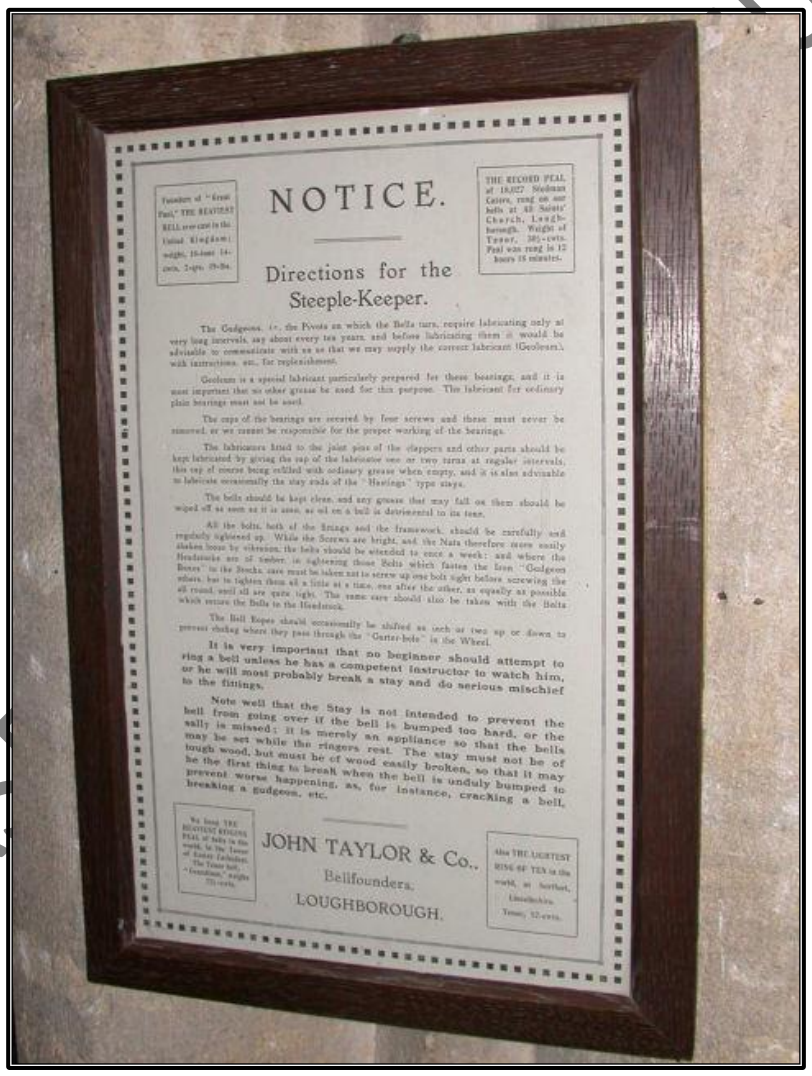
The Liverpool Ringing Simulator

Simulator Interface Software Manual



Author: Andrew Instone-Cowie

Date: 1 September 2018
Version: 1.6

Contents

Index of Figures

Index of Tables

Document History

Version	Author	Date	Changes
1.0	A J Instone-Cowie	01/08/2015	First Issue (Firmware 2.2/One Bell Firmware 1.1).
1.1	A J Instone-Cowie	25/09/2015	Tested against Abel 3.9.1.
1.2	A J Instone-Cowie	26/10/2015	Updated for Virtual Belfry 3.2.
1.3	A J Instone-Cowie	11/02/2016	Updated Firmware 2.4. Updated One Bell Firmware 1.3. Tested against Virtual Belfry 3.3.
1.4	A J Instone-Cowie	28/05/2016	Added sensor positioning bell and wheel diagrams.
1.5	A J Instone-Cowie	18/01/2017	Arduino IDE update to 1.6.12 (Boards Manager). Tested against Virtual Belfry 3.4.
1.6	A J Instone-Cowie	01/09/2018	Watermark added: Superseded by Type 2 Simulator.

Cover photograph: "Directions for the Steeple-Keeper". © 2008 Keith Edkins
[CC BY-SA 2.0 (http://creativecommons.org/licenses/by-sa/2.0)], via Wikimedia Commons

Bell and wheel diagrams on pp15-16: Adapted from original work © 2004 John Norris
[CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/)]

Licence

[1] http://creativecommons.org/licenses/by-sa/4.0/

Introduction

This Software Manual describes the design and operation of the firmware for a ringing simulator interface which allows sensors, attached to real tower bells or teaching dumb bells, to be connected to a computer simulator package such as Abel[2], Beltower[3] or Virtual Belfry[4]. It also covers the basic configuration of those simulator packages for use with the interface.

- Links are provided to the associated firmware source code and other supporting data hosted on GitHub.
- Details of the design and construction of the interface hardware are covered in the accompanying Hardware Manuals.
- Designs and construction details for a range of sensors are also covered separately.
- This is a build-it-yourself project. No pre-built hardware is available.

The firmware described in this manual is based on the approach adopted by, and seeks to maintain compatibility with, its predecessors, most notably the Bagley Multi-Bell Interface.

Licensing & Disclaimers

Documentation

All original manuals and other documentation (including PCB layout CAD files and schematics) released as part of the Liverpool Ringing Simulator project[5] are released under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA),[6] which includes the following disclaimers:

> *Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.*
>
> *To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.*

[2] http://www.abelsim.co.uk/
[3] http://www.beltower.co.uk/
[4] http://www.belfryware.com/
[5] http://www.simulators.org.uk
[6] http://creativecommons.org/licenses/by-sa/4.0/

Software

All original software (including interface firmware) released as part of the Liverpool Ringing Simulator project is released under the GNU General Public Licence (GPL), Version 3[7], and carries the following disclaimers:

> *This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.*
>
> *This program is distributed in the hope that it 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.*

The Liverpool Ringing Simulator Project

The Liverpool Ringing Simulator project is based on work undertaken at Liverpool Cathedral to provide a 12-bell simulator for demonstration and training purposes.

The tower of Liverpool Cathedral is open to the paying public during the day, except when ringing is taking place. During regular opening hours visitors are able to view the bells (which are usually left up) from a gallery above as they pass through the upper levels of the belfry, but have no access to the ringing room or belfry floor. These paying visitors are an important revenue stream for the Cathedral.

During ringing the tower is closed to visitors for health and safety reasons. Measurements of sound pressure taken in the belfry have shown that no reasonable level of mitigation would result in visitors' exposure to noise being reduced to a level which would be considered acceptable.

The Cathedral tower is also open weekly to the paying public for evening *Twilight Tours* during the summer months, and since 2012 this has been supplemented with access to the ringing room and belfry on one evening each month, under the supervision and escort of the Cathedral ringers. Statistic gathered by the Cathedral show that visitor attendances on the "with bells" nights were well above the average, and therefore the Cathedral were keen to continue to promote these evenings. The Cathedral ringers were also keen to use these events as a means of promoting the work of the Cathedral ringers and ringing generally to the public.

In 2013 a simulator sensor head was fitted to the Tenor bell, and this proved to be very successful. It was decided at the close of the 2013 season to explore the possibility of connecting more bells – possible as many as 12 – to a simulator. This was achieved and used successfully during the 2014 season.

[7] http://www.gnu.org/licenses/gpl-3.0.en.html

Project Objectives

The initial approach adopted by the original Liverpool Cathedral Simulator project was to develop and install experimental sensors and a prototype sensor aggregation hardware interface, based on the approach adopted by David Bagley's *Multi-Bell Interface* (MBI), with the following objectives:

- To build and install a 12-bell change ringing simulator at Liverpool Cathedral for use in training ringers and in demonstrating the bells and change ringing to the public.
- To build an experimental simulator sensor aggregation hardware interface, based, as far as possible, on off-the-shelf hardware, and with a minimum of custom electronics construction.
- To develop experimental simulator interface firmware, with additional measurement and debugging features, and to make this available to other experimenters.
- To maintain compatibility, as far as possible, with existing available simulator software packages, interface hardware and bell sensors.
- To document the design, construction and operation of the simulator installation for re-use by other experimenters.

The use of simulators has great potential in the training of new ringers, and therefore the Liverpool Ringing Simulator Project seeks to promote the installation and take-up of simulators as a teaching aid in other towers. The project presents the work undertaken at the Cathedral, and other towers, and makes it freely available for ringers to build and install simulators at relatively low cost.

Acknowledgements

The Liverpool Ringing Simulator project relies extensively on work already undertaken by others, notably David Bagley (developer of the Bagley MBI), Chris Hughes and Simon Feather (developers of the Abel simulator software package), Derek Ballard (developer of the Beltower simulator software package), Doug Nichols (developer of the Virtual Belfry simulator software package), and others. Their invaluable contributions are hereby acknowledged. Sources used are referenced in the footnotes throughout.

Thanks are also owed to the Ringing Masters of Liverpool Cathedral and of St George's, Isle of Man, for their willingness to be the crash test dummies of simulator design and testing.

The diagrams showing sensor positioning options are adapted from original artwork by John Norris.

Simulator Interface Design

Overview

Typical Simulator Installation

The following diagram illustrates the general arrangement of a Simulator installation using a sensor aggregation hardware interface.

Multiple Sensor Heads in the belfry are connected to a hardware Simulator Interface. A single data cable transmits the aggregated signals from the Simulator Interface to the Simulator PC, using a mutually agreed protocol. The same cable feeds power from a low voltage power supply in the ringing room back up to the Simulator Interface to power both Interface and Sensor Heads.

In the ringing room, a PC runs a Simulator Software Package which interprets the received signals and turns them into the simulated sound of bells.

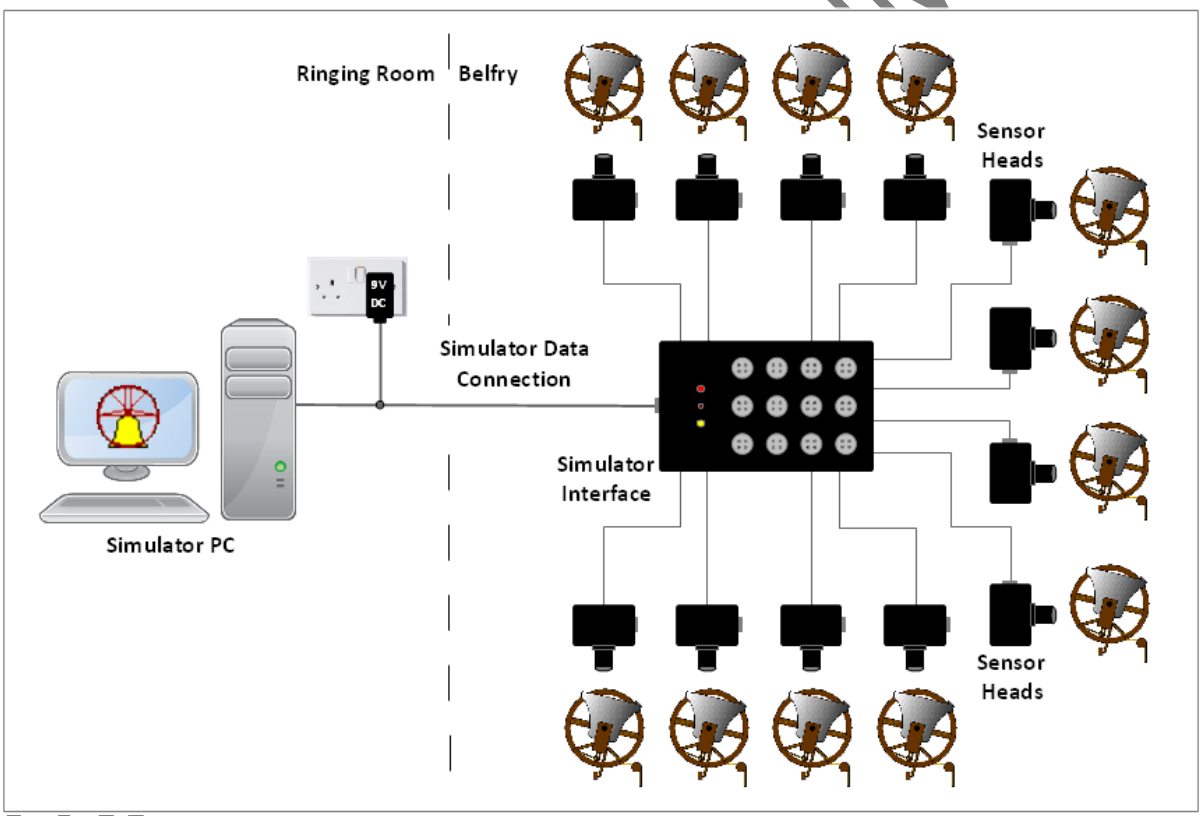


Figure 1 – Simulator General Arrangement

This manual describes the design and operation of the firmware running on the Simulator Interface hardware component of the Simulator, and the basic configuration of Simulator Software Packages for use with the Simulator Interface.

