  IF R%=0 =0
  145  PROCkill_all
  150  PROCswitch_off(PUMP%,E%):PROCswitch_off(1,E%)
  160  CLS:PRINT TAB(2,1)"LICKING"
  170  PRINT TAB(2,3)"MONKEY";:PRINT TAB(30,3)MONKEY$
  180  PRINT TAB(2,5)"TOTAL NOS OF LICKS";:PRINT TAB(30,5)X: IF BREAK=1
GOTO 480
  190  PRINT TAB(2,18)"PRESS 'R' WHEN READY TO CONTINUE";:INPUT TAB
```

```
(36,18)C$
  200  IF C$<>"R" GOTO 190
  210  PRINT TAB(2,20)"RE-RUN 'R' RETURN TO MENU 'M' OR EXIT'E'";:INPUT
TAB(45,20)C$
  220  IF C$="M" GOTO 260
  230  IF C$="R" GOTO 260
  240  IF C$="E" GOTO 260
  250  GOTO 210
  260 =0
  300  PROCkill_switch(LOC%,E%)
  310  PROCswitch_off(PUMP%,E%)
  330  PROCpipe_switch(LOC%,On,1,"FNban(",0,E%)
  340 =0
  360 DEFFNban(P%,R%)
  370  IF R%=0 =0
  380  PROCswitch_on(PUMP%,E%):X=X+1:PRINT TAB(15,3)X;
  385  PROCpipe_timer(2,100,0,"FNbanoff(",0,E%)
  390 =0
  410 DEFFNbanoff(P%,R%)
  420  IF R%=0 =0
  430  PROCswitch_off(PUMP%,E%)
  440 =0
```

Error & Exception handling

```
  450  DEFPROCbreak
  460  BREAK=1
  470  GOTO 150
  480  SESSION%=FNtimer(6,E%)
  490  PRINT TAB(2,7)"SESSION LENGTH";:PRINT TAB(30,7)((A%-SESSION
%)/6000)
  500 ENDPROC
```

Step 4. Set up a VB project and initialisation code

This is a nice simple task, so it is suitable to put all the code into a single form. Set up a new project in VB, for a standard Exe. Set the form's caption to be something sensible ("Licker for Whisker"),and its Border size to be 'Fixed Single'. Put a SDK control on the form (call it Whisker). Open the code editor and put 'Option Explicit' at the top of the declarations.

Converting the initialisation:

```
   10 REM LICKER 2
   12 PROCinit:PROCkill_all:MODE0: BREAK=0: SESSION%=0
   15 ON ERROR IF ERR=17 RUN ELSE REPORT: PRINT" at line"; ERL:
PROCbreak: END
   30 @%=&20107
   35 PROCfree_switch(0,E%):PROCfree_switch(6,E%)
   40 FOR Z%=1 TO 5:PROCgovn_switch(Z%,E%):NEXT
```

Error handling is different in VB (but less necessary because of the compiler and debugger) – we'll come back to this in Step 8. Neither MODE nor @% are needed. Line numbers are similarly unnecessary.

VB & Whisker handle things slightly differently to !Arachnid. Initialisation now means connecting to the server: So PROCinit: PROCkill_all becomes Whisker.ConnectToServer.

The server can also use names, as well as numbers, to refer to lines. (This is most useful, as it means that only the SERVER has to worry about any re-wiring. You don't have to rewrite all your clients!). So instead of PROCgovn_switch / PROCfree_switch we just use Whisker. ClaimLine, specifying input or output with the fourth parameter using one of the two constants, wsInput or wsOutput [Note: the server knows which lines are input, and which are output in its settings, and will send an error message if it doesn't match!].

Because Whisker uses names for devices, FrontPump, Houselight, etc. we should declare string constants that contain the names of the devices we are going to use. Put the following at the top of the program (declarations section).

```
Option Explicit

' LICKER 2 conversion to VB

'Whisker device names (defined by the server)
'Put these as constants, rather than typing them in every time.
'That way the compiler will notice if we misspell one of them.

'Group name
Const Box = "Box1"

'Input Devices
Const FrontLicker = "FrontLicker"
Const RearLicker = "RearLicker"

'Output Devices
Const FrontPump = "FrontPump"
Const RearPump = "RearPump"
Const Houselight = "Houselight"
```

Where do we want to put the Whisker initialisation? We're in VB, so we don't just have a first line to put it on. We could put it in Form_Load, so it goes as soon as we start, but that might be but it might make more sense to do it when the user is ready to go – i.e. give them a button to click to start. Add a button to the form, call it cmdGo, and set the caption to Go. Double click on the button to jump to it's _click handler function.

```
Private Sub cmdGo_Click()
'Initialise
    If Not (Whisker.ConnectToServer) Then
        Call Break("Could not connect to server!")
    End If
    'Claim all lines for box
    Call Whisker.ClaimGroup(Box, "")
    'Check output lines
    Call Whisker.LineClaim(Box, FrontPump,  FrontPump, wsOutput, wsResetOff)
    Call Whisker.LineClaim(Box, RearPump,  RearPump, wsOutput, wsResetOff)
    Call Whisker.LineClaim(Box, Houselight,  Houselight, wsOutput, wsResetOff)
    'Check input lines
    Call Whisker.LineClaim(Box, FrontLicker,  FrontLicker, wsInput,
wsResetInput)
    Call Whisker.LineClaim(Box, RearLicker,  RearLicker, wsInput, wsResetInput)
End Sub
```

The initialisation code in Whisker is a little longer, but seemingly clearer. The only differences are that we now check for whether we find the server (not needed in !Arachnid), the ClaimGroup function, and the addition of comments.

I've put in 'Call Break("Could not connect to server!") - later on we will make our version of the break subroutine able to report problems to the user.

ClaimGroup requests control of all lines that go to a single box. This is handy because it gives us control over devices (say a second tone generator that is added **after** we wrote the program, that could affect us – even if we don't need them – and thus stops other programs from controlling them by accident.

ClaimLine is, as we noted above, similar to PROCgovn / PROCfree. The names are repeated because I have chosen to use the same name as the server's names – however, we need not do this. We could use a different name in the second parameter, which might (for example) represent the line's use. In this way we can give two or more output lines the same name – and control them as if they were a single output line. The final parameter is a Reset parameter, which governs how the server behaves to output lines when the client ends: you may wish the houselight to remain on after the client has finished, for example. This value is ignored for input lines (for clarity, we can pass 'wsResetInputLine').

Comments are added because they are a Good Thing. Unlike in BBC Basic they don't make it into the final program, so they have no cost whatsoever. Use them.

There's one more thing that we could do while we consider initialisation. What if the user wants to change the server name, or the connection port for any reason, or the name that the client reports to whisker? It's unlikely they will, in this instance, but it is possible. I've added it here as a demonstration because it's easy to do.

The obvious solution is to use the Whisker.ChangeSettings method (see Tutorial 1). Put another command button on the form, with a caption "Whisker Settings…". Inside it's click_handler, put the command:

```
Call Whisker.ChangeSettings
```

Step 5: Getting information from the user

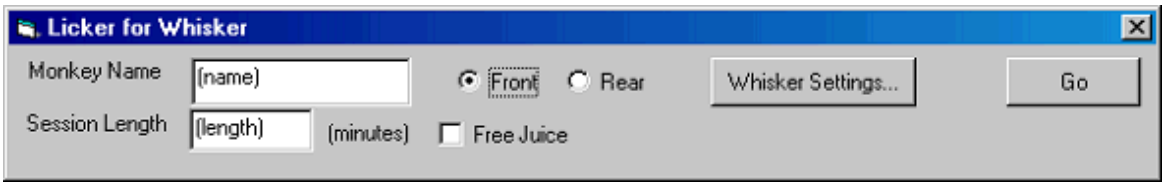
In !Arachnid:

```
42 PRINT TAB(2,2)"LICKING";
43 PRINT TAB(2,4)"MONKEY NAME";:INPUT TAB(36,4),MONKEY$
44 PRINT TAB(2,6)"SESSION LENGTH";:INPUT TAB(36,6),A%:A%=A%*6000
45 IF A%=0 GOTO 44
46 PRINT TAB(2,8)"FREE JUICE";:INPUT TAB(36,8),B$
47 IF B$="Y" OR B$="N" GOTO 48 ELSE 46
48 PRINT TAB(2,10)"FRONT (0) OR BACK (6)"::INPUT TAB(36,10),LOC%
49 IF LOC%=6 PUMP%=4 ELSE PUMP%=2
```

This code gets from the user a string, a number, a choice and a yes/no option. [The 'monkey name' is never actually used, but let's not worry about that.] All of this is done using simple keyboard input at the prompt. This is **not** the way VB works – VB makes pure Windows programs, and they don't have command line prompts in the same way.

So – what do we do? The answer is that we put controls (textboxes, checkboxes, and options) on our form, and let the user fill them in when they run the task. Open the form designer (click on View Object button), and add two options (for front and rear), a checkbox (for free juice Y/N), and two edit boxes (for monkey name & session length). Put labels by the edit boxes to confirm their use, and fill in the captions for the options and checkbox.

The layout I chose is shown below.



Give them all sensible names: txtName, txtLength, optFront, optRear, and chkJuice. In order to make sure that txtName and txtLength are labelled 'the right way round', it's a good idea to put the name of value in a each control's .Text property, so that it is shown inside the boxes while you are laying the controls out.

Note: I use the standard naming convention of a 3 letter prefix for controls showing their type. It is a good idea to use this system, and get used to, as most other VB programmers use it, so you will find their code is easy to read if you do it too!

Now – when the program starts we can read the options that the user has chosen from the controls, rather than from the command line. The options are a choice between 2, a yes/no, a string, and a number (which we'll want to convert to milliseconds). We need some variables to hold these settings, so we have to declare them.

Put the following into the declarations section at the top of the module (below the const declarations we put in earlier:

```
'Settings from form
Private bFreeJuice As Boolean
Private bUseFront As Boolean
Private strMonkey As String
Private lSessionLength As Long

