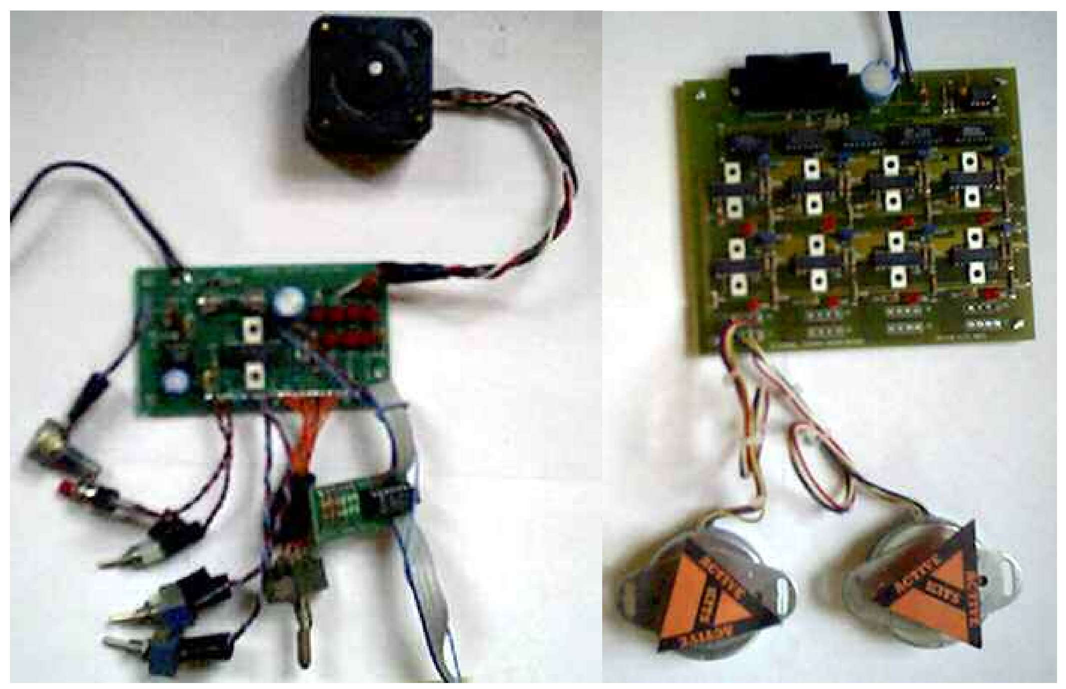
The Computer Control Program for the Active Surplus Single-Channel and Multi-Channel Stepper Motor Kits

Copyright © 2003 Geoff Phillips geoff@phillips.eu.org



Contents

A Quick Warning 3
A Quick Guide to Stepper Motors 3
Quick-Start for the Single-Channel Kit 3
Quick-Start for the Multi-Channel Stepper Motor Kit 5
Licensing Information 7
    What the Hell Did All That Mean? 7
    The PortTalk Driver by Craig Peacock 7
Introduction 8
Files Distributed With This Program 8
The Single-Channel Command Set 10
    The 'step' command: 10
    The 'wait' command: 10
The Multi-Channel Command Set 11
    The 'step' command: 11
    The 'wait' command: 11
    Stepping More Than One Motor at a Time – Begin and End Statements: 12
The 'guistep' Program 13
The 'cmdstep' Program 15
    Using cmdstep in Multi-Channel Mode: 16
The Limit-Input System 16
The Different Delay Methods 19
The Source Code 20
    'stepperClass.dll' 20
    'cmdstep.exe' 20
    'guistep.exe' 20
Using the stepperClass.dll Library in Your Programs 22
    Using stepperClass 22
    Example Programs using stepperClass 23
    Using multiChannelStepperClass 24
    Example Program using multiChannelStepperClass 25
Multi-Channel Address Mode 26
Troubleshooting 27
Bibliography 27
References 27

A Quick Warning

Interfacing with the PC's parallel port is the easiest way to control external circuitry via a computer, however there are risks involved.

It is possible to damage the parallel port of the computer by accidentally shorting connections together, or by applying voltage to the wrong pins. Most parallel ports are built directly into the motherboard of PCs, and fixing a broken parallel port will probably lead to replacing the entire motherboard.

Therefore it is highly recommended that the reader obtains an extra "IO card" with a parallel port on it. This will provide you with an extra parallel port on your computer with which you will be slightly more at ease in using. If anything does go wrong, it will only be this card you need to replace rather than the whole motherboard.

("USB to parallel converters" are available that provide an extra parallel port for use with printers, however these adapters are currently not compatible with this program)

Although I am warning you of this now, I personally could never be bothered to buy an IO card. If you do use your regular parallel port – be careful!

A Quick Guide to Stepper Motors

Stepper motors are similar to regular motors in that they turn round, but do so in small steps. The number of steps in circle varies depending on the stepper motor. In this manual it will be assumed that there are 100 steps in a circle (i.e. a 3.6° motor). This is not always the case, and you may find that turning your stepper motor round by 100 steps is not a full circle. It is fairly easy to figure out the number of steps for your stepper motor by trial and error.

Another issue to consider is the motor's maximum rotational speed, which is inversely proportional to the delay given in between each step. Therefore the delay time between steps will have a minimum value for which the motor will still turn round correctly. For delay times lower than this minimum value, you will find that the motor fails to complete every step and may finish in an unpredictable position. Finding this minimum delay time is again usually a matter of trial and error. For this manual it will be assumed that the minimum delay between steps is around 3 milliseconds (thousandths of a second).

Quick-Start for the Single-Channel Kit

Want to get your single-channel stepper motor turning right now? Well what are you waiting for?! (Skip to the next chapter if you have the multi-channel kit)
Follow these instructions:

- Set up your hardware, make sure it works using the on-board testing mode before trying to control it from the computer
- Connect the kit to the parallel port of your computer
- Run guistep.exe
- Go to File → New, to create a new command file
- Set the direction to Clockwise, the distance to full-step, the number of steps to 100 and the delay between steps to 10 milliseconds
- Click "Add Step Command", this will paste the command into the edit box below
- Click on the ▶ All button at the bottom

It should make your motor turn in a full circle clockwise in one second. If it does not turn at all refer to the Troubleshooting Chapter.
How about something a little more fancy?