A reduced version of the same firmware is used on the integrated One Bell Simulator Interface.

Simulator Component Definitions

Within this document the following terms are used as defined in the table below:

Table 1 – Definitions of Terms

Term	Definition
Sensor Head	A Sensor Head detects the position of a real bell (or a dumb bell) at the bottom of its swing, and sends a signal at that time to the Simulator Interface. Multiple Sensor Heads may be installed, one for each bell to be connected to the Simulator.
Simulator Interface	The Simulator Interface aggregates the incoming signals from the Sensor Heads, and relays the signals to the Simulator PC (or to a Hardware Simulator) over a single consolidated data connection. The Simulator Interface applies a configurable delay to each signal so that it is received by the Simulator at the time the clapper of a real bell would have struck.
Simulator Data Connection	The Simulator Interface is connected to the Simulator over a single power/data connection. For compatibility with existing installations and software, the Liverpool Ringing Simulator project Simulator Interface uses a RS-232 serial data link running at 2400 bps. The cable carrying the Simulator/Power Data Connection is also used to supply power to the Simulator Interface and Sensor Heads from a low-voltage power supply located in the ringing room.
Simulator Interface Protocol	The Simulator Interface communicates with the Simulator over the Data Connection using a mutually agreed logical protocol. For compatibility with existing installations and software, the Liverpool Ringing Simulator project Simulator Interface implements the core Bagley MBI Protocol, with additional extensions for debugging and configuration purposes. These extensions are described in this manual.
Simulator	Collectively, the combination of a Simulator Software Package running on a Simulator PC; or alternatively a dedicated Hardware Simulator device fulfilling the same function.
Simulator Software Package	The Simulator Software Package produces the sound of simulated change ringing through loudspeakers and/or headphones, in response to input signals from the Simulator Interface and its own internal instructions, method definitions, etc. The Liverpool Ringing Simulator project Simulator Interface has been tested with a number of popular Simulator Software Packages. These are listed below, and details of configuration can be found in this manual.
Simulator PC	The Simulator PC is a general purpose computing platform which runs a Simulator Software Package, typically an Intel-compatible PC running a version of Microsoft Windows. At least one RS-232 serial port (or a USB to RS-232 adapter) is required for the Simulator Data Connection, and loudspeakers or headphones.
Hardware Simulator	A Hardware Simulator is a dedicated stand-alone proprietary device which fulfils both the hardware and software functions of a Simulator.

	An example of a Hardware Simulator is the Bagley *Ringleader* Simulator[8]. This device has not been available for new supply for some years.

Bell Sensing Method

Sensor Types

Over the years a number of different approaches have been used for detecting the position of a tied or dumb bell, and translating that position into a signal for use by a Simulator. These are discussed in detail in the accompanying Hardware Manual, and designs for sensors are also available separately.

- For the purposes of this Software Manual, the essential feature of the sensor is that it should generate a signal pulse with a clean change of logic state when the bell is at the bottom dead centre of its swing.
- The Liverpool Ringing Simulator project adopts the convention that the output of a sensor should normally be logic high (~+5V), falling to logic low (~0V) for the duration of the pulse. This approach retains compatibility with other similar systems, and allows the use of the microcontroller's internal pullup resistors, reducing the overall component count.
- The minimum practical pulse duration is a few milliseconds, but should be long enough to allow for the positive detection of good signals and the rejection of noise. Practical implementations have shown actual pulse durations in the range 5ms-20ms.
- The design of the Simulator Interface firmware is such that the maximum pulse duration may be several hundred milliseconds. The state of the output from the sensor is ignored from the time a good signal is confirmed to the point of the simulated strike (i.e. when the delay timer expires and the Simulator Interface sends a signal to the Simulator PC).
- The sensor adopted as the current standard sensor for the project uses a low cost commercially available modulating infra-red detector. Other sensor types are possible.

Single vs Dual Trigger

There are generally two approaches used by Simulators when detecting signals from Sensor Heads:

- Dual triggers mounted on the shroud of the wheel of each bell, positioned at the handstroke and backstroke strike points.
- A single trigger on the shroud of the wheel, which triggers as the bell passes through the bottom dead centre (BDC) of its swing.

Dual Triggers

The dual trigger approach uses two triggers (for example, optical reflectors) on the shroud of the wheel of each bell. One is positioned to trigger at the moment that the bell would strike at handstroke, the other at the moment that the bell would strike at backstroke. The Simulator triggers the simulated sound of the bell as soon as the signal pulse is received (triggered by the reflector travelling past the sensor as the bell is on its way up to the balance), and is configured to ignore the next signal pulse (as the first reflector travels back past the sensor as the bell is on its way down), and so on.

[8] http://www.ringing.demon.co.uk/ringleader/ringleader.htm

This approach is illustrated in the following diagrams. In Figure 2 the bell is rising towards the handstroke position, and is passing through the backstroke strike point. The backstroke trigger passes the sensor and generates a signal to the Simulator. Figure 3 shows the opposite stroke as the bell rises towards the backstroke position and passes through the handstroke strike point.

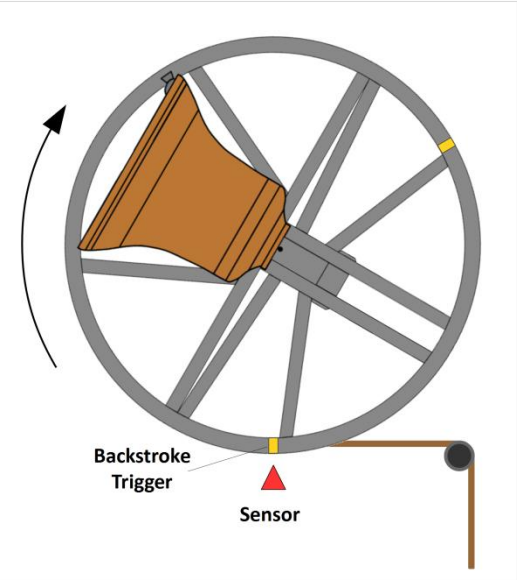


Figure 2 – Dual Triggers – Backstroke Strike Point

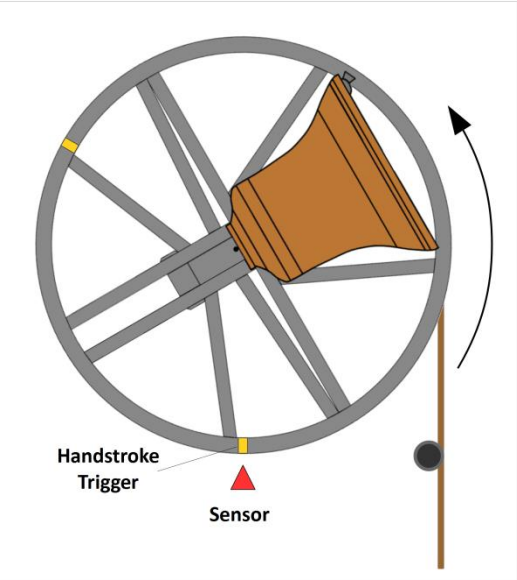


Figure 3 – Dual Triggers – Handstroke Strike Point

The sensor is shown at the BDC position only for clarity. The sensor may actually be located anywhere around the wheel, as long as the geometrical relationship between the sensor and the two triggers is maintained.

This approach has the capacity to provide a very accurate representation of the striking of a bell, but equally requires very accurate placement of the triggers to match the strike points of the bell in motion. This may be difficult and time-consuming to determine.

Single Trigger

The alternative approach is to use a single trigger, which triggers as the bell passes through the bottom dead centre of its swing. A delay is then applied (either in the Simulator Interface or the Simulator Software Package) before the Simulator triggers the simulated sound of the bell.

This approach is illustrated in the following diagrams. In Figure 4 the bell is rising towards the handstroke position. The trigger passed the sensor at BDC, and a delay is applied so that the signal to the Simulator is generated later, as the bell passes through the backstroke strike point. Figure 5 shows the opposite stroke as the bell rises towards the backstroke position and passes through the handstroke strike point.

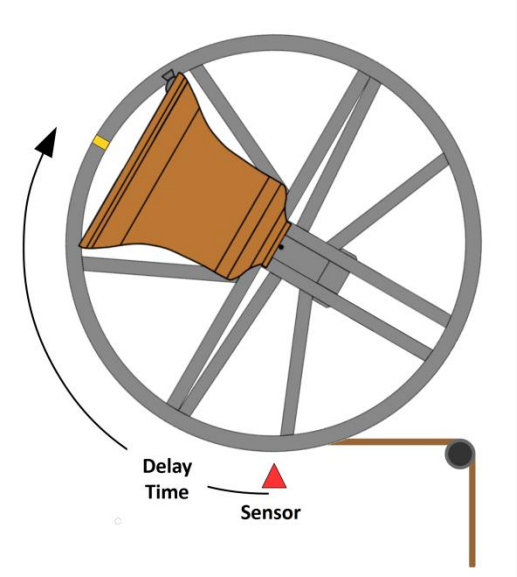


Figure 4 – Single Trigger – Backstroke Strike Point

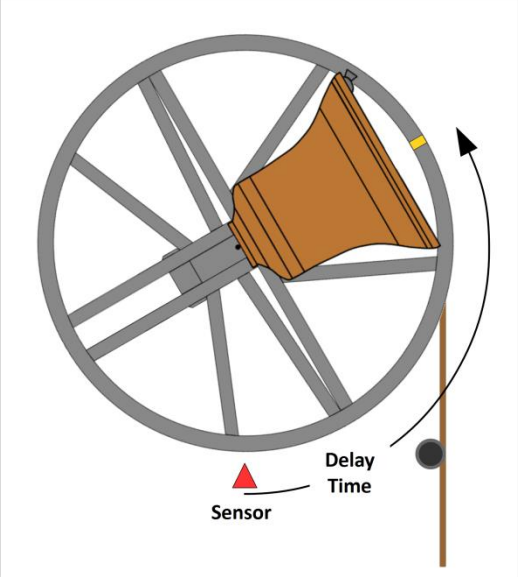


Figure 5 – Single Trigger – Handstroke Strike Point

The sensor is again shown at BDC only for clarity. The sensor may be located anywhere around the wheel, as long as the geometrical relationship between the sensor and the trigger is maintained, the trigger being positioned directly opposite the sensor when the bell is down.

This approach is simpler to implement, and the single trigger is much easier to align accurately against the Sensor Head with the bell down. Work by John Norris[9] has shown that this approach can provide an acceptable degree of accuracy, with any errors considered too small to be detectable to the human ear.

The Liverpool Ringing Simulator project Simulator Interface adopts the single trigger approach, on the grounds of simplicity of installation, and compatibility with other similar systems.

Compatibility & Interoperability

Interfaces Compatibility

The Liverpool Ringing Simulator project Simulator Interface firmware has been designed to allow operation with unmodified Simulator Software Packages which support the MBI protocol. Details of the MBI Simulator Interface Protocol were supplied independently by David Bagley[10] and Chris Hughes[11]. The operation of the MBI Simulator Interface Protocol is described in Appendix A, and

[9] http://www.jrnorris.co.uk/strike.html
[10] "I am happy to share [the protocol] with interested parties" David Bagley email, 06/09/2013.
[11] Abel Ext Connection Protocol document, Chris Hughes email, 06/09/2013.

details of the implementation of the MBI Simulator Interface Protocol commands by the Simulator Interface firmware can be found in Appendix B.

The following firmware design decisions have been made to maximise compatibility and interoperability:

- The Simulator Data Connection uses a RS-232 data link to the Simulator, running at 2400 bps, 8 data bits, 1 stop bit, no parity.
- The Sensor Head interfaces of the Simulator Interface operate at 5V DC TTL levels. Under normal (un-triggered) conditions the interface expects that the Sensor Head output pin will be HIGH (nominally +5V), and that when triggered the Sensor Head output pin will drop LOW (nominally 0V) for the duration of the trigger pulse.
- The firmware implements all documented features of the MBI protocol, and any additional feature (such as the configuration CLI[12]) have been designed to maintain compatibility. The protocol is summarised in Appendix A.

Simulator Software Compatibility

Tested Configurations

The Liverpool Ringing Simulator project Simulator Interface has been tested successfully with the current firmware version (v2.4) with the following Simulator Software Packages:

- Abel 3.9.1
- Beltower 2015 (12.29)
- Virtual Belfry[13] 3.4

In testing, the implementation of the MBI Simulator Interface Protocol by the three tested Simulator Software Packages was found not to be consistent. The differences observed are described in Appendix B.