Private strLicker As String
Private strPump As String
```

What this code does is 'create' the variables that will hold our settings: Three are **String**s (familiar), one is a **Long** (this is just a kind of integer [such as I% in BBC BASIC]), and two are **Boolean** variables. Booleans are variables that can only hold 2 values (True or False). This may sound like a very limited variable, but actually they are very useful! We'll use the Licker and Pump variables to hold the name of whichever set (front or rear) that we are actually using.

This syntax, and the whole notion of declaring variables, may be unfamiliar to you if you are used to BBC BASIC. Leave it for now, and I'll explain all after we finish adding code…

Put the following into the cmdGo_click subroutine (above the initialisation code we wrote earlier):

```
Private Sub cmdGo_Click()
'Get Settings
    bFreeJuice = chkJuice.Value
    bUseFront = optFront.Value
    strMonkey = txtMonkey.Text
    lSessionLength = 1000 * (Val(txtLength.Text) / 60)
'check settings
    If sSessionLength <= 0 Or strMonkey = "" Then
        Call Error("Invalid Settings")
    End If

    If bUseFront Then
        Pump = FrontPump
        Licker = FrontLicker
    Else
        Pump = RearPump
        Licker = RearLicker
    End If

'Initialise
...
```

This code reads the values that the user has entered from the controls and puts them into our variables. The SessionLength variable is converted from minutes to milliseconds in the process.

A brief check on the settings is performed, then we decide which pump and licker to use (compare with LOC% and PUMP% in the !Arachnid code).

*Note: It seems a lot more long winded to program using windows rather than text inputs (and messing around with user interfaces is **always** tedious), but a good interface makes a program much easier & quicker to use.*

*Just think: you only write a program once, but you run it many times (hopefully) so – even though it feels like a drag to add a easy to use interface, **you do generally save time in the long run**.*

An aside on declaring variables and scope

In BBC BASIC, you only generally have to declare (& reserve space for) array variables (using the DIM command), and you cannot use arrays you have not declared in that way. Ordinary variables, on the other hand, you do not usually declare, you just use them when you want.

This is a disaster (waiting to happen). I am sure that anyone who has programmed in !Arachnid has had a bug caused by a misspelling of a variable: the trouble is that these bugs often go unnoticed in BBC BASIC. Setting a 'non-existent' variable to 3 is not an error – BASIC just makes a new variable (with the misspelled name).

VB can operate in exactly the same way, but we type 'Option Explicit' at the top of a module to stop it doing so. If you are writing a very quick program (like in Tutorial 1) then you might not bother with Option Explicit. But don't leave it out for anything more complicated than a few lines, unless you like code that doesn't work, and is difficult to debug!

"Scope" of variables

Where you declare variables is crucial. In BBC BASIC you only ever declare normal variables (using the LOCAL keyword) inside subroutines – these variables then only "exist" inside the PROC or FN in which they are declared. Elsewhere in the program you cannot look at, or change, the value of these variables – this 'locale' in which the variable works is known as its scope.

Variables in VB are local by default – when declared with 'Dim' they only have meaning within the subroutine where they are declared and used. But of course, in VB, *all the code is within one subroutine or another*. So how do you declare variables that can be used from any subroutine? The answer, of course, is in the Declarations section – at the top of the module before the first Sub is declared.

Module variables can be declared by three keywords: Dim, Public or Private. **Always use Private** (at least, until you know when not to!).

Why? Later on, you'll start writing programs that use more than one module – Form modules, Basic modules & Class modules. If you declare your variables as Public in one module ***and then you decide to use a variable of the same name in another module****, you can cause bugs by accidentally changing the variable from 'outside'. If you make your variables Private, you never have to worry about using the same variable names in different places. This saves a lot of pain when writing big programs, and means that you can re-use your code without it not working.*

Very occasionally you might want to use a variable that can be seen by all modules. Only these variables should be declared public.

Step 6: Setting and responding to events.

The task starts in !Arachnid as follows:

```
50 X=0:PROCswitch_on(1,E%)
60 PROCpipe_timer(6,A%,0,"FNend(",0,E%)
61 CLS:PRINT TAB(2,3)"NOS OF LICKS";
62 PROCswitch_on(PUMP%,E%)
65 PROCpipe_switch(LOC%,On,1,"FNfree(",0,E%)
90 PROCwait(E%):*AE
92 IF C$="R" RUN
94 IF C$="M" CHAIN ":0.MENU"
96 END
```

This zeros the X variable, turns on line 1 (houselight) and pipes a timer to a service function, formats a display for the number of licks, and turns on whichever pump was chosen. It then pipes a 'lick' response to a service function and enters the wait state.

The conversion to Whisker in VB is trivial. Add the following under the initialisation code (at the bottom of cmdGo_click):

```
'Start
    iLicks = 0
    Call Whisker.LineSetState(Houselight, wsOn)
    Call Whisker.TimerSetEvent(EndSession, sSessionLength * 1000, 0)
    Call Whisker.LineSetState(Pump, wsOn)
    Call Whisker.LineSetEvent(Licker, Free, wsLineToOn)
```

The constants wsOn and wsLineToOn should be fairly self explanatory, they're use being similar

to On in !Arachnid.

Looking at the rest of the code I saw that X is used to count licks. I had to read the whole code, to work out what X is for – it might have been much easier if it had a more informative name. So I have replaced the name X with a more sensible name, iLicks (I because it is an integer).

I have asked for events called EndSession and Free. This will turn up when the timer finishes, or the licker is detected. I could have used literal strings, e.g. "EndSession", but it is better to use constants, and avoid the chance of bugs because of misspellings.

We want to use these variables throughout the module, so we need to declare them at the top. Add the following line to the declarations section:

```
Private iLicks As Integer

'EventNames
Const Free = "Free"
Const EndSession = "EndSession"
```

This is not quite the end. Whisker does not call service functions in the same way that !Arachnid does. Rather, it sends Event messages, which tell you what has happened. When one of these messages arrives, VB will call the Whisker_Event subroutine.

In order to convert !Arachnid "Pipes" into Whisker, the simplest thing to do is just to check which message we got, and call the corresponding subroutine. Visual Basic has just the command for this: the Select Case statement.

Insert the following in the Whisker_Event subroutine:

```
Private Sub Whisker_Event(ByVal EventMessage As String, ByVal Time As Long)
    Select Case EventMessage
    Case Free: Call Free
    Case EndSession: Call EndSession
    End Select
End Sub
```

When you come to write Whisker Clients from scratch you can continue to do this – add a Case: line for each pipe, and call a procedure. The Whisker system gives you a bit more flexibility than !Arachnid, however, and you can do 'what you like' with your event messages.

There is no equivalent to entering the 'wait' state in Whisker for Visual Basic. VB is an event-driven language and is therefore **always** in the equivalent of the wait state. But there is one more thing we have to do – we don't want the user to be able to start the whole session off again, or change the settings on the form, now we have started.

Add the following at the bottom of cmdGo_click:

```
'Disable the forms buttons
    Me.optFront.Enabled = False
    Me.optRear.Enabled = False
    Me.chkJuice.Enabled = False
    Me.txtLength.Enabled = False
    Me.txtName.Enabled = False
```

```
    Me.cmdGo.Enabled = False
```

We don't really need to disable anything apart from cmdGo – but by disabling the others, it is a bit more clear to the user that he/she cannot change them while it is running.

*Note: why is there a "Me." in front of all the control names? Me is a special object in VB, which can be used to refer to the form from within the form's code. It isn't **needed** here at all – but typing it means that VB will pop up one of the menus, from which you can select the control's name. This saves you typing them in…*

Step 7: Service functions

These should be fairly easy to transfer to whisker. We can continue to call them by their ! Arachnid names (i.e. beginning with FN), but they can be subroutines now.

```
  100 DEFFNfree(P%,R%)
  105  IF R%=0 =0
  110  IF B$="Y":PROCswitch_on(PUMP%,E%):X=X+1:PRINT TAB(15,3)X;
  115  IF B$="N" GOTO 300
  120 =0
  130 DEFFNend(P%,R%)
  140  IF R%=0 =0
  145  PROCkill_all
  150  PROCswitch_off(PUMP%,E%):PROCswitch_off(1,E%)
  160  CLS:PRINT TAB(2,1)"LICKING"
  170  PRINT TAB(2,3)"MONKEY";:PRINT TAB(30,3)MONKEY$
  180  PRINT TAB(2,5)"TOTAL NOS OF LICKS";:PRINT TAB(30,5)X: IF BREAK=1
GOTO 480
  190  PRINT TAB(2,18)"PRESS 'R' WHEN READY TO CONTINUE";:INPUT TAB
(36,18)C$
  200  IF C$<>"R" GOTO 190
  210  PRINT TAB(2,20)"RE-RUN 'R' RETURN TO MENU 'M' OR EXIT'E'";:INPUT
TAB(45,20)C$
  220  IF C$="M" GOTO 260
  230  IF C$="R" GOTO 260
  240  IF C$="E" GOTO 260
  250  GOTO 210
  260 =0
  300  PROCkill_switch(LOC%,E%)
  310  PROCswitch_off(PUMP%,E%)
  330  PROCpipe_switch(LOC%,On,1,"FNban(",0,E%)
  340 =0
  360 DEFFNban(P%,R%)
  370  IF R%=0 =0
  380  PROCswitch_on(PUMP%,E%):X=X+1:PRINT TAB(15,3)X;
  385  PROCpipe_timer(2,100,0,"FNbanoff(",0,E%)
  390 =0
  410 DEFFNbanoff(P%,R%)
  420  IF R%=0 =0
  430  PROCswitch_off(PUMP%,E%)
  440 =0
```

The FNban and FNbanoff functions are trivial. FNend function stops the program, and asks the

user if they wish to re-run the session. FNfree is a bit odd. Careful investigation of the code reveals that lines 300-340 are really part of FNfree – called by the GOTO on line 115.

This is one of those times when renumbering would be good – and putting lines 300-340 in after line 115. Even better – don't use GOTO at all! (VisualBasic would not let us do this kind of thing – jumping 'out' of one function's code, before returning. And that's just as well. It's EVIL. It works, but is very difficult to follow & understand).

So we can translate the code. Put the following into the code editor, at the bottom of the code: You can ignore the comments – they just illustrate which lines are replaced.