- Go to File → New, to create a new command file
- Set the delay between steps to be 20, and click "Add Step Command"
- Now select the command in the edit box below using the mouse, and copy the command text by pressing Ctrl-C
- Now paste command over and over again using Ctrl-V so that there are 12 copies of the command in the edit box
- Now manually edit the commands in the edit box so that they read as follows:

```
step , cw , full , 25  , 20000us
step , cw , full , 38  , 13333us
step , cw , full , 50  , 10000us
step , cw , full , 63  , 8000us
step , cw , full , 75  , 6667us
step , cw , full , 88  , 5714us
step , cw , full , 100 , 5000us
step , cw , full , 113 , 4444us
step , cw , full , 125 , 4000us
step , cw , full , 138 , 3636us
step , cw , full , 150 , 3333us
step , cw , full , 163 , 3076us
```

- Note that the delay time between steps (the last column) is now measured in microseconds (millionths of a second) instead of milliseconds because each number is followed by the characters "us"
- Try running the commands by clicking the ▶ **All** button
- You should find that the motor starts slow, then goes faster and faster
- Try checking the "Loop Commands" check box and run the program again. It should keep running the commands until you click either the Pause or Stop buttons
- However it might not be going as fast as it should because of the way in which the program delays itself between performing each step of the motor
- To improve the execution, go to Options → Advanced Timing Options…
- Then Set the "Priority Class" to High and the "Priority Level" to Highest and click OK
- Try the commands again by clicking the ▶ **All** button
  - You will find that although the program will execute the commands more accurately it will take longer to respond to the stop and pause buttons – be patient, it will take up to 10 seconds to stop
- You can also try running each command one at a time by clicking the ▶ **One** button repeatedly
- You can navigate the command list using the regular fast-forward and re-wind buttons
- To Save your command list, go to File → Save As…
  - It will be saved with a ".txt" extension so that you can easily edit it using notepad or your favourite text editor
- Have fun!

Quick-Start for the Multi-Channel Stepper Motor Kit

Want to get your multi-channel stepper motor turning right now? Well what are you waiting for?! Follow these instructions:

- Set up your hardware, make sure it works using the on-board testing mode before trying to control it from the computer
- Connect the kit to the parallel port of your computer
- Run guistep.exe
- **Go to the Options menu and select "Multi-Channel Stepper Mode"**
- **Check that the "8-Channel Clock Mode" is also checked**
- Go to File → New, to create a new command file
- Set the motor number to 1, the direction to Clockwise, the distance to full-step, the number of steps to 100 and the delay between steps to 10 milliseconds
- Click "Add Step Command", this will paste the command into the edit box below
- Click on the ▶ All button at the bottom

It should make motor one turn in a full circle clockwise in one second. If it does not turn at all refer to the Troubleshooting Chapter.

How about something a little more fancy?

- Go to File → New, to create a new command file
- Set the delay between steps to be 20, and click "Add Step Command"
- Now select the command in the edit box below using the mouse, and copy the command text by pressing Ctrl-C
- Now paste command over and over again using Ctrl-V so that there are 12 copies of the command in the edit box
- Now manually edit the commands in the edit box so that they read as follows:

```
step , 1 , cw , full , 25  , 20000us
step , 1 , cw , full , 38  , 13333us
step , 1 , cw , full , 50  , 10000us
step , 1 , cw , full , 63  , 8000us
step , 1 , cw , full , 75  , 6667us
step , 1 , cw , full , 88  , 5714us
step , 1 , cw , full , 100 , 5000us
step , 1 , cw , full , 113 , 4444us
step , 1 , cw , full , 125 , 4000us
step , 1 , cw , full , 138 , 3636us
step , 1 , cw , full , 150 , 3333us
step , 1 , cw , full , 163 , 3076us
```

- Note that the delay time between steps (the last column) is now measured in microseconds (millionths of a second) instead of milliseconds because each number is followed by the characters "us"
- Try running the commands by clicking the ▶ All button
- You should find that the motor starts slow, then goes faster and faster
- Try checking the "Loop Commands" check box and run the program again. It should keep running the commands until you click either the Pause or Stop buttons
- However it might not be going as fast as it should because of the way in which the program delays itself between performing each step of the motor
- To improve the execution, go to Options → Advanced Timing Options…
- Then Set the 'Priority Class' to High and the 'Priority Level' to Highest and click OK

- Try the commands again by clicking the ▶ **All** button
  - You will find that although the program will execute the commands more accurately it will take longer to respond to the stop and pause buttons – be patient, it will take up to 10 seconds to stop
- You can also try running each command one at a time by clicking the ▶ **One** button repeatedly
- You can navigate the command list using the regular fast-forward and re-wind buttons
- To Save your command list, go to File → Save As…
  - It will be saved with a ".txt" extension so that you can easily edit it using notepad or your favourite text editor
- Have fun!

Licensing Information

The “guistep”, “cmdstep” and “stepperClass” programs are free software; you can redistribute them and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation; version 2 of the License.

These programs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program (See "LICENCE.txt"); if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

What the Hell Did All That Mean?

The long and the short of it is that:

- I have written these programs in my spare time in the hope that you will find them useful, but don’t sue me if they do not work or crash your computer etc.
- You are freely permitted to copy / share / modify my programs and source code BUT:
    - You are not allowed to make people pay to get a copy of them
    - If you do “base your program on my source code” then you must also release your source code under the GNU General Public Licence.
    - Therefore:
        - You must freely distribute your program making clear the modifications you have made
        - You cannot charge people for a copy of your program

For full and proper details read “LICENCE.txt”.

The PortTalk Driver by Craig Peacock

This program makes use of the "PortTalk" driver written by Craig Peacock. This is used to enable low-level control over the parallel port under Windows NT (NT4, 2000 and XP).

The PortTalk driver and source code including "PortTalk_IOCTL.h" is freely available at: http://www.beyondlogic.org

The source code for PortTalk cannot be distributed with this application due to PortTalk copyright restrictions. These are:

" Any publication or distribution of this code in source form is prohibited
without prior written permission of the copyright holder. "

Craig Peacock 2002, Craig.Peacock@beyondlogic.org

Introduction

This program has been written to control the Active Surplus Single-Channel and Multi-Channel Stepper Motor Kits from a standard IBM-compatible PC running Microsoft Windows 95, 98, Me, NT4, 2000 or XP, via the computer's parallel port.