Untested Configurations

Compatibility with the Bagley Ringleader hardware simulator has not been tested. This device has been unavailable for purchase for some time due to key components no longer being available, however, hooks have been provided in the experimental Simulator Interface firmware to allow any necessary custom behaviour to be added in future should this be necessary.

Future Simulator Support

The Simulator Interface software has also been structured to allow for the future addition of support for other simulators should this be necessary.

[12] Command Line Interface

[13] 3.1b is the minimum version of Virtual Belfry required for proper handling of interfaces of this type.

Firmware Design

Interrupt Driven vs Polling

As noted in the Hardware Manual, an interrupt-driven approach to the Simulator Interface firmware would be problematic on the Atmel ATmega328P microcontroller hardware. The firmware therefore uses a polling approach. After completing initialization and setup, the main code loop cycles round all the configured bells' inputs, processing signals from the Sensor Heads and sending serial data to the Simulator as appropriate. After all bells have been processed, the code loops round and the process starts again.

In the current version of code (v2.4), with the ATmega328P driven by the internal 8MHz oscillator, each iteration of the main polling loop takes approximately 270µs when configured for 12 bells, which has been found to be fast enough for the disadvantages of polling not to be significant. Enabling debug mode increases the loop time by a few tens of microseconds, but as debug mode is not intended for active simulator use this is also not significant.

Processing of serial input from the Simulator, in the form of MBI Protocol commands, updated timer delay values, and non-MBI CLI commands, is triggered by the standard Arduino *serialEvent()* function.

State Machine Operation

The Simulator Interface firmware maintains a set of 12 simple state machines, one for each bell. At any given time, each state machine may be in one of three possible states: *Waiting for Input, Waiting for Debounce, Waiting to Send, Sensor Disabled, or Test Mode.*

When a bell's state machine is in the *Waiting for Input* state, on each iteration of the polling loop the firmware code reads the state of the input pin connected to the associated Sensor Head, and checks for a change in state (since the last poll) from logic HIGH to LOW, which would indicate the start of a signal pulse from the Sensor Head.

- If no HIGH to LOW change in state is detected, then the Interface code does nothing other than update its record of the last observed state of the input pin, and the code loop moves on to the next bell.
- If the start of a Sensor Head pulse is detected then the code calculates the time at which the consequent signal should be sent to the Simulator (by adding the value of that bell's delay timer to the current time from the microcontroller's internal clock, in effect setting a timeout value).
- The code also calculates the time at which the input de-bounce timer should expire (by adding the value of the global de-bounce timer to the current time from the microcontroller's internal clock, also in effect setting a timeout value), and switches the state machine into the *Waiting for Debounce* state. The default value for the de-bounce timer is 4ms, but this can be configured through the Simulator Interface CLI.

When a bell's state machine is in the *Waiting for Debounce* state, on each iteration of the polling loop the firmware code reads the state of the input pin connected to the associated Sensor Head.

- If the pin state is detected as HIGH, then the input pulse previously detected has ceased. The code treats this as a "misfire" and switches the state machine back to the *Waiting for Input* state. The code loop moves on to the next bell.
- If the pin state is still LOW, the code compares the value of the de-bounce timeout with the current time from the microcontroller's internal clock.
- If the timeout value has not yet been reached, the code does nothing and moves on to the next bell. The state machine remains in the *Waiting for Debounce* state.
- If the timeout has been reached or has been passed, the code switches the state machine into the *Waiting to Send* state.

When a bell's state machine is in the *Waiting to Send* state, the firmware code compares the value of the timeout with the current time from the microcontroller's internal clock.

- If the timeout value has not yet been reached, the code does nothing and moves on to the next bell. The state machine remains in the *Waiting to Send* state.
- If the timeout has been reached or has been passed, the code sends the appropriate character over the serial interface to the Simulator PC, and switches the state machine back into the *Waiting for Input* state, and the cycle starts again.

When a bell's state machine is in the *Sensor Disabled* state, the firmware code does nothing and moves on to the next bell. *Sensor Disabled* state is enabled through the CLI or from EEPROM configuration data during setup, and the state machine remains in the *Sensor Disabled* state until reset through the CLI.

The *Test Mode* state implements a very simple test function, ignoring sensor inputs and generating a stream of rounds on the serial interface. *Test Mode* state is enabled through the CLI, and persists until the interface is reset. This facility exists primarily for testing other components of a simulator installation.

These changes of state are illustrated in the following diagram:

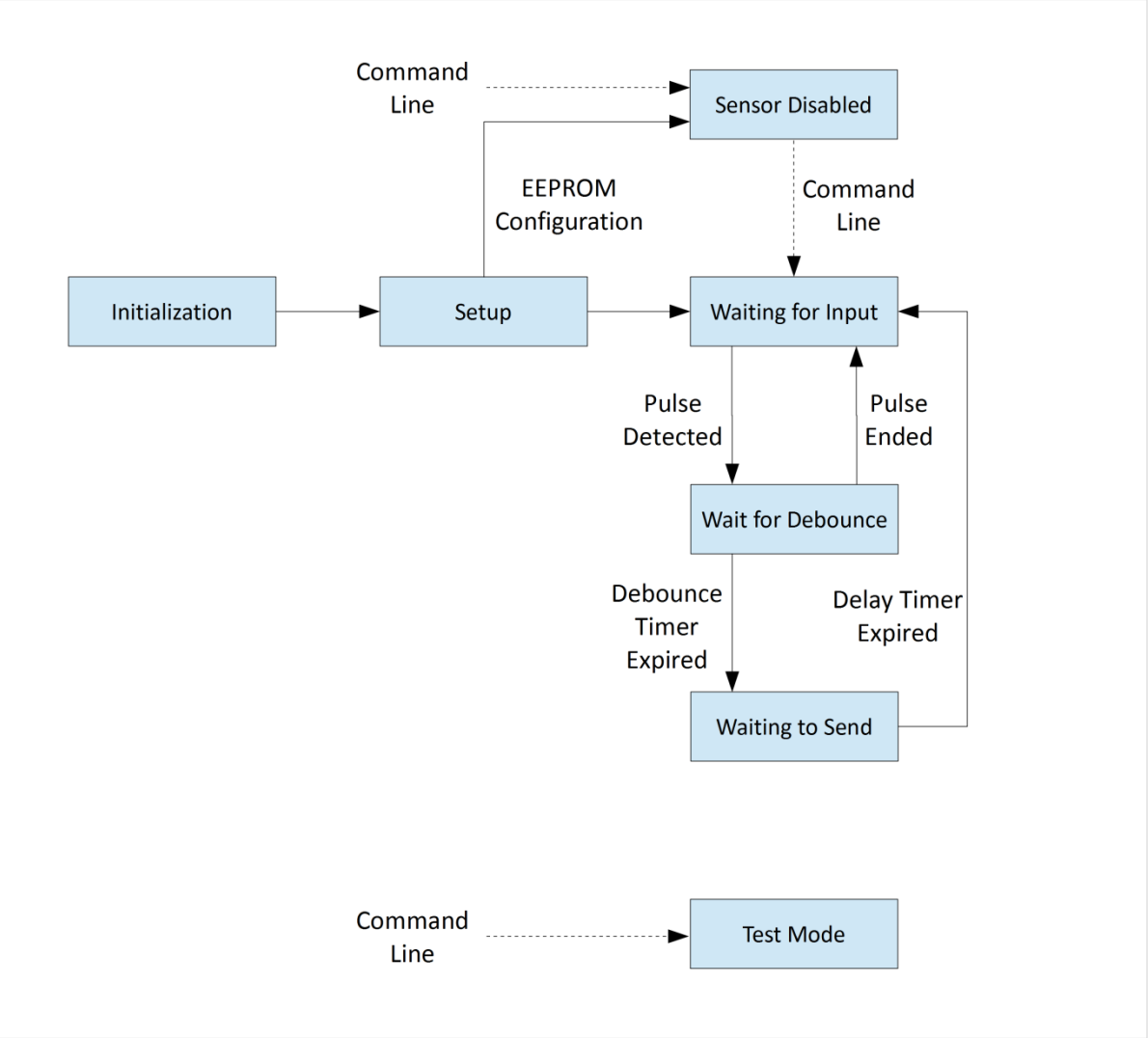


Figure 6 – State Machine Transitions Illustration

The foregoing description covers the core simulator functionality. In addition to this, debug code provides additional timing and data gathering functions which are described later.

Timer Issues

The ATmega328P microcontroller lacks a true Real Time Clock. The library inbuilt timing functions (*millis()* and *micros()*) return the number of milliseconds or microseconds respectively since the microcontroller was last powered on or the last hardware reset. All times used within the software are therefore offset times from this baseline.

The Simulator Interface firmware uses the *millis()* function for its main timing and delay calculations (*micros()* is used only by debugging code). This function returns a 32-bit unsigned long data type, and consequently the value returned will increase and then overflow back to zero after approximately 50 power-on days.

This overflow will disrupt simulator processing, as any state machines in the *Waiting to Send* state at the time of overflow will remain so for approximately 50 days. The current version of Simulator Interface software will require a power cycle or reset to clear the problem.

This issue may be addressed in a future version of the Simulator Interface software. However this is considered a low-priority issue because it is unlikely that a Simulator Interface will be left powered up for such extended periods of time.

The overflow time for the *micros()* function is approximately 70 minutes. This may cause spurious values in debugging output only. The resolution of the *micros()* function has a resolution of 8µs at a clock speed of 8MHz, so debug output values in microseconds are approximations only.

Sensor De-Bouncing

The Simulator Interface firmware reacts to the start of a pulse from a Sensor Head. In practice the output from some sensor types may be noisy and actually consist of a number of very short pulses, especially at the start or end of a pulse. This is known as “bounce”, a term derived from the physical elastic bounce of a mechanical contact. This is illustrated in the following diagram:

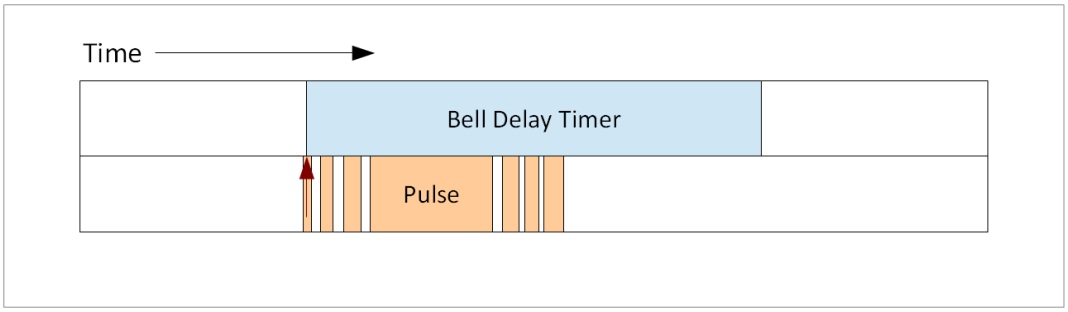


Figure 7 – Sensor De-Bounce Illustration

The current version of the Simulator Interface code reacts to the first observed transition in the output of the Sensor Head (indicated by the red arrow on the diagram).

The code then requires that the sensor signal remains stable for at least the duration of the global de-bounce timer (default 4ms, configurable). If the detected signal ceases within that interval then the signal is treated as a “misfire”, is ignored, and the code returns the state machine to the *Waiting for Input* state.

When the state machine has switched into the *Waiting to Send* state, the input pin is not read at all (other than by debug code, which includes specific detection and reporting of multiple pulses), and any sensor bounce is therefore ignored. Provided that the overall sensor pulse is shorter than the duration of the bell’s delay timer, this effectively de-bounces the input. Typically the input pulse from an optical sensor will be a few milliseconds in duration, and the delay time several hundred milliseconds, so this is not a practical issue.

In testing, no evidence of sensor bounce from the standard infra-red optical or magneto-resistive Sensor Heads was observed, within the resolution of the debug timing code and the microsecond timer.

Debugging & Timing Features

In addition to the core timing and delay functions described above, which are required for core Simulator operation, the Simulator Interface firmware implements further timing and data gathering for debugging and experimental purposes. This output is not suitable for sending to Simulator Software package, but may be accessed via a terminal emulator such as PuTTY.

This debugging functionality is not part of the MBI Protocol specification.

Under normal operations, each pulse from a Sensor Head causes one ASCII character (following the conventional ringing notation of 1-9, 0, E, T) to be sent via the serial port for use by the Simulator, as defined in the MBI Protocol specification.

The following debugging features are available in firmware v2.4:

Debug Mask

The *Debug Mask* allows for debugging output to be enabled on a specific subset of input channels.