```
Private Sub FNfree()
    If bFreeJuice Then                                  'IF B$ = 'Y' THEN
        Call Whisker.LineSetState(Pump, wsOn)           'PROCswitch_on
        iLicks = iLicks + 1                             'X=X+1
    Else                                                'IF B$ = 'N' THEN GOTO...
        Call Whisker.LineClearEventsByLine(Licker, wsLineToOn) 'PROCkill_switch
        Call Whisker.LineSetState(Pump, wsOff)                 'PROCswitch_off
        Call Whisker.LineSetEvent(Licker, ban, wsLineToOn)     'PROCpipe_switch
    End If
End Sub

Private Sub FNend()
    Call Whisker.LineSetState(Pump, wsOff)          'PROCswitch_off
    Call Whisker.LineSetState(Houselight, wsOff)    'PROCswitch_off
    Call Whisker.Disconnect                         'PROCkill_all
End Sub

Private Sub FNban()
    Call Whisker.LineSetState(Pump, wsOn)             'PROCswitch_on
    iLicks = iLicks + 1                                 'X=X+1
    Call Whisker.TimerSetEvent(banoff, 1000, 0)     'PROCpipe_timer
End Sub

Private Sub FNbanoff()
    Call Whisker.LineSetState(Pump, wsOff) 'PROCswitch_off
End Sub
```

I've ignored the printing of the data on screen when iLicks (X) is incremented, and the communication with the user at the end of FNend. This is only for clarity – we'll come back onto these points in the next step. In the meantime, we can note in passing how similar the code is: the translation of the 'innards' of an !Arachnid task to a Whisker Client is very simple indeed (Note that you have to convert timer durations from cs to ms however!).

I have chosen the direct equivalent of 'PROCkill_switch' to translate that function, by using ClearEventsByLine(). This clears the events set for a particular transition on a particular line. That could be replaced by LineClearEvent(free) – which would stop any line events from seding the event 'free', or KillEvent(free) – which would mean that the event 'free' would be ignored (Whisker_Event would not be called). This flexibility can be useful in complex designs in which several alternative actions could produce the same outcome. However – I have used the nearest equivalent command to the !Arachnid line code.

The observant reader will already know what is missing before we have finished, however – there are two 'pipes' in the code, which we have replaced with 'SetEvent' instructions. So we need to

1. declare the constant strings for our event names in the declarations section:

```
Const banoff = "BanOff"
Const ban = "Ban"
```

2. add Case: lines to the code in Whisker_Event, giving:

```
Private Sub Whisker_Event(ByVal EventMessage As String, ByVal Time As Long)
    Select Case EventMessage
    Case Free: Call FNFree
    Case EndSession: Call FNEndSession
    Case ban: Call FNban
    Case banoff: Call FNbanoff
    End Select
End Sub
```

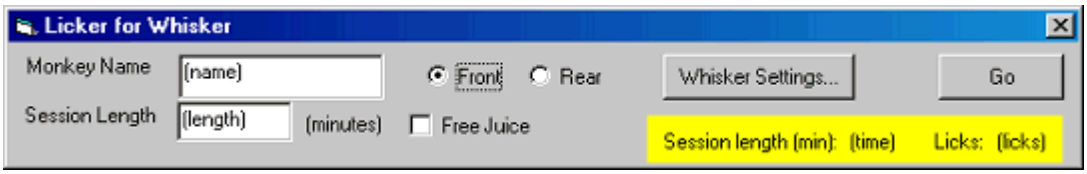
Now we're done – apart from 3 things: Showing the results, PROCbreak, and allowing a re-run. All of these are basically involve exchanging information with the user. As we saw in Step 5 – that is done differently in Visual Basic.

So that's it as far as the original code is concerned – the final step is just adding Visual Basic – style finishing touches to our translated client.

Step 8: Communicating with the user

Displaying session info

We have a form window on screen – we can easily write data to this by means of Label controls. I decided to put the data output onto a highlighted bit of the window as shown below:



Open the form designer, and put a Frame on the window where the output will appear – call it "fraOutput". Set its Border property to none, clear its caption, and set its background colour to something bright (say, yellow).

Put **four** labels in this frame. Set the background colours to be the same as that of the frame. Set the captions to be "Session length(min):", "(time)", "Licks:", and "(licks)". Set the Names of the (time) and (licks) labels to be lblTime and lblLicks. We will use these to show the data.

After each iLicks = iLicks + 1 line, add the following (again, the Me isn't needed):

```
Me.lblLicks.Caption = Str(iLicks)
```

This will then display an uptodate licks count to the user.

How about the session length? The !Arachnid version didn't display that until the end, but we are

actually receiving timing information from the server every time it contacts us – the Time parameter of Whisker_Event contains the time in ms since we connected, or reset our clock on the server. I suggest we use this to display the time to the user.

So just add this to the bottom of the Whisker_Event subroutine:

```
Me.lblTime.Caption = Str(Int(Time / 60000))
```

This is an example of the extra flexibility of the Event system over the !Arachnid Pipe system – what we are doing in this example is trivial, but we could be doing some more interesting post-processing on some or all of our events in this way.

Re-runnning the session

Rather than use the command line (and all those ugly GOTOs), we can allow the user to re-run the program once it has finished, simply by re-Enabling the cmdGo button.

Note that the FNend routine will disconnect us from the server – all we need to do is let the user re-enter the data, and click cmdGo to reconnect, and off we go again.

Add the following to the bottom of the FNend routine:

```
'Re-Enable the forms buttons
    Me.optFront.Enabled = True
    Me.optRear.Enabled = True
    Me.chkJuice.Enabled = True
    Me.txtLength.Enabled = True
    Me.txtName.Enabled = True
    Me.cmdGo.Enabled = True
```

Note: we need to make sure our variables are not affected by this – in more complex designs it might not be so simple! But here, we are okay - note that iLicks will be zeroed when we restart.

Handling errors

Full error-handling is tricky in VB, but not really needed here (refer to the online help & any VB tutorials you have to find out more). But there are only 3 kinds of 'exceptional' situation we need to take notice of in this task:

Problems starting – all we need to do is tell the user what the problem is and wait.

Server Problems are signified by Whisker_Error messages. They mean that the server can't do what we asked – which is a Bad Thing. So we should report these to the user and take the appropriate action (disconnect, in this fairly simple case).

User wants to Quit (this is the only error handling that happened in the !Arachnid version of "Licker" => escape jumped to PROCbreak)

Telling a windows program to Quit is usually done by clicking the close button – and VB will then ask the form if it wants to quit by calling the QueryUnload subroutine. It is a handy skill to

ask the user if they **really** want to quit from that subroutine, and pass the answer back to VB. I'll give an example here.

Add the following code to the module to complete the conversion:

```
Private Sub break(ByVal message As String)
Whisker.AlertOperator (message)                 'PASS THE MESSAGE ON
End Sub

Private Sub Whisker_Error(ByVal ErrorMessage As String, ByVal Time As Long)
Whisker.Disconnect
Whisker.AlertOperator (ErrorMessage)          'PASS THE MESSAGE ON
End Sub

Private Sub Whisker_Disconnected()
Whisker.AlertOperator("Session over - Disconnected", True, False)
End Sub

Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
'This will be called when VB wants to unload. See QueryUnload in vb help
'If it is because of a User clicking 'X', UnloadMode will be vbFormControlMenu
'(It could be for some other reason, e.g. because the computer is shutting
down)
If UnloadMode = vbFormControlMenu Then
  Cancel = (MsgBox("Are you sure you want to Quit?", vbOKCancel, "Confirm
Quit") = vbCancel)
End If
End Sub
```

This code shows how to use a VB's MsgBox function to ask a question of the user. Note that the MsgBox function is Modal – that is, it stops the subroutine while the messagebox is displayed (it has to, in order to return the value saying what the user chose). This is not as bad as it sounds:

Modal messageboxes will stop Whisker messages if you are running from VisualBasic (e.g. by pressing F5 VB environment). So, if you ran this program by pressing F5, then tried to quit, our client would ignore all events (licks, end of session, etc) while the message box was being shown. This makes it seem like using messageboxes is a complete disaster!

However, if you compile this program into an Exe file (using File->Make exe), then try the same thing, you'll see that Whisker responds perfectly well to events while the message box is displayed (Phew!).

What if you want to display a message to the user but **not** stop the flow of the program at all (i.e. you don't care about the answer)? For this purpose, Whisker supplies the "Alert Operator" method. This allows the client to display a messagebox on the monitor, without waiting for a reply. The second parameter of the AlertOperator function allows you to display a messagebox on any connected WhiskerStatusClients (this is done by means of a special clientmessage broadcast to all clients). This is provided to allow you to inform a remote machine (e.g. your workstation) that something has occurred without waiting for that machine to check.

So – that shows how to convert a simple digital io task from !Arachnid to Whisker.

***But**: please see the SDK documentation for the difference between KillEvent() and*

ClearEvent() in Whisker, when converting Arachnid programs.

A final quick question to check that you are still awake: why do I not try to let listeners know that I had finished (note the second parameter is 'False')? Would the following not have worked in Whisker_Disconnected?