There are three separate programs that make up this distribution:

i) **guistep.exe**
This is a Graphical User Interface (GUI (pronounced goo-ee)) program used to control the stepper motor kits

ii) **cmdstep.exe**
This is a command-line program used to control the kits, which can also be used in combination with scripting languages such as perl (http://www.perl.org)

iii) **stepperClass.dll**
This is a shared library used by guistep.exe and cmdstep.exe. It contains the code used to control the stepper motor. This file can be used as library in your programs to control the stepper motor kit

- This manual goes through how to use the programs and the commands they accept, and then goes on to show how you might use the stepperClass.dll in your own programs.
- There is an optional three bit feedback system that allows the zeroing, calibration or environment sensing for the stepper motor system. For Information about this system read The Limit-Input System Chapter.
- The source code included with this distribution is detailed in the The Source Code Chapter.
- This manual also has a reference for the hardware interface used in the programs' different modes of operation.

If you happen to find any bugs in any of the programs, or have any requests or recommendations please send them to geoff@phillips.eu.org

Files Distributed With This Program

`README.pdf` - This document
`LICENCE.txt` - The GNU General Public Licence; version 2
`cmdstep.exe` - The 'cmdstep' program
`example_single.txt` - A single-channel example command file
`example_multi.txt` - A multi-channel example command file
`guistep.exe` - The 'guistep' program
`porttalk.sys` - The Windows NT4, 2000 and XP port driver, written by Craig Peacock (Also available from http://www.beyondlogic.org)
`porttalk_uninstall.exe` - A program to uninstall the porttalk driver from the system
`stepperClass.dll` - The library file used by both guistep and cmdstep
`cmdstep_examples\example_single.pl` - A single channel perl script example of interfacing with cmdstep
`cmdstep_examples\example_multi.pl` - A multi-channel perl script example of interfacing with cmdstep

cmdstep_examples\limits.pl - A single-channel perl script example of interfacing with cmdstep and using the limit inputs
src - The main source code directory
src\step.dsw - The main Microsoft Visual C++ 6 Workspace

The Single-Channel Command Set

There are two commands that guistep and cmdstep program understand in single-channel mode. Both of them start with the name of the command, and are then followed by a comma delimited string of command parameters. The commands are case-insensitive, so upper or lower case can be used and all white space apart from new-line characters is ignored.

The 'step' command:

```
step , DIRECTION , DISTANCE , NUM_STEPS , DELAY [, LIMIT_MASK]
```

Where:

- DIRECTION can be either 'cw' or 'ccw' for clockwise or counter-clockwise respectively
- DISTANCE can be either 'full' or 'half' for either full or half steps respectively
- NUM_STEPS is the required integer number of steps. This can be set to **"inf"** in order to make the motor step forever
- DELAY is the required integer delay between steps measured in milliseconds (thousandths of a second). This can also be specified in microseconds (millionths of a second) by appending 'us' to the number, i.e. '1000us' would give a delay of one millisecond.
- LIMIT_MASK is an optional bit-mask to indicate under which limit-input conditions the command execution should be halted. For more information read The Limit-Input System Chapter.

Example:

The following command turns the motor clockwise in a circle with full steps (assuming there are 100 steps in a circle, therefore a 3.6° motor), with a delay of 8 milliseconds between steps:

```
step , cw , full, 100 , 8
```

The 'wait' command:

```
wait , WAIT_TIME [, LIMIT_MASK]
```

Where:

- WAIT_TIME is the required integer period of delay measured in milliseconds (thousandths of a second). This can be set to **"inf"** in order to make the program wait forever – which is generally used along with a limit mask
- LIMIT_MASK is an optional bit-mask to indicate under which limit-input conditions the command execution should be halted. For more information read The Limit-Input System Chapter.

Example:

The following example steps the motor half a turn clockwise, waits for half a second then steps the motor back half a turn counter-clockwise:

```
step , cw , full , 50 , 8
wait , 500
step , ccw , full , 50 , 8
```

The Multi-Channel Command Set

The multi-channel command set is very similar to the single-channel command set, however the step commands include a motor number, and there also are two extra commands 'begin' and 'end' which allow the user to turn more than motor at a time.

The 'step' command:

```
step , MOTOR , DIRECTION , DISTANCE , NUM_STEPS , DELAY [, LIMIT_MASK]
```

Where:

- MOTOR is the motor number
- DIRECTION can be either 'cw' or 'ccw' for clockwise or counter-clockwise respectively
- DISTANCE can be either 'full' or 'half' for either full or half steps respectively
- NUM_STEPS is the required integer number of steps. This can be set to **"inf"** in order to make the motor step forever
- DELAY is the required integer delay between steps measured in milliseconds (thousandths of a second). This can also be specified in microseconds (millionths of a second) by appending 'us' to the number, i.e. '1000us' would give a delay of one millisecond.
- LIMIT_MASK is an optional bit-mask to indicate under which limit-input conditions the command execution should be halted. For more information read The Limit-Input System Chapter.

Example:

The following command turns motor one clockwise in a circle with full steps (assuming there are 100 steps in a circle, therefore a 3.6° motor), with a delay of 8 milliseconds between steps:

```
step , 1 , cw , full, 100 , 8
```

The 'wait' command:

```
wait , WAIT_TIME [, LIMIT_MASK]
```

Where:

- WAIT_TIME is the required integer period of delay measured in milliseconds (thousandths of a second). This can be set to **"inf"** in order to make the program wait forever – which is generally used along with a limit mask
- LIMIT_MASK is an optional bit-mask to indicate under which limit-input conditions the command execution should be halted. For more information read The Limit-Input System Chapter.

Example:

The following example steps motor one half a turn clockwise, waits for half a second then steps the motor back half a turn counter-clockwise:

```
step , 1 , cw , full , 50 , 8
wait , 500
step , 1 , ccw , full , 50 , 8
```

Stepping More Than One Motor at a Time – Begin and End Statements:

If you have the multi-channel stepper motor kit, you will of course want to turn more than one motor at the same time, probably in different directions and rates. To do this you just need to enclose multiple **`step`** commands within a **`begin`** and an **`end`** statement.