- This may be used to reduce the volume of debug output generated during fault finding. By default, the mask enables debugging on all 12 channels.
- The Debug Mask setting may be saved in non-volatile EEPROM[14].
- Note that enabling Debug Mode globally suppresses normal simulator output for all channels, not just those set in the debug mask.

Debug Pulse Timer

When the *Debug Pulse Timer* flag is set, each valid input pulse will generate output on a separate line, with the following data fields:

- The channel number (1-9,0,E,T),
- The system offset time that the character was sent, in milliseconds,
- The identifier “S”,
- The duration of the (first) detected pulse, in microseconds,
- The total number of pulses detected before the expiry of the *Waiting to Send* timer.

Note that the pulse length timing code is not entirely robust: the assumption is made that the first Sensor Head pulse will be shorter than the bell delay timer. If the first pulse has not completed by the time the delay timer expires, a spuriously large result would be displayed for the pulse length, because the last recorded pulse end time (from the previous sensor pulse) will be before the current pulse start time, and the *unsigned long* data type will overflow. This is trapped by the code and displayed as a zero duration. This limitation affects debugging output only.

[14] The Debug Mask is saved to EEPROM if the Save Settings CLI command (“S”) is invoked when Debug Mode is active.

Debug Show Misfires

When the *Debug Show Misfires* flag is set, pulses shorter than the de-bounce timer will generate output on a separate line, with the following data fields:

- The channel number (1-9,0,E,T),
- The system offset time that the character was sent, in milliseconds,
- The identifier “M”,
- The duration of the (first) detected pulse, in microseconds.

Debug Show Debounce

When the *Debug Show Debounce* flag is set, pulses longer than the de-bounce timer will generate output on a separate line, with the following data fields:

- The channel number (1-9,0,E,T),
- The system offset time that the character was sent, in milliseconds,
- The identifier “D”.

Debug Show LED

When the *Debug Show LED* flag is set, the red diagnostic LED output will be enabled for all bells for which debugging is enabled. By default the red diagnostic LED indicates pulses detected on channel 1 only.

Debug Settings

Debug settings other than the Debug Mask are volatile and are not retained in EEPROM. Resetting the Simulator Interface will revert to the following default debug settings:

- Debug Mode: Off
- Debug Flags: *Debug Pulse Timer*
- Debug Mask: EEPROM Value

Note that changes to the configured Debug Flags and the Debug Mask are remembered if Debug Mode is disabled via the CLI, so debugging may be resumed easily with the same settings.

Serial Input & Command Line Interface

The Simulator Interface firmware implements a simple Command Line Interface (CLI), which may be used to configure certain aspects of the Interface, or to gather debugging and timing data. This functionality is not part of the MBI Protocol specification.

The Simulator Interface also has to handle legitimate MBI Protocol commands and delay data sent by the Simulator Software, and the software gives priority to these over CLI input. This processing is done inside the Arduino *serialEvent()* function, with the CLI itself handed off to a set of separate functions.

To achieve this, the firmware code adopts the following strategy when handling serial input.

- The first byte received is examined. If this byte is a recognised MBI Protocol command (Appendix B), it is processed and any appropriate response sent.

- If the first byte is not a recognised MBI Protocol command, it may be the first of a set of updated delay values. There is no defined command byte defined in the MBI Protocol to precede a set of delay values, so the Simulator Interface attempts to read a full set of 12 values plus the specified terminator byte within a defined timeout interval (one second in the current v2.4 software).
- If 12 values plus the correct terminator byte have been received within the timeout period, these are processed as new delay values and the delay timer values in EEPROM are updated.
- If a full set of values cannot be read, or the terminator is incorrect, the first (and possibly only) byte read is passed on to the CLI handler for further examination. Any bytes after the first are discarded.
- If the received byte is a recognised CLI command (Appendix B), it is processed and any appropriate response sent.
- If the received byte is not a recognised CLI command, it is discarded.

The yellow diagnostic LED is used to indicate that a CLI command has been received, or that an error occurred.

The use of the CLI is documented in detail in the *Operation* section of this document and in Appendix D below.

Active Channels

The MBI Protocol and the Abel and Beltower Simulator Software packages currently support a maximum of 12 bells per Simulator Interface unit. Configuration screens in both packages suggest that it would be possible to connect more than 12 bells through the use of multiple Simulator Interfaces connected to separate serial ports, but this functionality has not been tested during development.

The Liverpool Ringing Simulator project Simulator Interface has also been designed to support up to 12 bells (as originally required to allow a full demonstration at Liverpool Cathedral). However the design can be adapted for fewer bells.

During development and testing of the prototype Simulator Interfaces it was found that intermittent spurious triggering of disconnected inputs was sometimes a problem. This is described in detail later in this document.

The Simulator Interface firmware can therefore be configured to poll and process a number of active channels fewer than 12, always starting at channel 1. This behaviour is configured using CLI commands (Appendix D), accessed via a terminal emulator. The number of active channels is stored in non-volatile EEPROM.

Sensor Enable/Disable

The Simulator Interface firmware can be configured to disable any individual sensor input, for example to bypass temporarily a faulty sensor. Disabled inputs are ignored by the firmware, and generate no serial output, including debugging output. This behaviour is configured using CLI commands (Appendix D), accessed via a terminal emulator. The enabled or disabled state of each input channel is stored in non-volatile EEPROM.

Non-Volatile Timer Values

The MBI Protocol specification requires that the Simulator Interface will be programmed with a set of delay timers, one per bell, by the Simulator Software, and that these values will be retained in non-volatile memory by the Interface.

In practice, the implementation of this functionality has been found to differ between the tested versions of the Abel, Beltower and Virtual Belfry Simulator Software Packages:

- Abel masters the timer delay data in the Simulator Software Package application configuration.
    - The application sends a full set of delay data values to the Simulator Interface every time the application starts, and when values are changed within the application, overwriting any values currently stored by the Interface. Abel never reads the current values from the Interface.
    - Enabling the configuration option *Apply Delays in Software for Control Signals* causes the configured delay to be applied twice; it does not remove the delay timers from the Simulator Interface. This setting should not be used with the Liverpool Ringing Simulator interface.
    - For Abel, it is necessary that the Simulator Interface can be programmed with delay timers, but not strictly necessary that the Simulator Interface retains the delay values in non-volatile memory, because the application always sends current values when it starts.
- Beltower masters the timer delay data in the Simulator Interface.
    - Beltower does not send values to the Simulator Interface when the application starts. The application reads the current delay values from the Interface only when the application configuration screens are accessed, and saves updated values back to the Interface.
    - Unchecking the configuration option *Serial Interface: Bagley Master* prevents Beltower from reading or writing delay data from the Simulator Interface, but does not cause delays to be applied in software.
    - Beltower therefore does require that the Simulator Interface is already populated with the correct delay values when the application starts, and therefore the Simulator Interface must retain the delay values in non-volatile memory.
- Virtual Belfry masters the timer delay data in the Simulator Interface when a Sensor Group is configured explicitly for a *Bagley Multi-Bell Interface*.
    - Virtual Belfry does not send values to the Simulator Interface when the application starts. If Virtual Belfry has been configured to expect the Simulator Interface to handle delay timers, then it is required that the Simulator Interface is already populated with the correct delay values when the application starts, and therefore the Simulator Interface must retain the delay values in non-volatile memory.
    - The application never reads the current delay values from the Simulator Interface.
    - Unchecking the configuration option *All delays written to, and handled by, the interface* causes the stored interface timers to be set to zero, and the delay timers applied in the application.

- Virtual Belfry also supports a *Generic Data Interface*, which handles delay timers in only in the application. This setting should not be used with the Liverpool Ringing Simulator interface.

The Simulator Interface firmware stores the timer delay values in non-volatile microcontroller EEPROM. Additional EEPROM locations have been reserved to allow future expansion to 16 bells should this ever be a supported protocol option.

Simulator Type

The Simulator Interface firmware can be configured with the type of Simulator Software Package currently in use. The simulator type is stored in non-volatile EEPROM. This facility is provided so that the Simulator Interface may be configured to work around any quirks identified in the sensor handling of particular Simulator Software Packages.

As noted in Appendix D below, in the current firmware version (2.4), the only package-specific behaviour is to disable the yellow diagnostic LED for Beltower. This may change in future releases.

Memory Footprint

The Atmel ATmega328P has a very limited 2kByte SRAM capacity. The *MemoryFree* software library has been incorporated into the Simulator Interface code to track SRAM usage. This may be displayed via the Interface CLI. Flash (Program) Memory usage is reported by the Arduino IDE when the software sketch is compiled.

The memory footprint of the Simulator Interface firmware v2.4 on an ATmega328P is as follows:

- 15,188 / 32,768 Bytes Flash Memory
- 708 / 2,048 Bytes SRAM
- 25 / 1,024 Bytes EEPROM

The compiler *F()* macro has been used throughout the Interface code to store static text strings into program memory, thus relieving pressure on SRAM. As a result the current v2.4 of the Interface firmware can run on older ATmega128 microcontrollers with 16kBytes of Flash Memory and only 1kByte of SRAM.

Software Development Environment

The Simulator Interface firmware has been developed on the following versions of operating systems, development tools and Simulator Software packages:

- Microsoft Windows 7 SP1 32-bit
- Arduino IDE 1.6.12
- PuTTY 0.66
- Abel 3.9.1
- Beltower 2015 (12.29)
- Virtual Belfry 3.4
- PortMon 3.03

Software Availability

The current version of the firmware for the Simulator Interface is available from the project GitHub repository at https://github.com/Simulators. The firmware is released under the GNU General Public Licence (GPL), Version 3.

Metrics

The effects of three key metrics have been considered in the design of the Simulator Interface firmware. These are:

- The inter-blow interval: The rate at which the Simulator Interface is required to detect and transmit signals from all Sensor Heads to the Simulator during normal ringing.
- The inter-row interval: The rate at which the Simulator Interface is required to detect signals from any one Sensor Head during normal ringing.
- The Sensor Pulse Duration: The length of the pulse from each Sensor Head which the Simulator Interface is required reliably and unambiguously to detect.

Each of these metrics is considered further below.

Inter-Blow Interval Requirements

For any given ring of bells, the inter-blow and inter-row requirements can be estimated from the known speed of ringing.

For example, for the Liverpool Cathedral bells, taking 4h 30m as a representative speed for a peal of 5042 changes of Maximus on the (82cwt) twelve, the inter-blow interval in rounds is given by the calculation:

(270 minutes x 60 seconds) / (5042 changes x 12 bells) = 0.268s or 268ms

Taking 3h 15m as a representative speed for a peal of 5056 changes of Major on the light (24cwt) eight, the inter-blow interval in rounds is given by the calculation:

(195 minutes x 60 seconds) / (5056 changes x 8 bells) = 0.289s or 289ms

The serial protocol used by the Simulator Data Connection runs at 2400 bits per second, and sends a single ASCII character for each blow of each bell. Each character comprises 10 bits (8 data bits, plus 1 start bit and 1 stop bit), giving a maximum continuous data rate of 240 characters per second, or 4.2ms per character.

This confirms that the Simulator Data Connection has sufficient capacity, and has a resolution of approximately 1.6% of the inter-blow interval. Put another way, if the bells are fired accurately then the signals from all 12 bells can be sent over the data link to the Simulator PC within 51ms, or less than one fifth of the normal inter-blow interval. (True firing, the simultaneous striking of more than one bell, is not achievable with a Simulator Interface of this design.)

Under normal operation, where one byte at a time is sent to the Simulator, the serial print function call is non-blocking and operates asynchronously, so the main code loop is not slowed down or paused while data is sent to the Simulator[15].

(Note that generating detailed debugging output from a large number of bells can cause the serial output buffer to fill and the serial print function to block. This can cause issues with timing behaviour and debugging output.)

A more detailed analysis of the inter-blow intervals for a large sample of rings of bells may be found in Appendix C.

Inter-Row Interval Requirements

The inter-row interval follows from a similar calculation:

For the Liverpool Cathedral bells, taking 4h 30m as a typical speed for a peal of 5042 changes of Maximus, the inter-blow interval in rounds is given by the calculation:

(270 minutes x 60 seconds) / 5042 changes = 3.21s

Taking 3h 15m as a typical speed for a peal of 5056 changes of Major on the light (24cwt) eight, the inter-blow interval in rounds is given by the calculation:

(195 minutes x 60 seconds) / 5056 changes = 2.31s

As discussed below, the Simulator Interface firmware runs with a loop time of approximately 270 µs, giving a resolution of less than 0.009% of the typical 12-bell inter-row interval.

A more detailed analysis of the inter-row intervals for a large sample of rings of bells may be found in Appendix C.