```
Private Sub Whisker_Disconnected()
Whisker.AlertOperator("Session over - Disconnected", True, True)
End Sub
```

If not, why not? (answer is in Tutorial 2 files folder, along with the source code)

APPENDIX – Original Licker task code

```
   10 REM LICKER 2
   12 PROCinit:PROCkill_all:MODE0: BREAK=0: SESSION%=0
   15 ON ERROR IF ERR=17 RUN ELSE REPORT: PRINT" at line"; ERL:
PROCbreak: END
   30 @%=&20107
   35 PROCfree_switch(0,E%):PROCfree_switch(6,E%)
   40 FOR Z%=1 TO 5:PROCgovn_switch(Z%,E%):NEXT
   42 PRINT TAB(2,2)"LICKING";
   43 PRINT TAB(2,4)"MONKEY NAME";:INPUT TAB(36,4),MONKEY$
   44 PRINT TAB(2,6)"SESSION LENGTH";:INPUT TAB(36,6),A%:A%=A%*6000
   45 IF A%=0 GOTO 44
   46 PRINT TAB(2,8)"FREE JUICE";:INPUT TAB(36,8),B$
   47 IF B$="Y" OR B$="N" GOTO 48 ELSE 46
   48 PRINT TAB(2,10)"FRONT (0) OR BACK (6)"::INPUT TAB(36,10),LOC%
   49 IF LOC%=6 PUMP%=4 ELSE PUMP%=2
   50 X=0:PROCswitch_on(1,E%)
   60 PROCpipe_timer(6,A%,0,"FNend(",0,E%)
   61 CLS:PRINT TAB(2,3)"NOS OF LICKS";
   62 PROCswitch_on(PUMP%,E%)
   65 PROCpipe_switch(LOC%,On,1,"FNfree(",0,E%)
   90 PROCwait(E%):*AE
   92 IF C$="R" RUN
   94 IF C$="M" CHAIN ":0.MENU"
   96 END
  100 DEFFNfree(P%,R%)
  105  IF R%=0 =0
  110  IF B$="Y":PROCswitch_on(PUMP%,E%):X=X+1:PRINT TAB(15,3)X;
  115  IF B$="N" GOTO 300
  120 =0
  130 DEFFNend(P%,R%)
  140  IF R%=0 =0
  145  PROCkill_all
  150  PROCswitch_off(PUMP%,E%):PROCswitch_off(1,E%)
  160  CLS:PRINT TAB(2,1)"LICKING"
  170  PRINT TAB(2,3)"MONKEY";:PRINT TAB(30,3)MONKEY$
  180  PRINT TAB(2,5)"TOTAL NOS OF LICKS";:PRINT TAB(30,5)X: IF BREAK=1
GOTO 480
  190  PRINT TAB(2,18)"PRESS 'R' WHEN READY TO CONTINUE";:INPUT TAB
(36,18)C$
  200  IF C$<>"R" GOTO 190
```

```
  210  PRINT TAB(2,20)"RE-RUN 'R' RETURN TO MENU 'M' OR EXIT'E'";:INPUT
TAB(45,20)C$
  220  IF C$="M" GOTO 260
  230  IF C$="R" GOTO 260
  240  IF C$="E" GOTO 260
  250  GOTO 210
  260 =0
  300  PROCkill_switch(LOC%,E%)
  310  PROCswitch_off(PUMP%,E%)
  330  PROCpipe_switch(LOC%,On,1,"FNban(",0,E%)
  340 =0
  360 DEFFNban(P%,R%)
  370  IF R%=0 =0
  380  PROCswitch_on(PUMP%,E%):X=X+1:PRINT TAB(15,3)X;
  385  PROCpipe_timer(2,100,0,"FNbanoff(",0,E%)
  390 =0
  410 DEFFNbanoff(P%,R%)
  420  IF R%=0 =0
  430  PROCswitch_off(PUMP%,E%)
  440 =0
  450 DEFPROCbreak
  460  BREAK=1
  470  GOTO 150
  480  SESSION%=FNtimer(6,E%)
  490  PRINT TAB(2,7)"SESSION LENGTH";:PRINT TAB(30,7)((A%-SESSION
%)/6000)
  500 ENDPROC
```

9.3.3 Tutorial 3 - Programming tips

Source code for this project is supplied in the Tutorials folder (by default, within \Program Files\WhiskerControl).

Programming Tips

VB is a dirty language. It's quick and easy, but the flip side to this is that it is easy to write code that *nearly* does what you want. In fact, it is so easy to write bad code in VB, that many professional programmers will not touch it.

But – a little care can help you avoid some of the pitfalls.

Tips:

- Note the lines outside the subs in the code we saw above (Tutorial 2). These start with apostrophes, which shows that they are comments (they are ignored by Visual Basic, but they help the human reader understand what is going on). ALWAYS add comments to your code – one day soon, someone will try to work out what your program does (probably you) and it saves time if they can see what each bit is doing. It also makes it much easier to spot mistakes when you are debugging – if you have to look at the whole code to work out what a subroutine does, it is more difficult to see that it does the wrong thing! Code that is difficult to follow is almost worse than useless: most people learn this

through experience – either through looking back at their own code, or trying to follow someone else's code.

- **A warning for Arachnid Users:**

  Arachnid offers the 'KillSwitch' function – after this has been called, the pipe cannot be executed, and the service function will not be called. In Whisker, you can ask for an event to be Cleared [i.e. ask the server not to generate any more] **but** the event might (conceivably) be have been generated before you ask: i.e. occur 'in the gap' between your client processing the event, and your 'Clear event' response. In general, this will virtually never occur, and it is easy to write clients that only respond to events that they expect.

  However – people who wish to transfer tasks from Arachnid to Whisker should not rely on this, and there are occasions when it could even be likely. For example, if we set up a display document that had two adjacent objects, and put them on the screen, each with their own response: a touch on the screen at the boundary could produce two different events almost simultaneously. The following code might then fail:

```
Private Sub Whisker_Event(ByVal EventMessage As String, ByVal Time As
Long)
   Select Case EventMessage
          Case Box1: Whisker.DisplayBlank("Screen"): Call Correct
          Case Box2: Whisker.DisplayBlank("Screen"): Call Wrong
   End Select
End Sub
```

  Relying on the DisplayBlank to prevent another eventmessage arriving is a problem: we might end up with both Correct and Wrong being called. How can we prevent this? The Whisker SDK provides functions to "ignore" events, which works in a similar way to Arachnid:

```
Whisker.KillEvent(eventname)
```

And

```
Whisker.ReviveEvent(eventname)
```

  Once an event is Killed, then any message for that event will be ignored *from that point on* – note this is not the same as preventing these messages from being generated.

  ClearEvents prevents messages being generated from a particular source (doesn't garauntee that none are in the queue).
  KillEvent makes the library ignore any event with that name (no matter where it comes from) *from that moment* until ReviveEvent() is called.

  In order to undo a message, call ReviveEvent(). To undo **all** Killed messages, call ReviveEvent with an empty string (Whisker.ReviveEvent("")).

- Practice with the Debugger. It is useful and it is your friend!

- Always type **option explicit** at the top of each module or form. This tells VB that you

will declare (using a Dim, public or private statement) every variable before you use it. This means that you cant just invent a variable 'as you go'. This might sound like a rubbish idea **but** it means that you will never find that your program doesn't work because you've written TrailData = 3 instead of TrialData = 3. VB will notice the misspelling if you set option explicit, otherwise it will assume you want a new variable, and carry on happily

- The VB Environment can be annoying. Two ways of making it better can be done on the *Tools→Options* Dialog.
    - Select Environment, then check "Prompt to Save Changes" when a program starts. That way, you can save your program before you start it. This is handy - if your program crashes, you don't lose everything.
    - Select Editor, and **uncheck** "Auto Syntax Check". This feature just pops up message boxes if you ever leave a line unfinished in your code (e.g. while looking something up. The most annoying feature ever invented.
- Try to avoid reading BASIC 'as if it were English'. It isn't. The language has very precise meaning, and you should always read its **structure** if you can. To test yourself: what will the following code do? (Debug.Print prints to the immediate window).