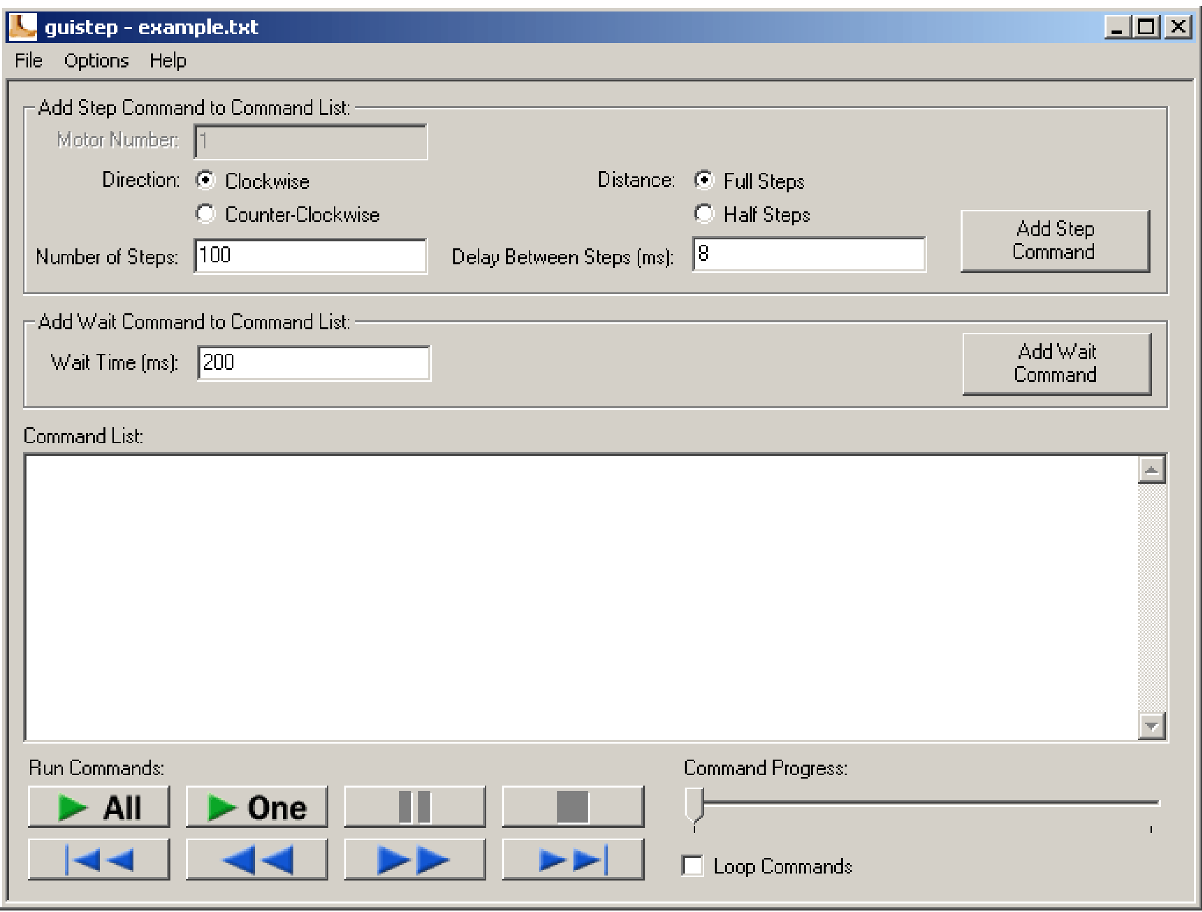
The following example steps motor 1 clockwise for two turns with an 8 millisecond delay between steps, and also steps motor 2 counter-clockwise for one turn with a 16 millisecond delay between steps. It then waits for half a second before turning the motors as before but in the opposite directions:

```
begin
step , 1 , cw , full , 200 , 8
step , 2 , ccw , full , 100 , 16
end

wait , 500

begin
step , 1 , cw , full , 200 , 8
step , 2 , ccw , full , 100 , 16
end
```

The number of step commands between a **`begin`** and an **`end`** statement is not just limited to two, there can be up to 8 commands using the ‘8-Channel Clock Mode’. To turn more than 8 motors a different mode called ‘Multi-Channel Address Mode’ can be used however the hardware included with the standard 8-Channel kit does not support this. Read the Multi-Channel Address Mode Chapter for more information.

The ‘guistep’ Program

The Graphical User Interface (GUI (pronounced goo-ee)) program ‘guistep.exe’ is a simple interface that lets the user create a list of commands, then execute them with the option to stop, pause and navigate the command list.



The upper part of the interface comprises of simple point-and-click controls to add step and wait commands to the command list in the centre.

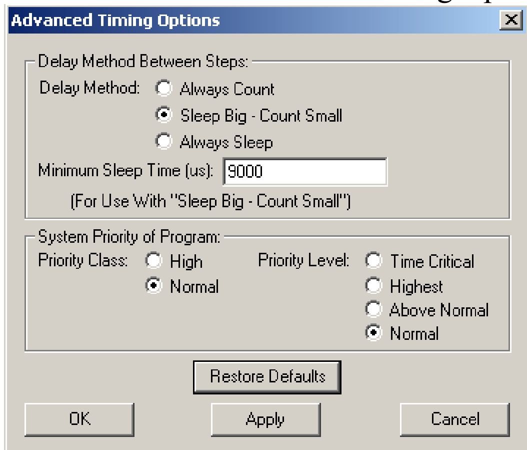
The command list is an edit box in which the user can manually type in commands, re-arrange and delete them in the usual fashion.

The Run Command Controls at the bottom of the window allow the user to navigate the command list using the “fast-forward” and “rewind” buttons, and run the commands using the ▶ All or ▶ One buttons. During execution the user can pause or stop the commands with the appropriate buttons.

There is a File menu to save the command list to a text file, or open an existing command file.

The Options menu allows the user to select either LPT1 or LPT2 as the parallel port in use, and has an option to pause the command execution when a limit is found (See the The Limit-Input System Chapter).There are also options to switch in between single and multi channel modes of operation.

The Options menu also leads to the Advanced Timing Options dialog:



This interface allows the user to select the delay method between steps and set the priority of the program. For more information on the delay methods read The Different Delay Methods Chapter.

The ‘cmdstep’ Program

The CoMmanD Step program ‘cmdstep’ is a program that provides control of the stepper motor via the command-line. By itself it provides the same amount of control over the stepper motor as guistep, however in combination with a powerful scripting language such as “perl” (http://www.perl.org) the possibilities of its use are endless.