Sensor Pulse Duration

For a standard optical sensor, the theoretical expected duration of the signal pulse from the Sensor Head is a function of the width of the reflector (25mm reflective tape in the current installations) and the linear speed of the rim of the bell wheel as the bell passes through the bottom dead centre trigger point.

This linear speed can be approximated as $4\pi R/T$, where R is the radius of the wheel, and T is the pendulum period of the bell for small oscillations. This calculation is based on the work of Frank King of Cambridge[16]. Applying the basic equations of motion to these parameters produces theoretical pulse durations of approximately 4ms.

However, this does not take into account the actual size of the sensor head, and the fact that the sensor head will trigger when the reflective tape only partially covers it (and will not stop triggering until after the tape has ceased fully to cover it). In addition to this, despite the shielding tube fitted to the optical detectors used, the detection zone is effectively conical, and at the wheel is therefore somewhat larger than the diameter of the detector.

[15] http://blog.arduino.cc/2011/10/04/arduino-1-0/
[16] http://www.cl.cam.ac.uk/~fhk1/Bells/equations.pdf

The following diagram illustrates this effect:

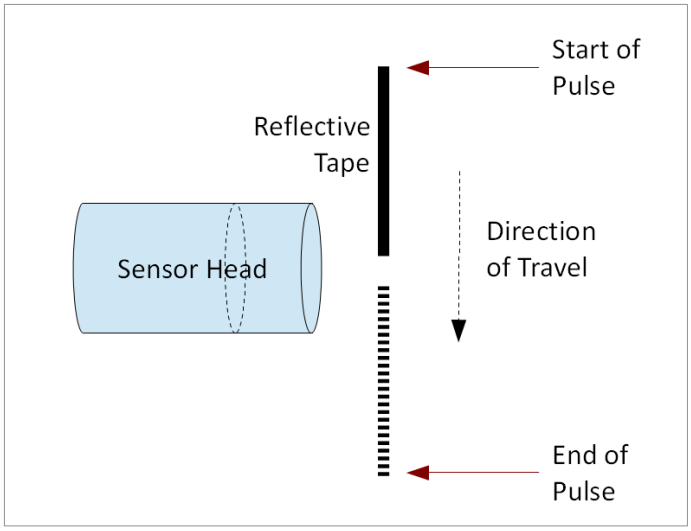


Figure 8 – Sensor Pulse Duration Illustration

In the diagram above the diameter of the Sensor Head tube and the width of the reflective tape are approximately to scale. The Sensor Head signal pulse starts before the reflective tape has fully covered the sensor aperture (solid black line, top red arrow), and ends after the reflective tape has ceased fully to cover the aperture (dashed black line, bottom red arrow). Thus the effective width of the reflective tape is increased.

An analysis of the theoretical and observed sensor pulse durations from the optical Sensor Heads of the Liverpool Cathedral Simulator installation can be found in Appendix C. The standard Sensor Head and shielding tube have an internal diameter of approximately 17mm, and the observed values of pulse duration, approximately 6ms, indicate that the effective width of the reflector is increased from the actual 25mm to approximately 40mm.

Potential Problems

The use of a polling approach to the Simulator Interface results in three potential problems:

Missed Sensor Signals

If the Simulator Interface firmware polling interval is longer the pulse emanating from a Sensor Head, then there is a risk that some pulses may be missed. If a pulse starts just after the polling loop has examined a particular input pin, the pulse will have completed before the polling loop next returns to that input pin. As a result the pulse will be missed and no signal will be sent to the Simulator.

The following diagram illustrates this problem:

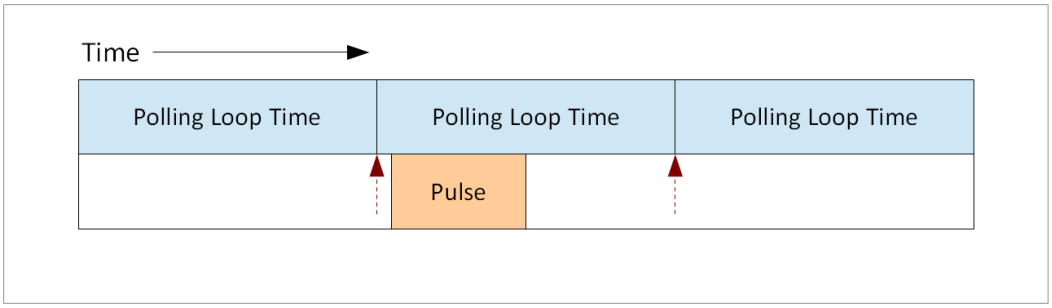


Figure 9 – Missed Sensor Signals Illustration

The red dashed arrows indicate the point at which the polling loop examines a particular input pin. During the remainder of the polling loop an incoming Sensor Head pulse begins and completes without ever being detected.

In the case of the Liverpool Cathedral installation, the duration of standard Sensor Head pulses is approximately 6ms, while the polling loop is much short in duration at approximately 270µs. Each pulse therefore lasts approximately 22 polling loop cycles, so this problem does not arise.

Duplicated Sensor Signals

If the Simulator Interface firmware polling interval is shorter than the pulse emanating from a Sensor Head (which, as noted above, is a requirement for reliable detection), then there is a probability that some pulses may be detected more than once. If the polling loop examines a particular input pin just after a pulse starts, then that pulse may still be in progress when the polling loop next returns to the same input pin. As a result the pulse will be detected more than once and duplicate signals will be sent to the Simulator.

The following diagram illustrates this problem:

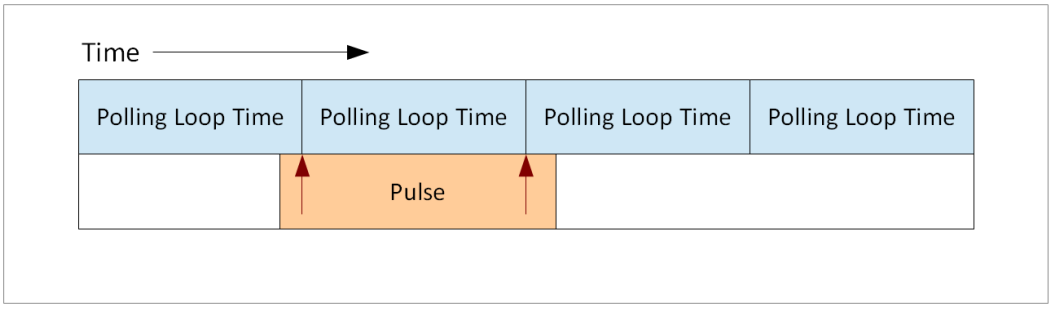


Figure 10 – Duplicated Sensor Signals Illustration

The red arrows indicate the points at which the polling loop examines a particular input pin. Because the incoming Sensor Head pulse is longer than the polling loop, it is still in progress during the next loop, and is detected again.

In the case of the experimental Simulator Interface, this problem is avoided by triggering based on detecting the transition of the input pin state from HIGH to LOW, and keeping state in the Interface

code. Therefore when the polling loop next returns to the input pin, no state transition is detected and no duplicate signals are sent.

It will be observed that decreasing the polling loop duration to avoid the Missed Sensor Signals problem results in the manifestation of the Duplicated Sensor Signals problem, and vice-versa.

Variable Odd-Struckness

A further problem arising from the use of a polling architecture in the Simulator Interface is variable odd-struckness.

- If a pulse from any particular Sensor Head starts a fraction of a second before the polling loop examines the associated input pin, the pulse will be detected almost immediately.
- If the pulse starts a fraction of a second after the polling loop examines the associated input pin, the pulse will be detected on the next iteration of the polling loop, in this case approximately 270µs later.
- There is no fixed correlation between the start time of a pulse and the current position of the polling loop in its cycle, and therefore each pulse is subject to an effectively random delay of between zero and the duration of the polling loop.
- Because the latency of the simulator system is not critical (as long as it is less than tens of milliseconds, and is constant), this will be experienced as the simulated bell striking randomly early or late by between zero and half the polling interval.

The following diagram illustrates this problem:

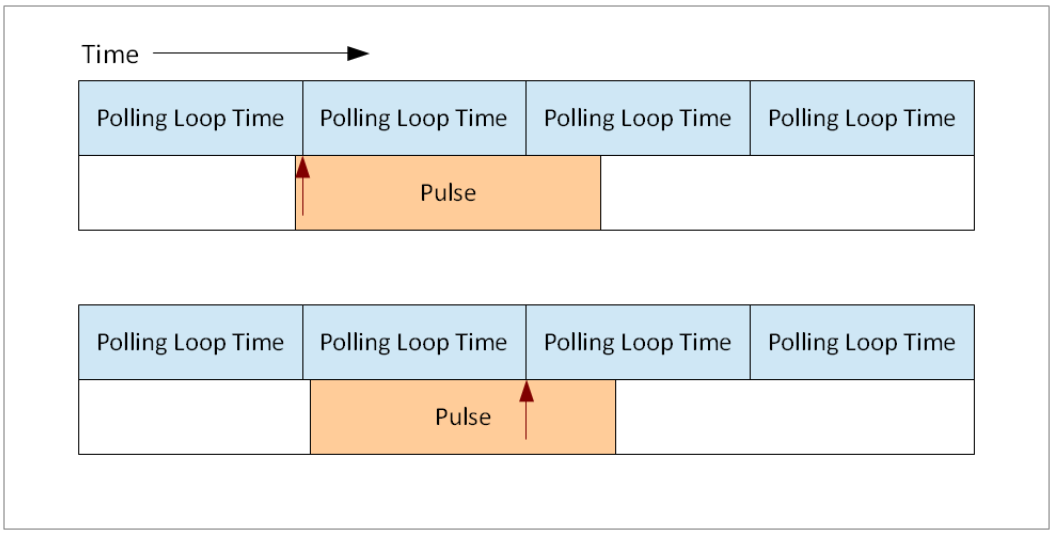


Figure 11 – Variable Odd-Struckness Illustration

The upper red arrow indicates the polling loop examining a particular input pin just after the start of an incoming Sensor Head pulse. The delay due to polling is effectively zero. The lower red arrow indicates the polling loop examining a particular input pin just before the start of an incoming Sensor Head, and detecting the pulse on the next iteration of the loop. The delay due to polling is effectively equal to the polling loop interval.

In the case of the Liverpool Ringing Simulator Interface, the polling loop interval has been measured at approximately 270µs, or 0.27ms. On the Liverpool Cathedral bells the inter-row interval (for 12-bell ringing) is approximately 3.2s. The variable odd-struckness contributed due to this problem is therefore between zero and ±0.004% of the inter-row interval and in practice this is not detectable. Even for much lighter rings of bells with much shorter inter-row intervals this may still be assumed to be negligible.

Scope does exist to reduce the length of the polling loop further. The debug and timing code contributes approximately several tens of microseconds to the loop time, and could be omitted entirely from production code.

Construction & Testing Issues

Spurious Triggering

In prototype testing, issues were observed with valid Sensor Head signals causing intermittent spurious triggering of adjacent disconnected channels. The rate of spurious triggering was <1%.

This effect was reproducible on a prototype Interface, but not on a bench-test setup, and appears to be due to the proximity of the connector wiring within the Interface enclosure, despite the use of the ATmega328P's internal pullup resistors (pin mode *INPUT_PULLUP*).

Further testing showed that this problem was not manifest when all active inputs were connected to real Sensor Heads, and a code change was made to implement the "#" and "S" CLI commands to allow unused or unconnected channels to be deactivated. This has the added benefit of reducing the firmware pooling loop interval for smaller rings of bells.

Abel USB Data Loss

During testing on a bench Arduino prototype connected to Abel directly via the Arduino USB port, it was observed that the sending of timer delay values on startup almost invariably failed. Further investigations showed that the Arduino was not receiving the expected set of 13 bytes (12 delay values plus terminator).

This problem was not observed when a conventional serial link was used on a full-scale prototype interface, and is therefore not an issue for the current implementations which uses RS-232 data connections.

This is believed to be a timing issue relating to the behaviour of the Arduino USB controller, and the result of the secondary Arduino microcontroller (an ATmega16U2, which controls the USB interfaces) going through the motions of a DTR reset in preparation for a code upload, even when a reset of the ATmega328P has been inhibited and no bootloader is installed.

Operation

This section of the Software Manual covers the initial configuration of the Simulator Interface, and the basic configuration of the tested Simulator Software Packages (Abel, Beltower and Virtual Belfry). Full details of the use of each package with external sensors should be sought from the packages' own documentation.

The instructions and screenshots are based on interface firmware version 2.4.