```
Dim A As Integer
For A = 1 To 3
 If A = 1 Or 2 Then
    Debug.Print "Low"
 Else
    Debug.Print "High"
 End If
Next
```

Put the code into Form_Load and try it. Does it do what you expect? If not, why not? The answer is in the 'general tips files' folder.

- Look at other people's code. Try to learn from it. If it is easy to understand, try to see why. There are three main ways of learning to code: by example, by example and by example…
- Give your subroutines and variables sensible names. It is often a good idea to name your variables with names that show what type of data they refer to (e.g. iNumber for integers, strName for strings, txtATextBox for controls). Try to avoid the Variant data type (that is, declaring variables **without** giving them a data type) unless you have to. It's easier to debug code when you use data types.
- Understand variable scope, and don't use global variables unless you have to. Learn about static variables.
- If you write procedures that might be useful in lots of experiments, put them in a separate module. This can help you reuse your code.
- Learn what you can about object oriented programming. It is (honestly) worth it!

Good Luck…

9.4 Tutorials with Visual Basic .Net

These are updates of the tutorials. Compiled by MRFA using the VisualBasic 2005 (Express Edition)

- Tutorial 1: Getting started writing Whisker tasks in Visual Basic
- Tutorial 2: Converting a simple !Arachnid task to a Whisker Client
- Tutorial 3: Programming tips

9.4.1 Tutorial 1 - Getting started writing Whisker tasks in Visual Basic

Source code for this project is supplied in the Tutorials folder (by default, within \Program Files\WhiskerControl).

Requirements

- A Windows machine with Visual Basic .net and Whisker SDK installed.
- A Windows machine running WhiskerServer (any edition).

Typically, these will be the same machine, running the Programmer's edition of Whisker.

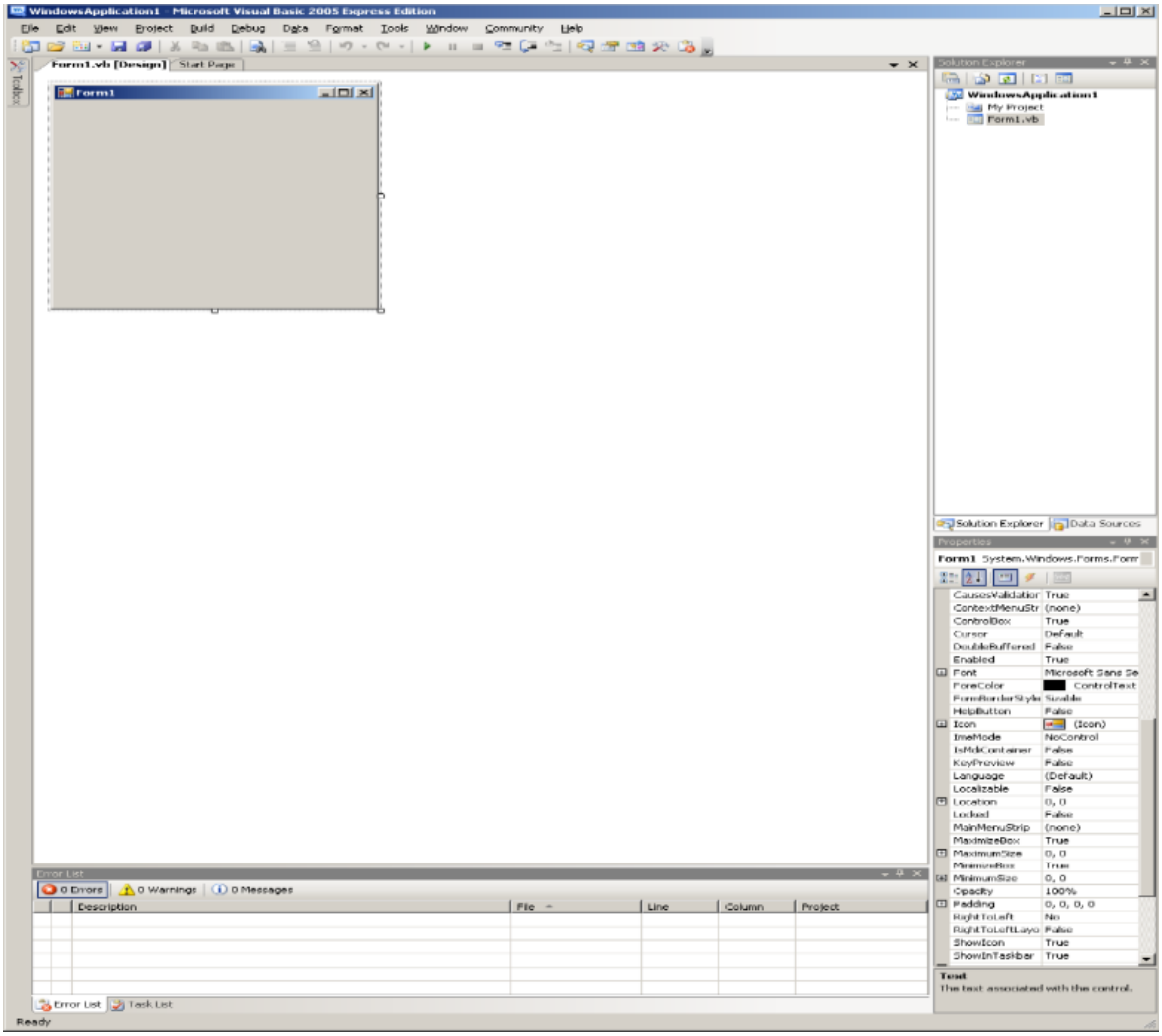
Setting up a new VB project

Run Visual Basic.

Create a new Project, of type "Windows Application".

You will see something like the screen below. This is the main window, the toolbox (left), the project explorer (top right) and the properties panel (bottom right), and errors window (bottom). There may be some other windows showing, which you can close.

*Note: These different windows can be closed, and reopened from the **View** menu. Save yourself time later, by getingt used to closing, opening and positioning these windows – otherwise it can be very confusing when one suddenly disappears!*



A new project in Visual Basic 2005

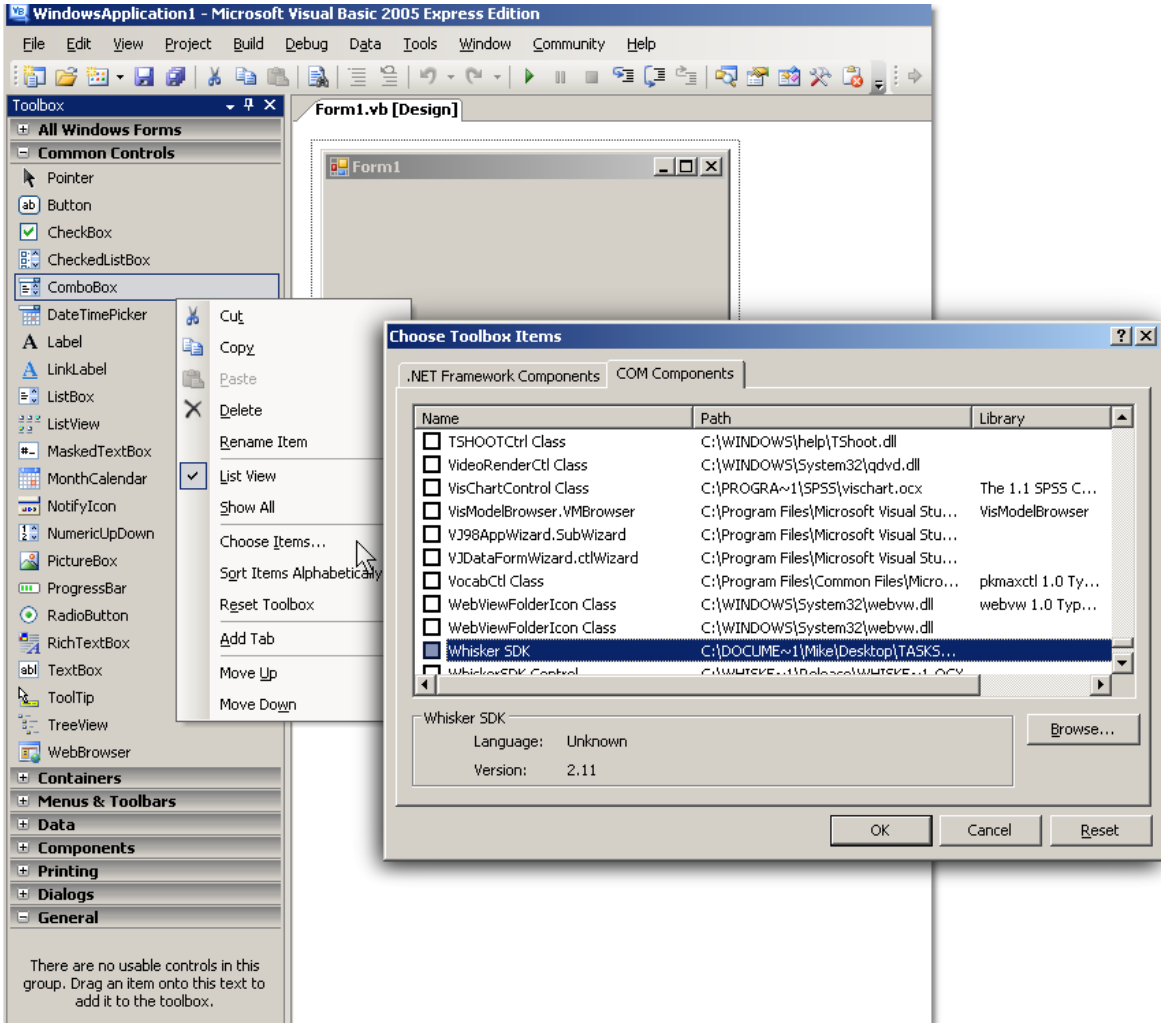
Visual Basic has given you a Windows Form (a simple kind of 'window' template) to play with. When the VB application is running, that form will be displayed to the user, and (most importantly) that form will receive messages ("EVENTS") from Windows.

Usually in VB these events will be things like mouse clicks – in our application, they are going to be the Whisker Events (licks, lever presses, etc) that we care about.

Making the form capable of receiving these events is simply a matter of placing an object on the form. This object is the Whisker SDK Control (A 'Control' is the name of the things that Visual Basic can put on forms). The control toolbox – is on the left. It contains all the things we can add to the form. At the moment, our SDK Control isn't in there, so we have to tell VB to add it to our toolbox.

Adding the Whisker SDK control to the toolbox

Open out the toolbox (by clicking on it), then select *Choose Items…* from the menu.



The Choose Toolbox Items menu

This will bring up a dialog like the one shown. The Whisker SDK is a COM item, so select the COM Components tab, as shown above, and find Whisker SDK, and check the box to the left of the name.

[If you cant see anything like 'Whisker SDK' in the list of COM components, then you can look for it manually by selecting browse. It exists as a file called "Whisker SDK.ocx" in the system directory (if you have installed the SDK). Click 'Browse' and type Whisker SDK.ocx in the dialog that appears.]

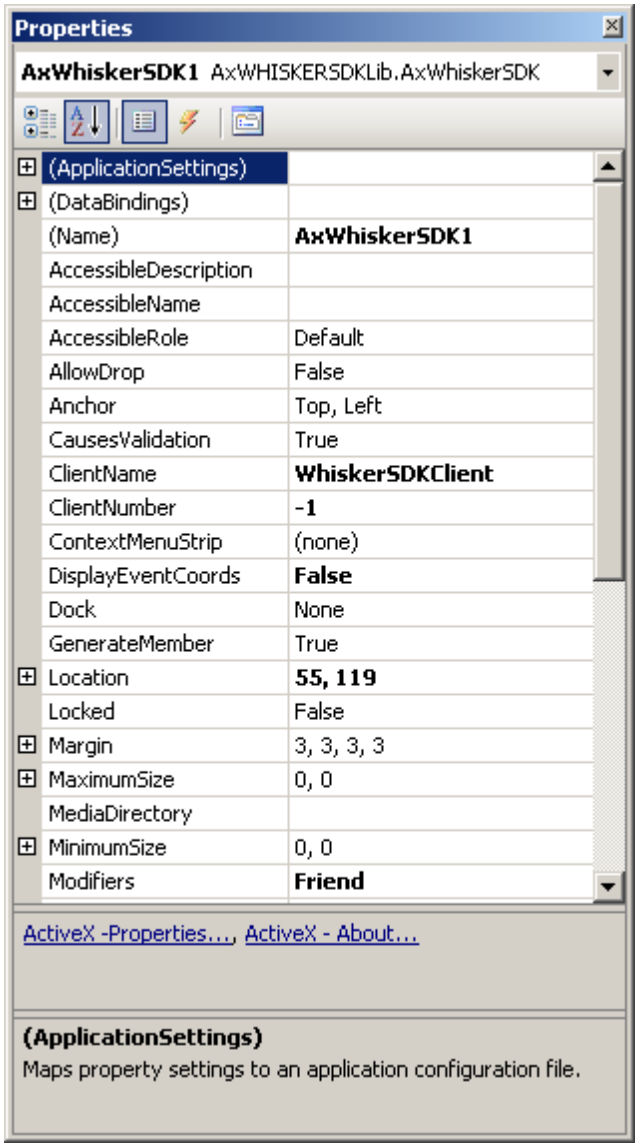
Click OK, and the Control should now be shown in the toolbox.

Adding the Whisker SDK control to the form

Now, if you select the Whisker SDK control in the toolbox (by clicking it), and then click-drag an area on the form, a control will appear on the form. The control is a fixed size (unlike most VB controls, so you can't resize it: it will always snap back). You can put the control where you like on the form, as it wont be visible on the screen when your task is running.

*Note: many aspects of Visual Basic programming require us to distinguish between how things behave whilst we are programming (**design time**) and whilst the application is running (**run time**). So to use the standard parlance, we would say that the Whisker SDK Control is "invisible at runtime". You will become more familiar with this terminology as you go on.*

Setting the Control's properties



Control properties.

When the Control is selected, the **properties panel** will show a set of properties associated with the control object. Some of these properties govern how the control will behave (in terms of which Server it tries to talk to, etc.), and some govern how the form (i.e. VB) will deal with the control.

If you click on a property, the pane at the bottom of the property panel gives a brief introduction to what each property does. Near the top of the list is the important line to begin with **(Name)**.

This shows and sets the name that VB will use internally to refer to the control. By default this will be "AxWhiskerSDK1" which we can change to something else. Change it to "Whisker".

Why the funny name?

The Microsoft technology used in previous versions of VB controls was called ActiveX. The SDK Control is an ActiveX control - hence the 'Ax' before the name

This name 'Whisker' can now be used within your program to access all of the Whisker functions from within the control.

Two special properties of the control are not found on the main list, but are shown as hyperlinks below.

Clicking **ActiveX_About...** shows an 'about box' – which shows you the copyright notice, the version number of the control, and (of course) my name in lights!

Clicking **ActiveX - Properties** shows a **property page** for the control. Many of the other properties relating to your task (the name it will use to identify itself to the server, which server to connect to, where to look for media files, etc) can be accessed from this property page.

Note: These dialogs (unlike the main properties pane) can be viewed at run-time – that is, the user of your program can be shown the ActiveX-Properties page (by means of a control function "ChangeSettings") in case they want to change these settings. This means that you don't have to write the same code in each of your programs to allow the user to choose the server, for example.

Take a moment to look at the properties. Many of them will seem peculiar (and you will never use them), but don't worry! Visual Basic initialises them with sensible defaults.

The important ones are those on the ActiveX-Properties dialog. These include the **ServerName** (the machine which is running Whisker Server): 'localhost' is a IP networking shorthand for 'this machine'. With the servername set to this, the application will try to talk to a server running on the same machine. That's (nearly always) what we'll want to do! Use both the ActiveX-Properties dialog, and the VB Properties panel to play with the settings. Change the name that the program will use when it talks to WhiskerServer (the **ClientName**) – call it something like "Harry's first whisker task" if you want….

Adding code to the form

In VB, code can live in 3 main places.

1. In a form
2. In a code module
3. In a class module

For now, we're only going to scratch the surface of VB programming, and put all of our code in the form. The code in a form is used to manipulate the data represented by that form (remember – the form is a kind of window which can be presented to the user), and also to respond to events that windows sends to the form.

Note: Using code modules and class modules allows you to code in a much more concise & logical way than just plonking all your code in a form. VB .Net is based entirely aroun 'object oriented' approaches. This will allow you to write programs quickly, and produce code that is much easier to understand and to debug!

If you double click on the form, or click the **view code button**, the main window will switch from the 'form designer' to the 'code editor'.

The 'View Code' button

The code editor is where you type in the program. The code for a VB application will **all be inside subroutines** (similar to PROCedures in older forms of BASIC, for those of you with very, very long memories).

Why? Why doesn't it just start at the beginning?

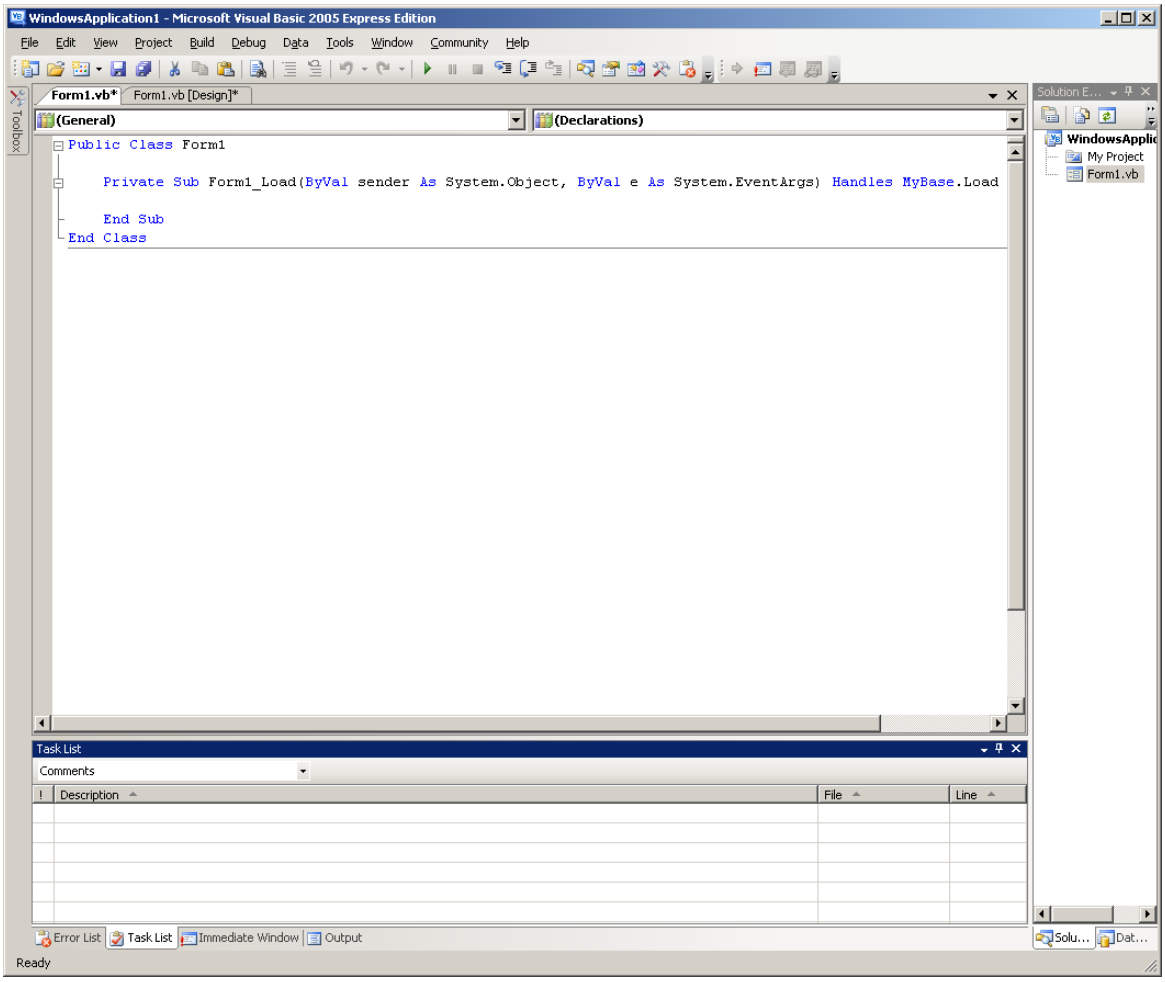
The idea of a program needing a 'first line' is something people who have learned BASIC, or similar languages, are used to. But it doesn't make much difference to the logic where in the code the 'entry point' actually is – for example we could take a traditional BASIC program and put all of the stuff that isn't inside PROCedures, into procedure called 'start', and replace the first line with PROCstart. It wouldn't make any difference to how the program worked.

VB will call your code when it needs to – in response to an event, or at certain times, such as when it has started the program, or when the user has clicked the 'close' button on the form. It is possible to setup a VB project so that it starts with one procedure (sub main) as the 'beginning of the application', rather than loading a form. But you can learn how to do that later!

The Form_Load subroutine

If you double-clicked on the form to get to the code editor, you will now see the beginning and end of a subroutine marked by VB. This is the routine that VB will call as it starts the application, so we can regard it as the 'start' of our application.

If you clicked the View Code button to start the editor, you can insert the Form_Load subroutine by using the two pull-down menus at the top of the editor window. Select 'Form1 (Form 1 events)' in the left hand box, and 'Load' in the right hand box.



Form → Load

Note: The Form_Load subroutine is an Event-handling routine. That is, VB calls this routine when the 'being loaded' event happens to the form.

All event routines, by default, have the same naming practise "objectname_eventname". This was enforced in previous versions of VB, and (like other naming conventions) is worth sticking to, as the bits of code you see elsewhere will use it!

The Event that is handled is the Mybase.Load event, as shown in the Handles part of the declaration. It is this part of the declaration that tells VB which event the subroutine will handle.

If our application just shows this form, the 'load' event happens as soon as the application starts & loads the form.

Calling Whisker functions

Let's say that we want to connect to the whisker server as soon as the form is loaded. We will need to tell VB to connect the control to the server – this is done by calling one of the control's methods "ConnectToServer".

Under the line