The cmdstep program is essentially a “wrapper” for the stepperClass.dll library. If you would like use the library functions in stepperClass.dll in your own programs or read the Using the stepperClass.dll Library Chapter.

It will be assumed that the user is familiar with using command line programs, as the instructions for navigating to the cmdstep program directory from the command line will not be given.

Typing “cmdstep –h” at the command prompt yields the following information:

```
cmdstep - Version 1.00
Copyright (C) 2003 Geoff Phillips
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute
it under the terms of the GNU General Public License; Version 2;
for details read `LICENCE.txt'.
USAGE: cmdstep [OPTIONS] [COMMAND]
After performing any given commands the program returns the number
of steps performed in the last 'step' command.
For details of the COMMAND syntax read `README.pdf'.
OPTIONS:
   -h         - This information
   -f FILE    - Performs commands in a given FILE
   -i         - Performs commands taken from the standard input
   -l         - Outputs the value of the 'limits' at the time of exit
                See README.pdf for more information about the limit inputs
   -e         - Outputs the execution time of the last command in milliseconds
   -q         - No output will be given apart from error messages
   -2         - LPT2 will be used as the parallel port output, instead of LPT1
Multi-Channel Options:
   -m         - The commands given will be executed in multi-channel mode
                instead of the default single-channel mode
   -a         - The method used to drive the stepper motors will be the
                proposed 'Address Mode' instead of the default
                '8-Channel Clock Mode' which is supported by the standard
                hardware. See README.pdf for details
Priority Options:
   -p NUMBER  - Set the priority of the program to number between 1 and 10
                 Priority Number:  Priority Class: Priority Level:
                 1 (Default)       Normal          Normal
                 2                 Normal          Above Normal
                 3                 Normal          Highest
                 4                 Normal          Time Critical
                 5                 High            Normal
                 6                 High            Above Normal
                 7                 High            Highest
                 8                 High            Time Critical
Timing Options:
   -t NUMBER  - Set the timing method of the program to a number
                between 1 and 3
                 Timing Number:    Timing Method:
                 1                 Always Sleep
                 2 (Default)       Sleep Big - Count Small
                 3                 Always Count
   -s TIME    - Set the minimum Sleep time in milliseconds of the
                'Sleep Big - Count Small' timing method
Read `README.pdf' for information about the different timing methods.
```

There are three ways in which to give commands to cmdstep:

i) On the command-line:

Example: `cmdstep step , cw , full , 100 , 8`

N.B. only a single command can be given

ii) By specifying a file containing commands
Example: `cmdstep –f commands.txt`
Where the file commands.txt contains the required commands

iii) By feeding commands to cmdstep using standard-input
Example 1, using the keyboard:
- type `cmdstep -i` at the command prompt
- type all of the required commands needed for execution, including new-lines
- on the last blank new-line press Ctrl-Z to create "an end-of-file signal"
- then hit enter to perform the commands

Example 2, using a command file:
`cmdstep –i < commands.txt`
Where commands.txt contains the required commands.

The output of another program can also be "piped" to cmdstep when it is in standard input mode.
Example 3, using a pipe:
`another_program | cmdstep -i`

The standard input method is best used when more than one command needs to be given and the use of a temporary command file is unwanted.
For an illustration of this read the example.pl file in the **`cmdstep_examples`** directory

Using cmdstep in Multi-Channel Mode:

To use cmdstep in multi-channel mode you must always add the –m option, as the program runs in single-channel mode by default, e.g:

`cmdstep –m step , 1 , cw , full , 100 , 8`

The Limit-Input System

The original single stepper motor kits do not come with any documented method of feedback to the stepper motor program. Therefore the user was unable to perform operations such as "zeroing", calibration, or event sensing.

The step and wait commands of this program have the optional ability to stop their execution when an input is received from the parallel port, thus *limiting* their execution. There are three such input pins available on the DB25 connector of the parallel port, which will be called limit-inputs or *limits*:

LIMIT1: Pin 12
LIMIT2: Pin 13
LIMIT4: Pin 15

You may ask, well where is LIMIT3? Well the limit names follow a "power of two" format so that the limit numbers may be added together to form the "limit mask" mentioned in previous chapters.

For example, if you want to make the motor step clockwise, with full steps, for 1000 steps and with an 8 millisecond delay between steps AND you want the motor to stop if the LIMIT1 input is received, you would do the following:

Work out the limit mask:

LIMIT_MASK = LIMIT1 = **1**

Therefore the command would be as follows in the single-channel mode:

`step , cw , full , 1000 , 8 ,` **1**

Or as follows in multi-channel mode:

`step , 1 , cw , full , 1000 , 8 ,` **1**

If you wanted to perform the same command but have the motor to stop if either the LIMIT1 or LIMIT4 signals received, then you would do the following:

Work out the limit mask:

LIMIT_MASK = LIMIT1 + LIMIT4 = 1 + 4 = **5**

Therefore the command would be:

`step , 1 , cw , full , 1000 , 8 ,` **5**

If you wanted to turn forever until a limit-input signal was received on any of the limits, you would do the following:

Work out the limit mask:

LIMIT_MASK = LIMIT1 + LIMIT2 + LIMIT4 = 1 + 2 + 4 = **7**

Therefore the command would be:

`step , 1 , cw , full ,` **inf** `, 8 ,` **7**

If you wanted to wait forever until a limit-input signal was given on LIMIT2 you would do the following:

Work out the limit mask:

LIMIT_MASK = LIMIT2 = **2**

Therefore the command would be:

`wait ,` **inf** `,` **2**

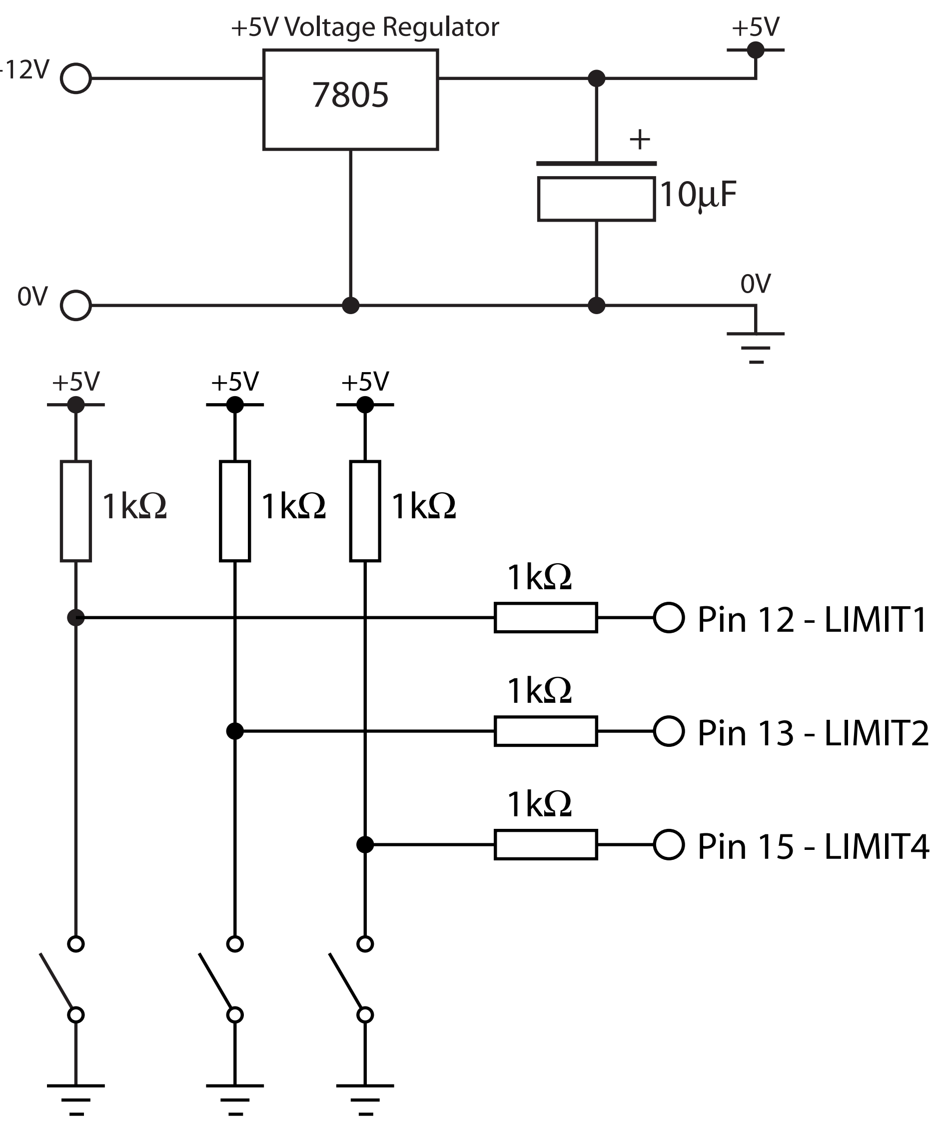
As was mentioned above the input pins for the limit-inputs are pins 12, 13 and 15 on the DB25 parallel connector. To provide a signal to a given limit-input pin all the user has to do is apply +5V to the relevant pin and one of the ground pins (Pins 18-25).

IT IS STRONGLY RECOMMENDED THAT YOU APPLY THE +5VDC VOLTAGE THROUGH A RESISTOR OF AROUND 1KΩ. THIS IS FORM OF DAMAGE LIMITATION, IN THE CASE OF EITHER CONNECTIONS SHORTING TOGETHER OR THE VOLTAGE BEING APPLIED TO THE WRONG PINS OF THE PARALLEL PORT.

Applying the input voltage through a resistor will protect the parallel port to a given extent, however it is just as important to pay close attention to construction of such an interface, as it is very easy to permanently damage your parallel port – You have been warned!

The stepper motor kit runs on a +12V power supply, therefore it will be necessary to either have a separate voltage source or to derive the +5V source from the +12V source through the use of a voltage regulator etc.

The following circuit diagram is an example of how the reader may connect three switches to the limit-input pins of the parallel port, using a +5V voltage regulator to power the circuit from the +12V supply of the stepper motor kit:


+5V Voltage Regulator
+12V
7805
+5V
+
10µF
0V
0V
+5V
+5V
+5V
1kΩ
1kΩ
1kΩ
1kΩ
Pin 12 - LIMIT1
1kΩ
Pin 13 - LIMIT2
1kΩ
Pin 15 - LIMIT4

The Different Delay Methods

stepperClass relies on pauseClass as a method of pausing in between consecutive steps. pauseClass has three different methods in which to perform the delay operation.

Readers may already know of the WIN32 API function Sleep(). When the Sleep() function is called, the operating-system (OS) stops executing the current program (or thread), and swaps to the next thread in the scheduling queue. In theory the execution of the program should start again after the requested "Sleep" time has elapsed, however this is not always the case. If other applications are also running on the computer, the operating system may not hand back to our program when we had hoped for. Tests [1] show that Sleep has an accuracy of at best around 1 millisecond.

Another method of delaying a program is to query a time source repeatedly until the required delay has elapsed. The benefit of this method is that the program does not have to rely on the OS to restart execution after swapping to a different thread, but it is very processor intensive method.

The most accurate time source on a Pentium computer is the high-performance counter. On a Pentium III 733MHz, this counter has a frequency of 3.58MHz, a resolution of around 280 nanoseconds. Tests [1] show that this counter can produce an accurate timing method with an accuracy of around 2 microseconds for a 733MHz Pentium III.

To provide different methods of delay pauseClass has these five delay modes:

- **Always Sleep**
  This function always uses the Sleep() function
- **Always Sleep(0)**
  This function waits in a Sleep(0) loop, until the required time has elapsed. Therefore it is possible to exit the delay loop before the time has elapsed.
- **Sleep Big – Count Small**
  This function uses the Sleep() function when the required delay time is over a given threshold, and the high-performance counter querying loop for delays smaller than the threshold. The threshold is known as the "Minimum Sleep Time"
- **Sleep(0) Big – Count Small**
  This is the same as the previous method, but it uses a Sleep(0) loop instead of the plain Sleep() function, and can therefore exit the delay loop if required.
- **Always Count**
  The function always uses the high-performance counter querying loop.

The default method is the "Sleep(0) Big – Count Small" function, as it has the best trade off between accuracy and processor load minimisation.