Simulator Interface Configuration

Firmware Upload

The firmware for the Simulator Interface Board is released under the GNU General Public Licence (GPL), Version 3, and the source code and other supporting files can be downloaded from GitHub.

- https://github.com/Simulators/simulator/tree/master/firmware/simulatorinterface

The Simulator Interface firmware is held in non-volatile flash memory on the Atmel microcontroller. It should only be necessary to re-upload the software in the event that the microcontroller is replaced, the flash memory has become corrupted, or the Simulator Interface firmware requires updating.

The firmware upload process is described in the accompanying Hardware Manual.

Connections & Power

Before using the Simulator Interface, ensure that the following connections are securely made, and that the equipment is clean and dry:

1. Connect the Sensor Heads to the 4-pin sockets on the Simulator Interface using the Sensor Head Cables. If not all channels are being connected to sensors, connect the Sensor Heads sequentially starting at Channel 1 of the Simulator Interface, and leave any unconnected channels at the top end of the range.
2. Connect the Simulator Data Connection cable to the 5-pin socket on the Simulator Interface.
3. Connect the 9-pin "D" connector on the Simulator Data Connection cable to an RS-232 serial port on the Simulator PC in the ringing room. This port is usually Serial Port 1 (*COM1*). If there is no RS-232 serial port then a USB-to-Serial adapter may be used. Notes on the configuration of USB-to-Serial adapters appear below, but it is important to note at this point that current versions of specific Simulator Software Packages require that the adapter is assigned a virtual COM port (VCP) number in a specified range[17].
4. Connect the keyboard, mouse, power cable and speakers to the Simulator PC. The following diagram shows the connections to the rear panel of a typical Simulator PC.

[17] Abel: COM1 – 8. Beltower: COM1 – 8 (32 in Beltower 2016).

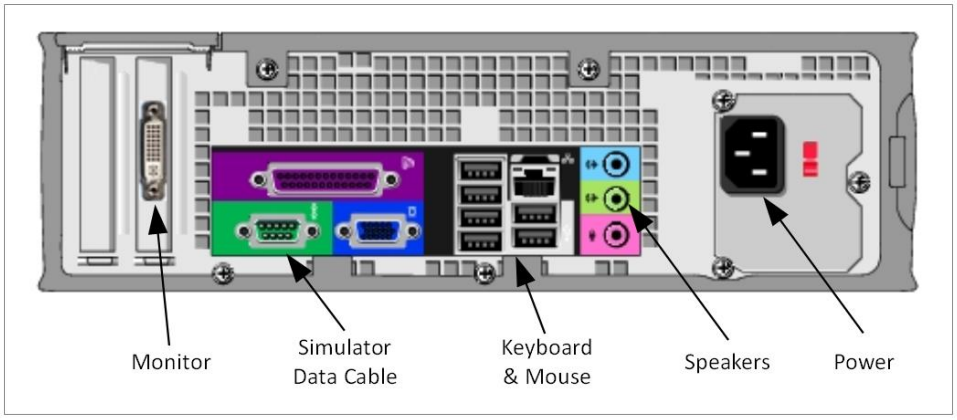


Figure 12 – Simulator PC Rear Panel Connections

5. Connect the DC power connector on the Simulator Data Connection cable to the plug-in power supply. Use the adapter tip with the red ring (for 2.5mm pin), and ensure that the tip is plugged in and oriented so that the centre pin is positive. If using the suggested power supply, the embossed "+" and the word "TIP" should be on the same side of the connector, as illustrated.

Figure 13 – Power Plug Orientation

6. Set the slide switch on the power supply to 9 volts, and plug the power supply into a 13A mains socket.
7. The yellow diagnostic LED will give a number of long and short flashes corresponding to the interface firmware version. Currently this will be two long and four short flashes, for firmware version 2.4.
8. If the yellow diagnostic LED does not flash then the microcontroller may not have the Simulator Interface firmware loaded. Instructions for uploading the firmware can be found in the accompanying Hardware Manual.
9. If the yellow LED remains lit after announcing the firmware version then the Simulator Interface may not be configured with a serial port speed of 2400 bps, or one or more sensors have been disabled.
10. Switch on the Simulator PC.

You are now ready to configure the Simulator Interface.

Interface Configuration

Configuration of the Simulator Interface should only need to be done once. All settings are retained in non-volatile EEPROM when the interface is powered off.

1. On the Simulator PC, ensure that a Simulator Software Package (e.g. Abel) is not running. Close the Simulator Software Package down if it is running.
2. Download and install a serial terminal emulator package[18]. This manual assumes the use of the Open Source PuTTY terminal emulator.
3. Start the PuTTY terminal emulator by double-clicking the PuTTY icon on the desktop.

4. Configure a Serial connection using the COM port number of the serial port (e.g. COM1), running at 2400 bps, and then click Open. You should not need to change any other settings in PuTTY.

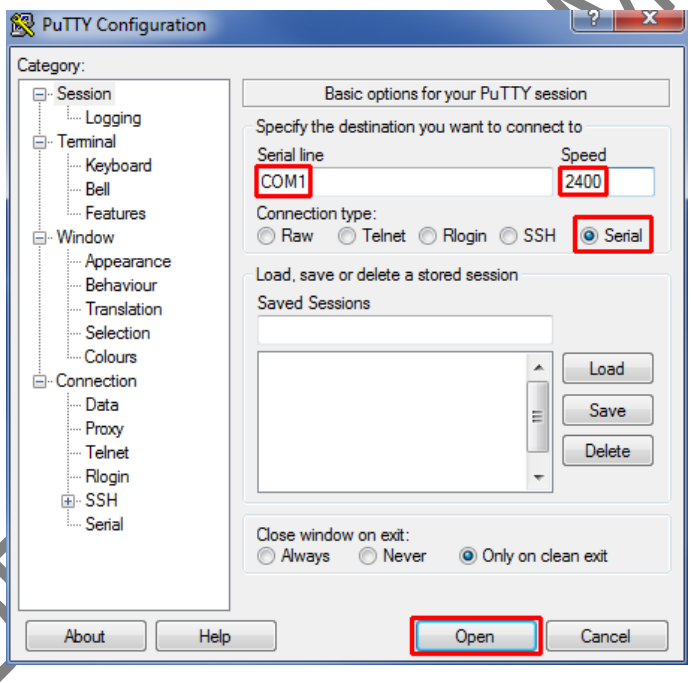


Figure 14 – PuTTY Configuration Dialogue

[18] http://www.chiark.greenend.org.uk/~sgtatham/putty/

5. Click on the PuTTY terminal window, then type "?" (question mark). There is no need to press Enter. After a short pause the Simulator Interface will respond by displaying its current settings, which may not be identical to these examples[19].

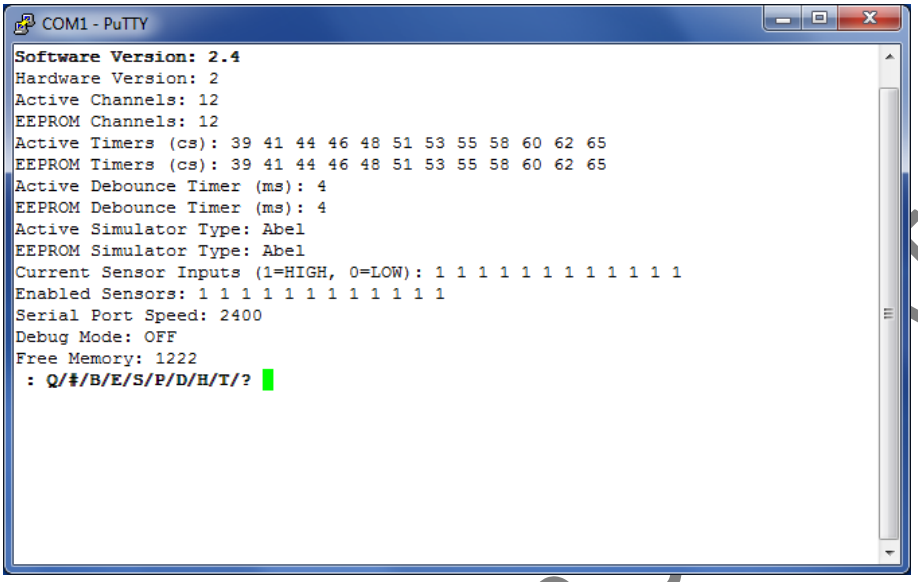


Figure 15 – CLI Display Settings Example

6. Type "H" to display the help screen for the Simulator CLI (Command Line Interface).

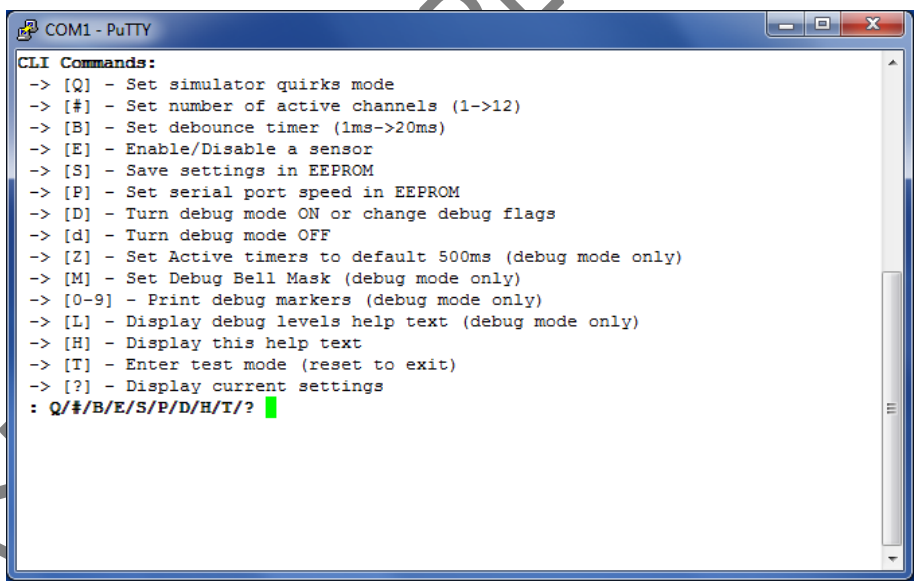


Figure 16 – CLI Display Help Text Example

[19] The default PuTTY colour scheme is white (or coloured) text on a black background. In these examples this has been reversed and reduced to black on white for better printing.

7. If the Simulator Type is not set correctly, type “Q”, followed by “A”, “B”, “R”, “V” or “X” to set it. As noted in Appendix D below, at present the only package-specific behaviour is to disable the yellow diagnostic LED for Beltower. This may change in future releases.

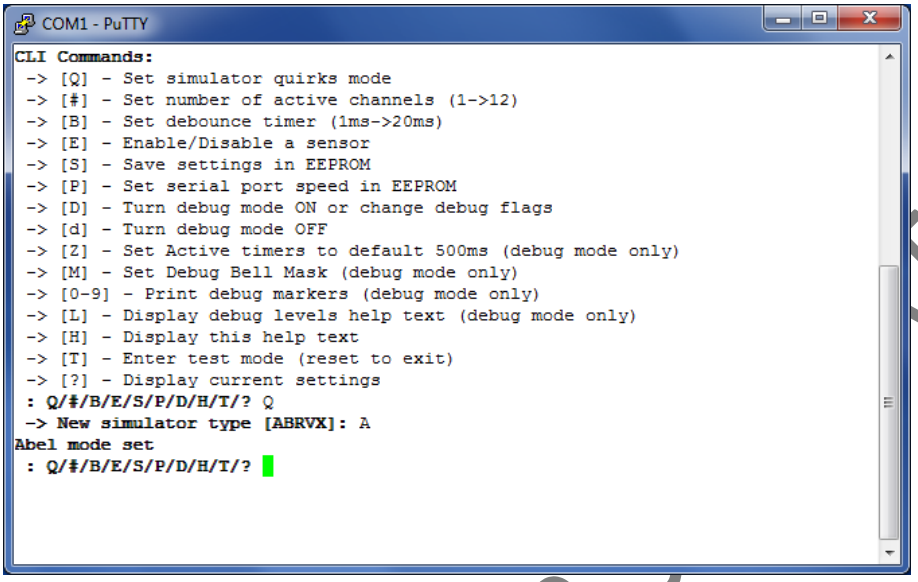


Figure 17 – CLI Set Simulator Type Example

8. If the number of Active Channels is not set to the number of Sensor Heads actually connected (which should be connected sequentially from Channel 1 upwards), type “#”, enter the number of channels in use, and press return. If necessary the de-bounce timer can also be changed in a similar way using the “B” command.