```
Private Sub Form1_Load(...
```

Type the following

```
Whisker.ConnectToServer
```

Did something funny happen when you were typing? You should have seen a box pop up as soon as you typed the period (full stop) after whisker. (If you didn't make sure that your SDK control has its name property set to Whisker). This is a handy feature of VB – it knows all of the methods (functions) and properties that can be accessed through the Whisker SDK control object – so you don't have to remember them all!

Now we can test the program. Run WhiskerServer (on the same machine), and once it has started, run the VB program (Run->Start; press F5; or click the blue arrow on the toolbar). You should see the form appear. On the Server Window's left hand panel, click on the plus next to the server, and then on the plus next to Clients. You should see your client's name.

Well done – you've written your first Whisker client.

Extending the client: The VB Status Client Deluxe

Although we have written a client, even in our flush of pride at our new creation, we will have to admit one thing. It's a bit rubbish. So – let's make it into something useful.

A useful client could be one that monitored what was happening on the server. There is a simple client that does just this - the Whisker StatusClient which comes with the server. The status client connects to the server, and asks for information about all connected clients, which it then displays. The display does not update unless the user clicks a button. For our first 'useful' client, we can make an enhanced version of a status client which asks Whisker for a Timer, so that it can refresh its information every couple of seconds.

First - we should give the user something sensible to look at. Select the Form Designer tab (or double click on Form1.vb in the Solution Explorer window).

1. Using the Properties Pane, change the title (**Text** Property) of the Form to be something explanatory, like 'VB .Net Whisker Status Client'.

2. Using the Toolbox, put a **TextBox** control onto the Form.
3. Make it a multiline box (click the multiline option above the box in the designer, or set the **Multiline** property of the textbox to true.
4. Make the textbox bigger, so that it roughly fills the form.

Right - you should now have a form which includes a SDK object (called **Whisker**) and a TextBox object (called **TextBox1**). We now need to add some code to make the form ask Whisker for its current status, and pass that information to the user.

All the code we need is below – paste or type it into the code window...
Note that you can get VB to insert the Whisker_Event and Whisker_Info subroutine lines by using the two menus at the top of the code window...