The Source Code

The guistep, cmdstep and stepperClass.dll programs were written in C++ using Microsoft Visual C++ 6. To use the source code as it is at the moment you will also need to use Microsoft Visual C++ 6 or higher.

The programs were going to be written using Borland C++ Builder but unfortunately that package had removed support for the legacy IO functions outportb() and inportb() needed for easy low-level access to the parallel port in Windows 9x.

Opening the step.dsw workspace file in the src directory loads the three following projects: cmdstep, guistep and stepperClass

The reason that the program was written in three separate parts is that both cmdstep and guistep make use of the same basic functionality, i.e. controlling a stepper motor. So rather than having the same code being duplicated in both executables, a common library file stepperClass.dll is used. The added benefit of this arrangement is that the library can also be incorporated into other applications.

'stepperClass.dll'

The source code making up the shared stepperClass.dll library is as follows:

- **pportClass.cpp**
  This class provides low-level access to the PC's IO ports. It detects which version of Windows the code is being executed on and either uses the porttalk driver if used under Windows NT4, 2000 or XP, or accesses the port directly under Windows 95, 98 or Me using _outp() and _inp()
- **pauseClass.cpp**
  This class provides highly accurate delay methods, using the high-performance counter built into Pentium processors.
- **stepperClass.cpp**
  This class depends on both the pportClass and pauseClass. It has methods to parse a buffer containing commands in string format. It can then execute the commands using pportClass to write to the parallel port, and pauseClass to pause between steps.
- **multiChannelStepperClass.cpp**
  This is the multi-channel stepper motor control class. It has the same basic functionality as stepperClass, apart from the different command set described earlier.
- **multiChannelCommandClass.cpp**
  This class is used to create command objects with multiple step commands. These are then used multiChannelStepperClass to perform steps on more than one motor at a time.

All five classes are exported to the library, for more information read the Using the stepperClass.dll Library Chapter.

'cmdstep.exe'

The source code for the cmdstep program consists of one main file:

- **cmdstep-main.cpp**
  This is a simple "wrapper" program for the stepperClass library. It parses options given to it on the command line, then executes them using stepperClass as required.

'guistep.exe'

The source code making up the guistep program consists of:

- **configClass.cpp**
  This is a simple class to load and save a configuration structure to a binary data file.
- **guiStepperBaseClass.cpp**
  This is an abstract base class from which both guiStepperClass and guiMultiChannelStepperClass inherit from. It is used by appClass to control either one of the two decendant objects depending on whether the program is in either single or multi-channel mode.
- **guiStepperClass.cpp**
  This class inherits from the stepperClass of the stepperClass.dll library.
  It adds interruptible versions of the step and command-running functions, so that the user is able to pause and stop the command execution.
  It also adds a feedback method to the application object, to update the user interface during command execution.
- **guiMultiChannelStepperClass.cpp**
  This class inherits from the multiChannelStepperClass of the stepperClass.dll library.
  It adds interruptible versions of the step and command-running functions, so that the user is able to pause and stop the command execution.
- **appClass.cpp**
  This class encapsulates the application's global variables and functions into a single object. It includes the following functionality:
    - Initialisation of the main dialog window
    - File Open and Save methods
    - Window resizing
    - Control over the command execution
- **guistep-WinMain.cpp**
  This file contains the top-level WinMain() function, the main entry point of any Windows program. This program creates the appClass object and the main dialog window, and then waits from messages sent to the program by Windows. The main message processing function processes these messages, branching to the appClass object as required.

Using the stepperClass.dll Library in Your Programs

In order to use the stepperClass.dll library in a C++ program you will need to include the "stepperClass.hpp" header file and link your program with "stepperClass.lib". The stepperClass.dll file will then need to be in the same directory as your executable.

stepperClass.hpp also depends on both pauseClass.hpp and pportClass.hpp, so these files will also need to be in the working directory.

Another method would be to compile "stepperClass.cpp" along with your program, for which you would only need to include "stepperClass.hpp", and then add "stepperClass.cpp" to the project.

Using stepperClass

The stepperClass object's main public methods and properties are as follows:

- **Method to parse a buffer containing commands in string format:**
  ```
  bool parseCommandString(char *string, ULONG maxLen);
  ```
- **Method to add a step command to the command list:**
  ```
  void addStepCommand(UCHAR directionDistance,
                      ULONG numSteps,
                      ULONG delay_us, // Measured in microseconds
                      UCHAR limitMask,);
  ```
- **Method to add a wait command to the command list:**
  ```
  void addWaitCommand(ULONG waitTime_us, // Measured in microseconds
                      UCHAR limitMask);
  ```
- **Method to find the current number of commands in the command list:**
  ```
  ULONG getNumCommands();
  ```
- **Method to delete the command list:**
  ```
  void reset();
  ```
- **Method to run the commands in the command list:**
  ```
  ULONG doCommands();
  ```
- **Method to perform a single step command:**
  ```
  ULONG step( UCHAR directionDistance,
              ULONG numSteps,
              ULONG delay_us, // Measured in microseconds
              UCHAR limitMask);
  ```
- **Wait method, N.B. takes the time in microseconds:**
  ```
  void wait(ULONG time_us, UCHAR limitMask);
  ```
- **Flag to indicate whether a limit was found during command execution:**
  ```
  bool bFoundLimit;
  ```
- **Method to read the current value of the limits:**
  ```
  UCHAR readLimits();
  ```
- **The values of the limits from the last call to readLimits():**
  ```
  UCHAR lastLimits;
  ```
- **The last number of steps completed after performing a step command:**
  ```
  ULONG lastNumSteps;
  ```
- **The time taken to complete the last command that was executed in milliseconds:**
  ```
  double lastCommandTime;
  ```
- **Flag to indicate whether an error has occurred during any operation, including initialisation, string parsing or command execution:**

bool bError;

- **A string buffer containing any error description:**
  char lastError[];
- **The pauseClass object used to time delays between steps, this object is public in order that modifications can be made to the delay method etc.:**
  pauseClass *stepDelay;

Example Programs using stepperClass

Both of the following examples are included with the source code in the “src” directory. They are named “stepperClass_example1” and “stepperClass_example2” respectively.