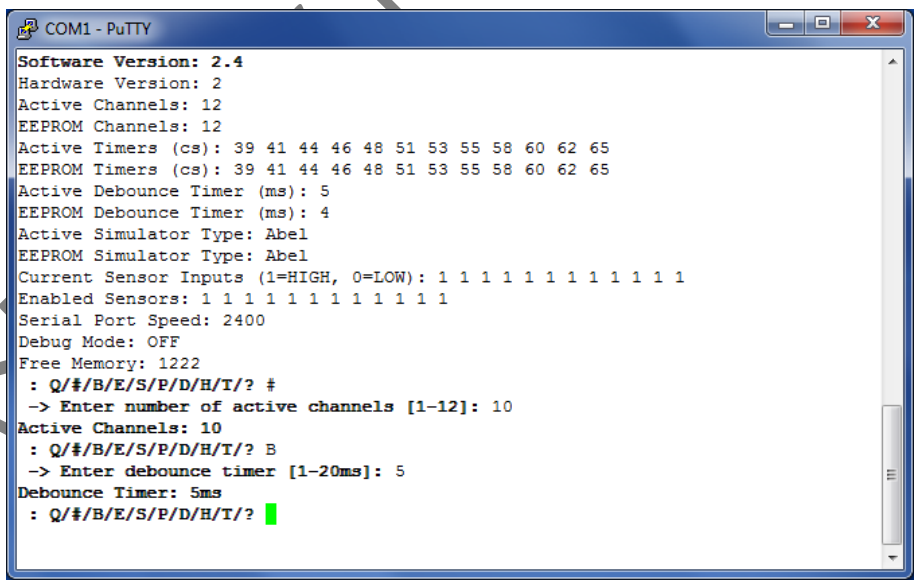


Figure 18 – CLI Set Active Channels Example

9. Type "?" to review and check the configuration. Then type "S" to save all values to non-volatile EEPROM memory. The Simulator Interface is now configured. Close the PuTTY window and start the Simulator Software Package. Note that the delay timer values are set by the Simulator Software Package, not via the Interface CLI.

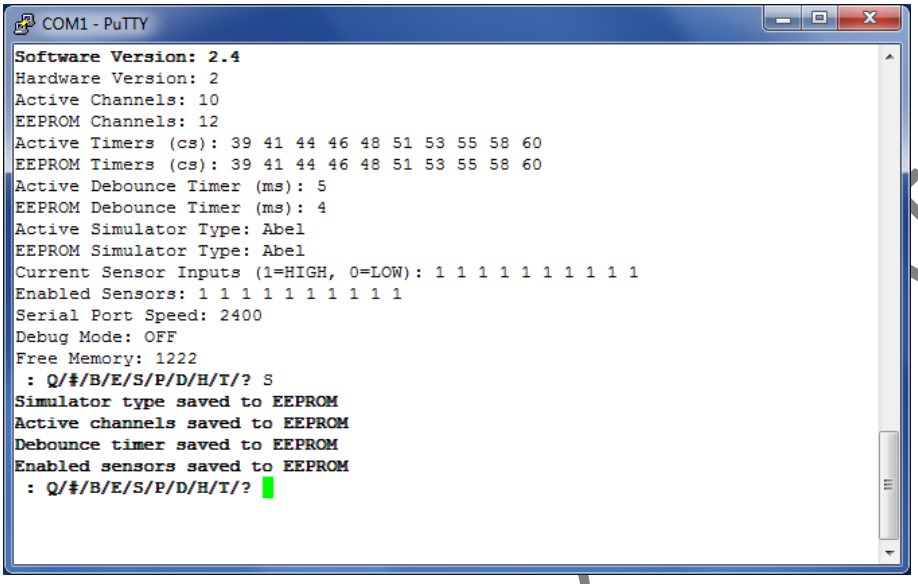


Figure 19 – CLI Save Settings Example

Debug Modes

The Simulator Interface CLI includes debug modes for fault-finding or data gathering purposes. The functions of these are described in detail in an earlier section of this manual.

1. On the Simulator PC, ensure that the Simulator Software Package is not running, and then open a PuTTY session to the Simulator CLI as described above.
2. Enable Debug Mode by typing "D".

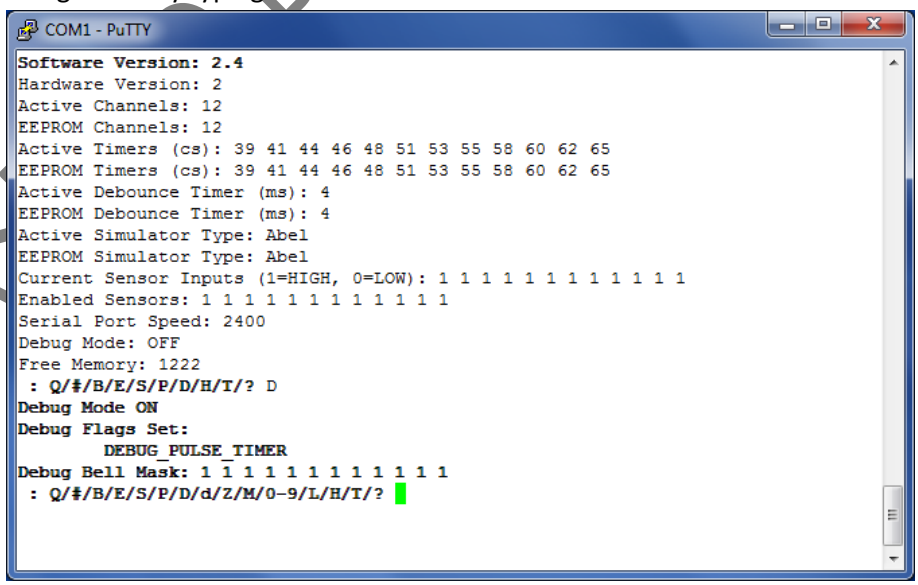


Figure 20 – CLI Debug Mode Activation

3. When Debug Mode is active, change the active Debug Flags by typing “D” again.

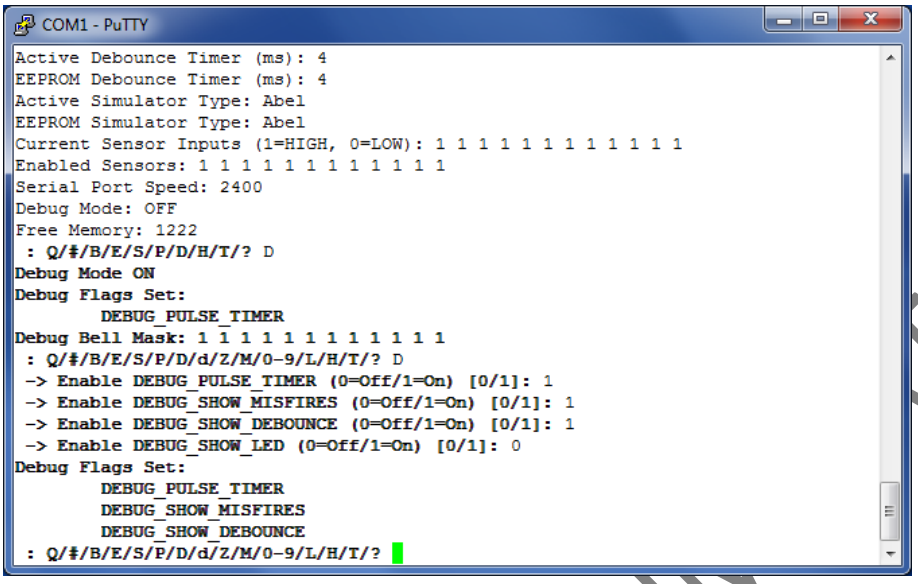


Figure 21 – CLI Debug Flag Setting

4. When Debug Mode is active, change the active Debug Mask (i.e. the bells for which debugging output is enabled) by typing “M”. Typing “S” with Debug Mode active will also save the Debug Mask to non-volatile EEPROM memory.

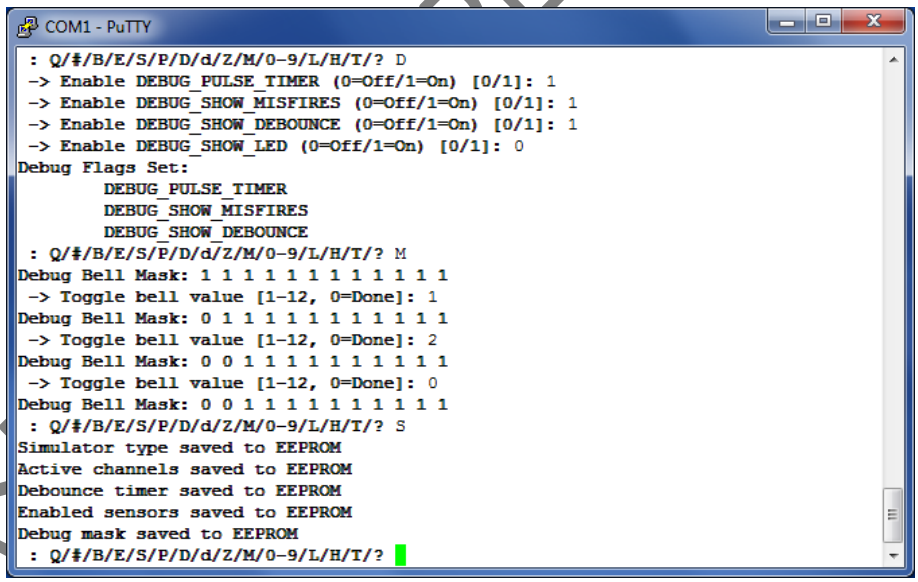


Figure 22 – CLI Debug Mask Setting

5. The Active Timers may be set temporarily to 50cs (500ms) by typing “Z” when Debug Mode is active.

6. When Debug Mode is active, additional debugging and timing information is displayed each time a Sensor Head is triggered. The fields displayed depend on which Debug Flags are active, and are summarised in the table below.

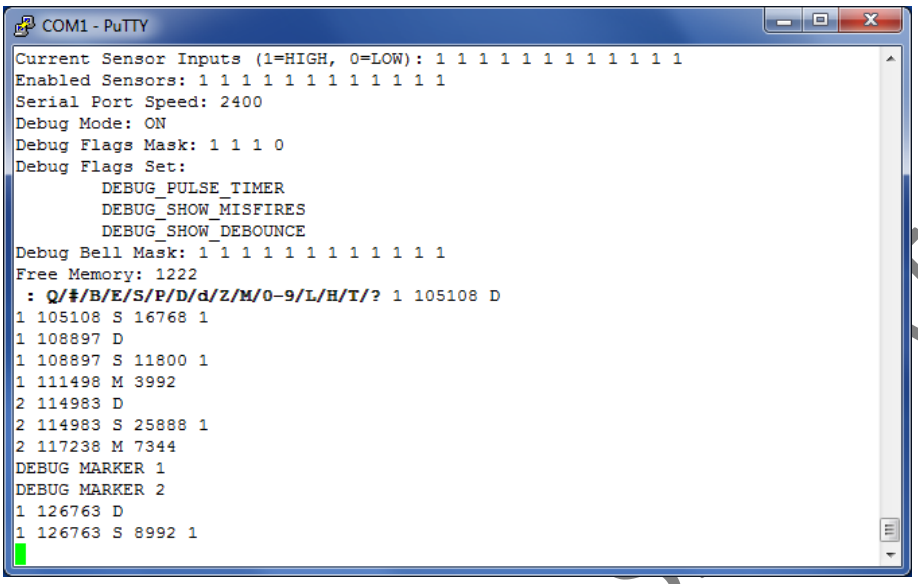


Figure 23 – CLI Debug Mode Output Example

4. The data fields displayed on each line when each Debug Flag is active are as follows:

Table 2 – Debug Output Details

Debug Mode	Output
DEBUG_PULSE_TIMER	A single line per valid pulse containing the following fields, separated by spaces: • The number of the Sensor Head channel triggered (as 1 – 9, 0, E, T) • The time at which the trigger signal was sent to the Simulator PC, in milliseconds since the last interface reset or power cycle. • The identifier "S". • The duration of the first pulse observed from the Sensor Head, in microseconds. • The total number of pulses detected before the expiry of the *Waiting to Send* timer.
DEBUG_SHOW_MISFIRES	A single line for each pulse shorter than the de-bounce timer, containing the following fields, separated by spaces: • The number of the Sensor Head channel triggered (as 1 – 9, 0, E, T) • The time at which the trigger signal would have been sent to the Simulator PC, in milliseconds since the last interface reset or power cycle. • The identifier "M". • The duration of the pulse observed from the Sensor Head, in microseconds.
DEBUG_SHOW_DEBOUNCE	A single line for each pulse longer than the de-bounce timer, containing the following fields, separated by spaces: • The number of the Sensor Head channel triggered (as 1 – 9, 0, E, T) • The time at which the trigger signal will be sent to the Simulator PC, in milliseconds since the last interface reset or power cycle. • The identifier "D".
DEBUG_SHOW_LED	Causes all sensors for which debugging is enabled to trigger the red diagnostic LED (normally only triggered by the Sensor Head on Channel 1). Useful for testing operation of individual sensors in the belfry.