```
Public Class Form1

    Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles MyBase.Load
        'called by VB when we start the application.
        'It connects to the server, and then asks Whisker for a timer.
        'The Timer will send the EVENT (a string 'refresh') after 2000 ms, and
repeat indefinitely.
        Whisker.ConnectToServer()
        Whisker.TimerSetEvent("refresh", 2000, -1)
    End Sub

    Private Sub Whisker_Event(ByVal sender As Object, ByVal e As
AxWHISKERSDKLib._DWhiskerSDKEvents_EventEvent) Handles Whisker.Event
        'Called by VB whenever an event occurs.
        'Clears the textbox, then sends a command message to the server
        'Updates the client's reported status to reflect how long we've been
connected
        Me.TextBox1.Text = ""
        Whisker.SendToServer("WhiskerStatus")
        Whisker.Status = "Connected for " + Str(e.time \ 1000) + " seconds"
    End Sub

    Private Sub Whisker_Info(ByVal sender As Object, ByVal e As
AxWHISKERSDKLib._DWhiskerSDKEvents_InfoEvent) Handles Whisker.Info
        'Called by VB whenever the server sends information.
        'Prints the information onto the form
        Me.TextBox1.Text &= e.infoMessage & vbCrLf
    End Sub
End Class
```

Now run the program. If you copied the code right, you now have a truly useful program in 6 lines of hand-written code (count them). That's why VB is called a Rapid Application Development language!

Note: in response to receiving the command "WhiskerStatus" the server sends a set of information about all connected clients. See the programmer's guide for details.

Making it truly useful

Although this client works, it is not actually that useful. Why not? Well we wouldn't want to use a Whisker Timer to update the display, because we might well want this client to be running on a different machine to the Server. Generally, it's not a good idea to allow clients from other machines to control devices on the Server across the network (although you can set the server up that way if you want). Because a Timer counts as a device – a remote monitor client will not be able to do this.

We can make it more useful by asking Windows to time the interval for us using the Timer control instead of the Whisker Timer. This is a simple modification to the code above, and is left as an exercise. You will have to look at the online help for VB and find out about Timer controls [Don't worry about updating the client's status with the connection time in your new version – that is only included to show that the client is updating!]

If you can't work out how to do it, then the answer can be found, along with the source code, in the Tutorial 1 Files folder – but try to figure it out for yourself first!

When you've got the program working, you can turn it into an .EXE file which runs without VisualBasic (under *Build→Build Application...*). There's your first completed Whisker Client!

A word of warning:

1. Any program that is written using .NET will only run on machines with the correct version of .NET framework installed.
2. A client, such as this, that uses the SDK Control may only run on machines with the SDK installed [?? Not sure of this. .NET seems like it might deploy its own 'interop' version of the SDK inside the Assembly??]. However, your licence allows you to install the SDK on as many machines as you like (although you can only install one copy of the real Editions of WhiskerServer per licence).

A note on oversimplifications

Second, the Whisker_Event() procedure doesn't check the contents of the event message to find out what it was – this is no good at all for most clients, as they need to respond to many different events. See the tutorial example on converting an !Arachnid client to Whisker to see how this is done.

Pointers for future programming

Some of the remaining Tutorials wont make sense yet to entirely novice programmers. However, glance through to get a feel of where things are going. Then have a look at any of the VB tutorials available on the web, to get more confident with the basics, practice writing some VB code, and refer back there!

Part X

Lab-specific guides

10 Lab-specific guides

10.1 Cambridge, Experimental Psychology

The critical extra issue for the IV group is **safety of i.v. infusion pumps.**

There is a section in the manual that explains the implications of the way the Amplicon hardware initializes itself when the computer is turned on. **There is a risk that all devices are turned on** in this situation, which is dangerous if those devices include intravenous infusion pumps. It is strongly recommended that you read the section of the manual devoted to this topic.

The problem is wholly avoidable, but **it is IMPERATIVE that you install a safety relay** to lock out power to the pumps when Whisker is not running, configured appropriately. It is desirable that you install an **uninterruptible power supply (UPS)** for computers running Whisker.

See also

- Danger with critical devices
- Fail-safe devices (safety relays)
- Configuring fail-safe devices

Also in this lab guide: wiring maps for different systems. Explore the help system to see these.

10.1.1 Maria's box wiring, Jan 2000

Wiring as cocked up and then misdescribed by Sandowne is in fact this...

N.B. All 25-way cables are straight through. Older Camden cables have different pinouts, so use RNC's map (or similar).

Note: black/grey is different from black/dirty white and black/cyan, though they are very similar!
(Black/grey is pin 24, ground; I think black/white is pin 22, unused.)

Device	**Box terminal#**	**25-way connector**	**Wire colour (Maria's 25-way straight-through cables)**	***Camden pin#***	***Camden panel label***
Left lever report	IN 1	9	yellow	*1*	*Left lever report*
Right lever report	IN 2	10	yellow/black	*5*	*Right lever report*
Tray IR detector	IN 3	11	dark green	*3*	*Tray report*
Photobeam #1 FRONT	IN 4	12	dark green/white	*4*	*Spare 5*
Photobeam #2 MIDDLE	IN 5	25	white	*6*	*Spare 6*
Photobeam #3 READ	IN 6	13	cyan (or light green!)	*18*	*Spare 2*
Left lever operate	OUT 1	1	brown	*17*	*Spare 1*
Right lever operate	OUT 2	15	blue	*19*	*Spare 3*
Liquid dipper	OUT 3	2	brown/white	*7*	*Reinforce*
Left light	OUT 4	16	blue/white	*13*	*Left light*

Centre light	OUT 5	3	red		
Right light	OUT 6	17	light blue	*15*	*Right light*
Houselight	OUT 7	4	red/white	*16*	*Houselight*
Syringe pump	OUT 8	18	light blue/black	*12*	*Traylight*
Traylight	OUT 9	5	pink		
Clicker	OUT 10	19	purple		
	OUT 11	6			
	OUT 12	20			
	OUT 13	7			
	OUT 14	21			
	OUT 15	8			
	OUT 16	22			
	+28V	23	grey		
	28V GND	24	grey/black		

Camden pin 20 (spare 4) – not connected
Not used on the Camden panels: spare 4 / loudspeaker / centre
Must connect Camden panel COMMON and OPERATE to ground before the boxes will work.
Sandowne had described RIGHT LIGHT as connected to OUT 5, and had connected it to OUT 5, but they should have used OUT 6. Oops.

10.1.2 Maria/Rutsuko - 25-way cables

For IV boxes.

Pin	Colour (Rutsuko)	Colour (Maria)	Variant cables (Rutsuko)
1	brown		
2	brown/white		
3	red		red
4	red/white		pink
5	pink		pink/black
6	pink/black		red/white
7	orange		
8	orange/black		
9	yellow		
10	yellow/black		
11	dark green		
12	dark green/white		
13	light green	cyan	
14	light green/black	cyan/black	
15	blue		
16	blue/white		
17	sky blue	light blue	
18	sky blue/black	light blue/black	
19	purple		
20	purple/white		
21	black		

22	black/white		
23	grey		
24	grey/black		
25	white		

10.1.3 Maria/Rutsuko - box definition files

```
WhiskerServer v2.0 - DEVICE DEFINITION FILE - DO NOT ALTER THIS LINE
########################################################################

########################################################################
#       These definitions are for Maria + Rutsuko's boxes
########################################################################

# This file defines device names used by the WhiskerServer program.
# Lines beginning with a hash (#) are comments and are ignored.
#
# Each line takes the following format:
#
#       <device_type> <device_number> <group_name> <device_name>
#
# where <device_type> may be
#       line            = digital I/O line
#       display         = display device (monitor)
#       audio           = audio device (sound card, or half-sound card; see
manual)
#
# The <device_number> is the number of the line/display/audio device that you see
# on the server's console - the number that you would otherwise claim.
#
# The COMBINATION of the <group_name> and <device_name> must be unique.
# If the server encounters non-unique device group/name pairs in this file,
# all but the first will be ignored.
# Neither the <group_name> nor the <device_name> may start with a number.