A simple program using the stepperClass library could be as follows:

```
#include <stdio.h>
#include “stepperClass.hpp”

int main()
{
	stepperClass stepper;
	if (stepper.bError)
	{
		printf(“Initialisation Error: %s\n”, stepper.lastError);
		return 1;
	}
	printf(“Stepping in a circle...\n”);
	stepper.step(	STEPPER_BM_CLOCKWISE | STEPPER_BM_FULL_STEP,
			100,
			8000, // N.B. delay measured in microseconds
			0); // No Limit Mask
	printf(“Done.\nThat turn took %lf ms\n”, stepper.lastCommandTime);
	return 0;
}
```

Here is another example, this time adding a few commands to the command list, then executing them with doCommands():

```
#include <stdio.h>
#include “stepperClass.hpp”

int main()
{
	stepperClass stepper;
	if (stepper.bError)
	{
		printf(“Initialisation Error: %s\n”, stepper.lastError);
		return 1;
	}
	printf(“Adding the commands...\n”);
	for (int i = 0; i < 4; i++)
	{
		stepper.addStepCommand(STEPPER_BM_CLOCKWISE
		                       | STEPPER_BM_FULL_STEP,
		                       25,
		                       8000, // N.B. delay measured in microseconds
		                       0); // No Limit Mask
		stepper.addWaitCommand(500000, // Wait for half a second
		                       0); // No Limit Mask
	}
	printf(“Executing the commands...\n”);
	stepper.doCommands();
	printf(“Done\n”);
	return 0;
}
```

Using multiChannelStepperClass

The multiChannelStepperClass object’s main public methods are as follows:

```
// Parse Command string method:
bool parseCommandString(char *string, ULONG maxLen);
// Add step command method:
void addStepCommand(multiChannelCommandClass
                    *multiChannelCommand);
// Add wait command method:
void addWaitCommand(ULONG     waitTime_us,
                    UCHAR     limitMask);
// Step method:
void step(multiChannelCommandClass *command);
// Do commands method:
ULONG doCommands();
// Command list reset method:
void reset();
void wait(ULONG waitTime_us, UCHAR limitMask);
ULONG getNumCommands();
UCHAR readLimits();
```

Example Program using multiChannelStepperClass

The following example is included in the src directory and is called multiChannelStepperClass_example. The program makes motors one and two first turn one way, wait for a bit, and then turn back the other way.

```
#include <stdio.h>
#include "multiChannelStepperClass.hpp"

int main()
{
	multiChannelStepperClass stepper;
	stepper.delay->sleepMode = PAUSE_MODE_SLEEPBIG_COUNTSMALL;
	if (stepper.bError)
	{
		printf("Initialisation Error: %s\n", stepper.lastError);
		return 1;
	}
	// Create a new multi-channel step command, and add two motors two it:
	multiChannelCommandClass command1;
	command1.addMotor(1,
			MULTI_CHANNEL_STEPPER_BM_COUNTER_CLOCKWISE
			| MULTI_CHANNEL_STEPPER_BM_HALF_STEP,
			192,
			5000,
			0);
	command1.addMotor(2,
			MULTI_CHANNEL_STEPPER_BM_CLOCKWISE
			| MULTI_CHANNEL_STEPPER_BM_HALF_STEP,
			96,
			10000,
			0);
	// Add the command to the list:
	stepper.addStepCommand(&command1);
	// Add a wait command:
	stepper.addWaitCommand(500000, 0);
	// Execute the commands:
	stepper.doCommands();
	return 0;
}
```

Multi-Channel Address Mode

The method used to interface with the original multi-channel stepper motor kit involved one bit for every motor's clock signal, summarised as follows:

Pin 2 – Motor 1 Clock
Pin 3 – Motor 2 Clock
Pin 4 – Motor 3 Clock
Pin 5 – Motor 4 Clock
Pin 6 – Motor 5 Clock
Pin 7 – Motor 6 Clock
Pin 8 – Motor 7 Clock
Pin 9 – Motor 8 Clock
Pin 14 – Direction
Pin 16 – Distance

This limited the maximum possible number of stepper motors to eight; hence the original kit is an "eight channel stepper motor kit".

If the user would like to use more than eight motors they may want to consider the following proposed "multi-channel address mode" interface:

Pin 2 – Motor Clock
Pin 3 – Direction
Pin 4 – Distance
Pin 5 – Motor Address Bit 0
Pin 6 – Motor Address Bit 1
Pin 7 – Motor Address Bit 2
Pin 8 – Motor Address Bit 3
Pin 9 – Motor Address Bit 4

This proposed interface uses five bits as a motor address, and only a single clock bit. To use this mode you will have to drastically re-design the stepper motor controller hardware to include address decoding. Whilst doing this it would be well worth including multiplexing circuitry to feedback the limit-input signals for the given motor address. This would allow three limit-inputs per motor rather than the same three for all of the motors.

If you do plan to take on this project, the programs included with this distribution support this interface mode as follows:

- guistep – Go to the Options menu, and select Multi-Channel Address Mode
- cmdstep – Add the –a option
- multiChannelStepperClass – After creating a new multiChannelStepperClass object, set the stepMode property to: MULTI_CHANNEL_STEPPER_CLASS_MODE_USE_ADDRESSES

Troubleshooting

Try checking these things if you cannot get your stepper motor to turn using this program:

1. Check your hardware
2. Check your hardware again:
   - Is the power connected?
   - Does the motor turn in the test mode using the 555 timer on the board?
   - Have you set the board to computer control mode?
   - Have you built the kit correctly?
     - Are there any short circuits? – Test with a multimeter
     - Is each component in the right place
     - Are all the soldering joints good?
     - Have you soldered the right wires to the right pins on the DB25 parallel connector and the buffer-board? – Check Again
3. Are there any error messages displayed when you try to run the program? If so follow any advice they give you.
4. Finally if you still cannot get the motor running contact Active Surplus to see if they know of a solution.

Bibliography

Beyond Logic Website: http://www.beyondlogic.org
Has a lot of very useful information about programming using the parallel port and a lot of other information too.

MSDN Website: http://msdn.microsoft.com
A good reference to the WIN32 API, however I would recommend using http://www.google.com to search it, rather than the site's own search system. I.e. add "site:msdn.microsoft.com" to the beginning of your search query.

E. Siever et Al, **"Perl In A Nutshell, A Desktop Quick Reference,"** *O'Reilly*, 1999

References

[1]Timing in WIN32: http://www.geisswerks.com/ryan/FAQS/timing.html
A very useful guide to the different methods of timing in Windows.