########################################################################
#
# For full security, clients should check that the lines they get are inputs/
outputs
# as appropriate, and set the correct reset state. For example, to claim box
# 0, you can simple execute the following command:
#
#       ClaimGroup box0 -prefix whatever -suffix ifyouwant
#
# but for safety, you should also issue these:
#
#       ClaimLine box0 NOSEPOKE -input
#       ClaimLine box0 LEFTLEVER -input
#       ClaimLine box0 RIGHTLEVER -input
#       ClaimLine box0 LOCOBEAM_FRONT -input
#       ClaimLine box0 LOCOBEAM_MIDDLE -input
#       ClaimLine box0 LOCOBEAM_REAR -input
#       ClaimLine box0 HOUSELIGHT -output -ResetOff
#       ClaimLine box0 TRAYLIGHT -output -ResetOff
#       ClaimLine box0 PUMP -output -ResetOff
#       ClaimLine box0 DIPPER -output -ResetOff
#       ClaimLine box0 LEFTLEVERCONTROL -output -ResetOff
#       ClaimLine box0 RIGHTLEVERCONTROL -output -ResetOff
#       ClaimLine box0 LEFTLIGHT -output -ResetOff
#       ClaimLine box0 RIGHTLIGHT -output -ResetOff
#
```

```
######################################################################

# Box 0 definition

line   0      box0    NOSEPOKE
line   1      box0    LEFTLEVER
line   2      box0    RIGHTLEVER
line   3      box0    LOCOBEAM_FRONT
line   4      box0    LOCOBEAM_MIDDLE
line   5      box0    LOCOBEAM_REAR

line   24     box0    HOUSELIGHT
line   25     box0    LEFTLIGHT
line   26     box0    RIGHTLIGHT
line   27     box0    TRAYLIGHT
line   28     box0    PUMP
line   29     box0    DIPPER
line   30     box0    LEFTLEVERCONTROL
line   31     box0    RIGHTLEVERCONTROL

# Box 1 definition

line   6      box1    NOSEPOKE
line   7      box1    LEFTLEVER
line   8      box1    RIGHTLEVER
line   9      box1    LOCOBEAM_FRONT
line   10     box1    LOCOBEAM_MIDDLE
line   11     box1    LOCOBEAM_REAR

line   32     box1    HOUSELIGHT
line   33     box1    LEFTLIGHT
line   34     box1    RIGHTLIGHT
line   35     box1    TRAYLIGHT
line   36     box1    PUMP
line   37     box1    DIPPER
line   38     box1    LEFTLEVERCONTROL
line   39     box1    RIGHTLEVERCONTROL

# Box 2 definition

line   12     box2    NOSEPOKE
line   13     box2    LEFTLEVER
line   14     box2    RIGHTLEVER
line   15     box2    LOCOBEAM_FRONT
line   16     box2    LOCOBEAM_MIDDLE
line   17     box2    LOCOBEAM_REAR

line   40     box2    HOUSELIGHT
line   41     box2    LEFTLIGHT
line   42     box2    RIGHTLIGHT
line   43     box2    TRAYLIGHT
line   44     box2    PUMP
line   45     box2    DIPPER
line   46     box2    LEFTLEVERCONTROL
line   47     box2    RIGHTLEVERCONTROL

# Box 3 definition

line   72     box3    NOSEPOKE
line   73     box3    LEFTLEVER
line   74     box3    RIGHTLEVER
line   75     box3    LOCOBEAM_FRONT
```

```
line   76     box3    LOCOBEAM_MIDDLE
line   77     box3    LOCOBEAM_REAR

line   48     box3    HOUSELIGHT
line   49     box3    LEFTLIGHT
line   50     box3    RIGHTLIGHT
line   51     box3    TRAYLIGHT
line   52     box3    PUMP
line   53     box3    DIPPER
line   54     box3    LEFTLEVERCONTROL
line   55     box3    RIGHTLEVERCONTROL



# Box 4 definition

line   78     box4    NOSEPOKE
line   79     box4    LEFTLEVER
line   80     box4    RIGHTLEVER
line   81     box4    LOCOBEAM_FRONT
line   82     box4    LOCOBEAM_MIDDLE
line   83     box4    LOCOBEAM_REAR

line   56     box4    HOUSELIGHT
line   57     box4    LEFTLIGHT
line   58     box4    RIGHTLIGHT
line   59     box4    TRAYLIGHT
line   60     box4    PUMP
line   61     box4    DIPPER
line   62     box4    LEFTLEVERCONTROL
line   63     box4    RIGHTLEVERCONTROL

# Box 5 definition

line   84     box5    NOSEPOKE
line   85     box5    LEFTLEVER
line   86     box5    RIGHTLEVER
line   87     box5    LOCOBEAM_FRONT
line   88     box5    LOCOBEAM_MIDDLE
line   89     box5    LOCOBEAM_REAR

line   64     box5    HOUSELIGHT
line   65     box5    LEFTLIGHT
line   66     box5    RIGHTLIGHT
line   67     box5    TRAYLIGHT
line   68     box5    PUMP
line   69     box5    DIPPER
line   70     box5    LEFTLEVERCONTROL
line   71     box5    RIGHTLEVERCONTROL
```

10.1.4 Pat/Dan - box definition files

```
WhiskerServer v2.0 - DEVICE DEFINITION FILE - DO NOT ALTER THIS LINE
#####################################################################

#####################################################################
#       These definitions are for Pat + Dan's boxes
#####################################################################

# This file defines device names used by the WhiskerServer program.
# Lines beginning with a hash (#) are comments and are ignored.
#
```

```
# Each line takes the following format:
#
#       <device_type> <device_number> <group_name> <device_name>
#
# where <device_type> may be
#       line            = digital I/O line
#       display         = display device (monitor)
#       audio           = audio device (sound card, or half-sound card; see
manual)
#
# The <device_number> is the number of the line/display/audio device that you see
# on the server's console - the number that you would otherwise claim.
#
# The COMBINATION of the <group_name> and <device_name> must be unique.
# If the server encounters non-unique device group/name pairs in this file,
# all but the first will be ignored.
# Neither the <group_name> nor the <device_name> may start with a number.


######################################################################
#
# For full security, clients should check that the lines they get are inputs/
outputs
# as appropriate, and set the correct reset state. For example, to claim box
# 0, you can simple execute the following command:
#
#       ClaimGroup box0 -prefix whatever -suffix ifyouwant
#
# but for safety, you should also issue these:
#
#       ClaimLine box0 NOSEPOKE -input
#       ClaimLine box0 LEFTLEVER -input
#       ClaimLine box0 RIGHTLEVER -input
#       ClaimLine box0 LOCOBEAM_FRONT -input
#       ClaimLine box0 LOCOBEAM_MIDDLE -input
#       ClaimLine box0 LOCOBEAM_REAR -input
#       ClaimLine box0 HOUSELIGHT -output -ResetOff
#       ClaimLine box0 TRAYLIGHT -output -ResetOff
#       ClaimLine box0 PUMP -output -ResetOff
#       ClaimLine box0 DIPPER -output -ResetOff
#       ClaimLine box0 LEFTLEVERCONTROL -output -ResetOff
#       ClaimLine box0 RIGHTLEVERCONTROL -output -ResetOff
#       ClaimLine box0 LEFTLIGHT -output -ResetOff
#       ClaimLine box0 RIGHTLIGHT -output -ResetOff
#
######################################################################

# Box 0 definition

line    0       box0    NOSEPOKE
line    3       box0    LEFTLEVER
line    6       box0    RIGHTLEVER
line    9       box0    LOCOBEAM_FRONT
line    12      box0    LOCOBEAM_MIDDLE
line    15      box0    LOCOBEAM_REAR

line    24      box0    HOUSELIGHT
line    27      box0    TRAYLIGHT
line    30      box0    PUMP
line    33      box0    DIPPER
line    36      box0    LEFTLEVERCONTROL
line    39      box0    RIGHTLEVERCONTROL
line    42      box0    LEFTLIGHT
```

```
line    45      box0    RIGHTLIGHT

# Box 1 definition

line    1       box1    NOSEPOKE
line    4       box1    LEFTLEVER
line    7       box1    RIGHTLEVER
line    10      box1    LOCOBEAM_FRONT
line    13      box1    LOCOBEAM_MIDDLE
line    16      box1    LOCOBEAM_REAR

line    25      box1    HOUSELIGHT
line    28      box1    TRAYLIGHT
line    31      box1    PUMP
line    34      box1    DIPPER
line    37      box1    LEFTLEVERCONTROL
line    40      box1    RIGHTLEVERCONTROL
line    43      box1    LEFTLIGHT
line    46      box1    RIGHTLIGHT

# Box 2 definition

line    2       box2    NOSEPOKE
line    5       box2    LEFTLEVER
line    8       box2    RIGHTLEVER
line    11      box2    LOCOBEAM_FRONT
line    14      box2    LOCOBEAM_MIDDLE
line    17      box2    LOCOBEAM_REAR

line    26      box2    HOUSELIGHT
line    29      box2    TRAYLIGHT
line    32      box2    PUMP
line    35      box2    DIPPER
line    38      box2    LEFTLEVERCONTROL
line    41      box2    RIGHTLEVERCONTROL
line    44      box2    LEFTLIGHT
line    47      box2    RIGHTLIGHT

# Box 3 definition

line    72      box3    NOSEPOKE
line    75      box3    LEFTLEVER
line    78      box3    RIGHTLEVER
line    81      box3    LOCOBEAM_FRONT
line    84      box3    LOCOBEAM_MIDDLE
line    87      box3    LOCOBEAM_REAR

line    96      box3    HOUSELIGHT
line    99      box3    TRAYLIGHT
line    102     box3    PUMP
line    105     box3    DIPPER
line    108     box3    LEFTLEVERCONTROL
line    111     box3    RIGHTLEVERCONTROL
line    114     box3    LEFTLIGHT
line    117     box3    RIGHTLIGHT

# Box 4 definition

line    73      box4    NOSEPOKE
line    76      box4    LEFTLEVER
line    79      box4    RIGHTLEVER
line    82      box4    LOCOBEAM_FRONT
```

```
line    85      box4    LOCOBEAM_MIDDLE
line    88      box4    LOCOBEAM_REAR

line    97      box4    HOUSELIGHT
line    100     box4    TRAYLIGHT
line    103     box4    PUMP
line    106     box4    DIPPER
line    109     box4    LEFTLEVERCONTROL
line    112     box4    RIGHTLEVERCONTROL
line    115     box4    LEFTLIGHT
line    118     box4    RIGHTLIGHT

# Box 5 definition

line    74      box5    NOSEPOKE
line    77      box5    LEFTLEVER
line    80      box5    RIGHTLEVER
line    83      box5    LOCOBEAM_FRONT
line    86      box5    LOCOBEAM_MIDDLE
line    89      box5    LOCOBEAM_REAR

line    98      box5    HOUSELIGHT
line    101     box5    TRAYLIGHT
line    104     box5    PUMP
line    107     box5    DIPPER
line    110     box5    LEFTLEVERCONTROL
line    113     box5    RIGHTLEVERCONTROL
line    116     box5    LEFTLIGHT
line    119     box5    RIGHTLIGHT
```

Index

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- H -

- I -

- K -

- L -

- M -

- N -

- O -

- P -

- R -

- S -

- T -

- U -

- V -

- W -