This information may be useful for verifying correct Sensor Head function, and for identifying any noisy or intermittent connections. The values returned are best explained by reference to the source code for the Simulator Interface firmware.

5. It is also possible to insert debug markers into the output to identify areas of interest, by typing a number between 0 and 9. This is useful if debug output is logged to a file for later review.
6. Type “L” to display the debug help screen when any debug mode is active.

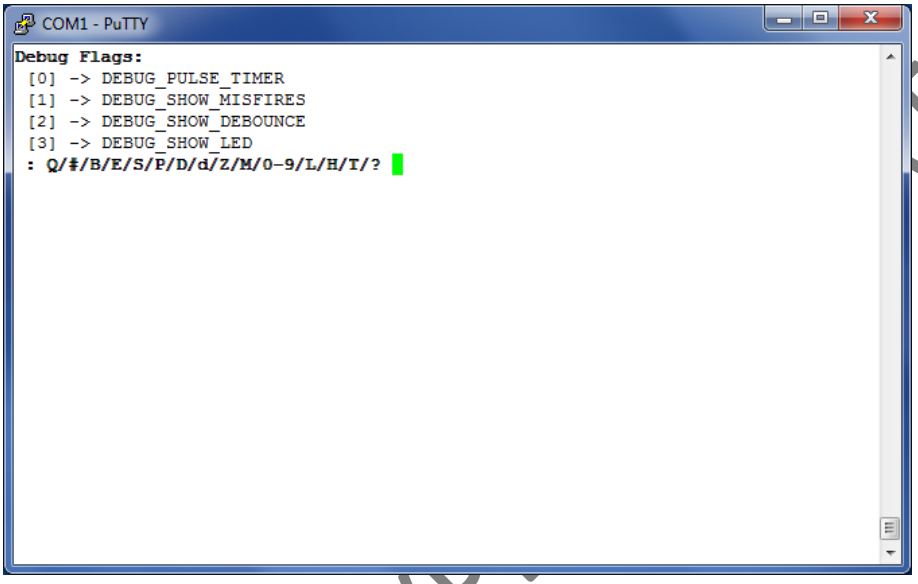


Figure 24 – CLI Debug Mode Help Text Example

7. Debug Mode must be disabled before the Simulator Interface is used with a Simulator Software Package again. Debug output is not suitable for consumption by Simulator Software Packages. Debug mode activation is not saved to non-volatile memory, so a reset or power cycle of the interface will disable it, as will typing “d”.
8. Debug mode is also automatically disabled if the Simulator Interface receives updated delay timers from the Simulator PC.

Sensor Enable/Disable

1. Toggle the Enable Mask state for each sensor (i.e. the bells for which simulator output is generated) by typing “E”, followed the number of the channel to enable or disable. Typing “S” will also save the Enable Mask to non-volatile EEPROM memory.

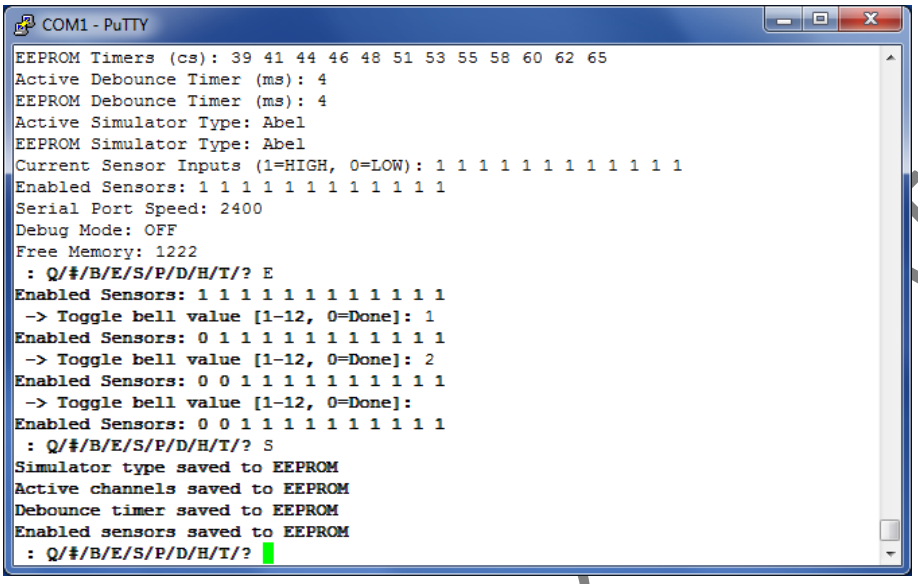


Figure 25 – CLI Sensor Enable/Disable Setting

2. A value of “1” indicates that the sensor is enabled, and “0” that it is disabled. Disabling a sensor prevents all interface output for that channel, including debugging.

USB-to-Serial Adapters

If the Simulator PC does not have an available RS-232 serial port, then a USB-to-Serial adapter may be used. These are available from a variety of manufacturers, but mainly use controller chips manufactured by Prolific and FTDI.

Adapters usually require the installation of a software driver. Drivers for adapters based on Prolific[20] and FTDI[21] controllers are available from the respective websites; the driver required is the *Virtual COM Port* or *VCP* version.

A typical USB-to-Serial adapter (in this case from Prolific) is shown in the following photograph.

Figure 26 – Example of a USB-to-Serial Adapter

Driver Installation

Follow the installation instructions supplied with the USB-to-Serial port driver software package. This example uses adapters and drivers with both Prolific and FTDI chipsets, but the approach is similar for adapters from other vendors.

[20] http://www.prolific.com.tw/US/ShowProduct.aspx?p_id=225&pcid=41
[21] http://www.ftdichip.com/Drivers/VCP.htm

1. Run the installer supplied with the driver software package. Both Prolific and FTDI drivers use a standard Windows installer.

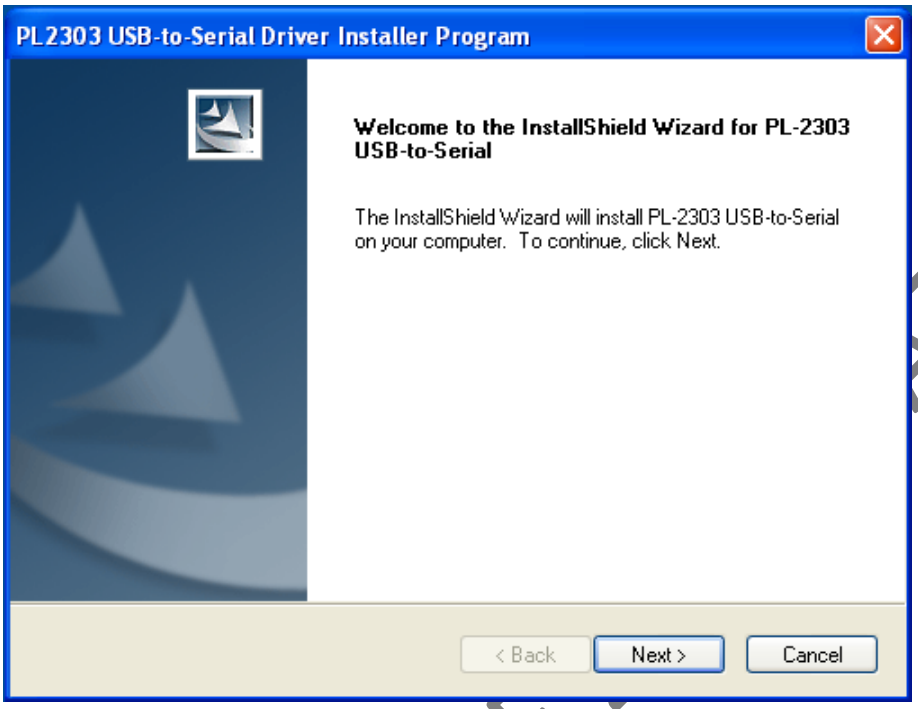


Figure 27 – Prolific Driver Installation

2. When the installation is complete, close the installer. A restart of the PC may be required.

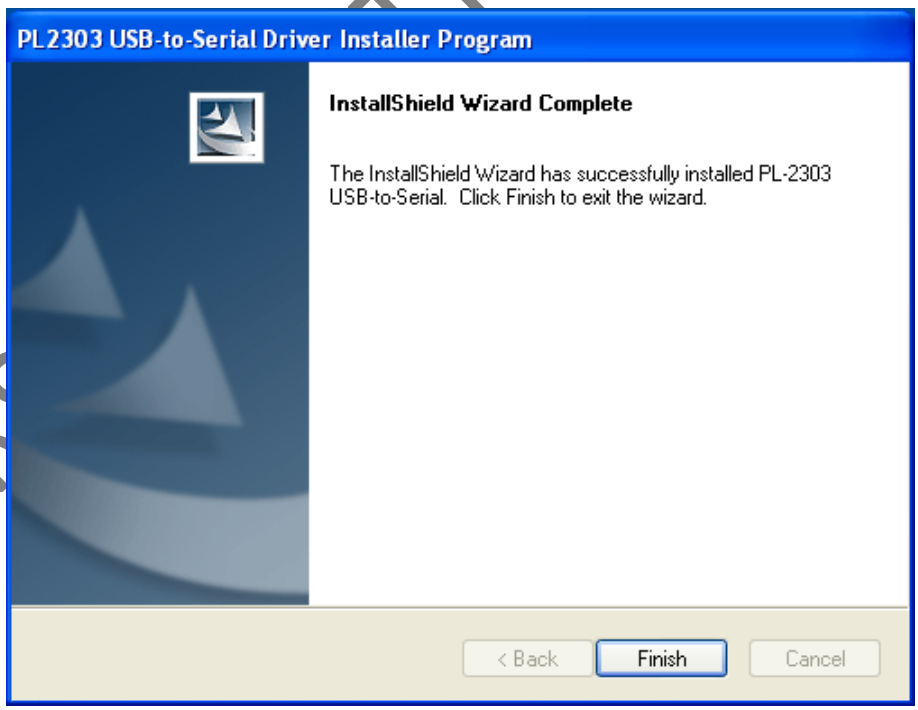


Figure 28 – Driver Installation Complete

3. Connect the USB-to-Serial adapter to a spare USB port on the Simulator PC. To ensure consistent COM port numbering, and avoid the need to reconfigure the Simulator Software Package, it is best always to use the same USB port for the adapter.

4. The Simulator PC should detect the USB-to-Serial adapter, configure the driver, and a confirmation balloon message should appear briefly above the Windows system tray.

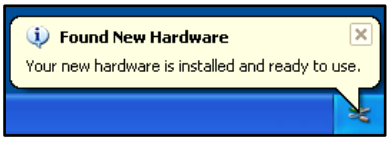


Figure 29 – Found New Hardware Message

Driver Verification

To verify that the USB-to-Serial adapter driver is correctly configured, open the Windows Device Manager and look for the adapter in the Ports section.

1. Right-click the *Computer* or *My Computer* icon on the Windows desktop, and select the *Properties* menu item. These examples are from Windows XP and Windows 7 PCs, the process and appearance may vary for other versions of Windows.

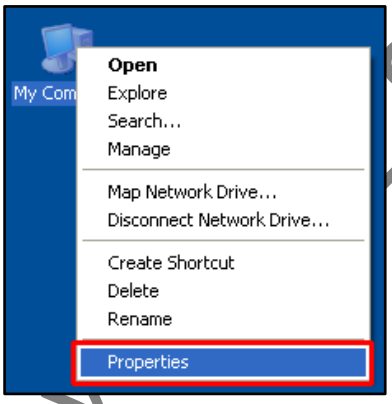


Figure 30 – My Computer Context Menu (Windows XP)

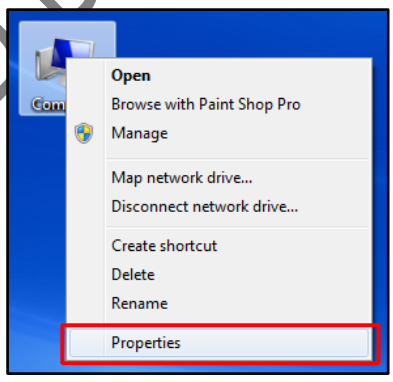


Figure 31 – Computer Context Menu (Windows 7)

2. On Windows XP, click on the *Hardware* tab, and then click the *Device Manager* button.

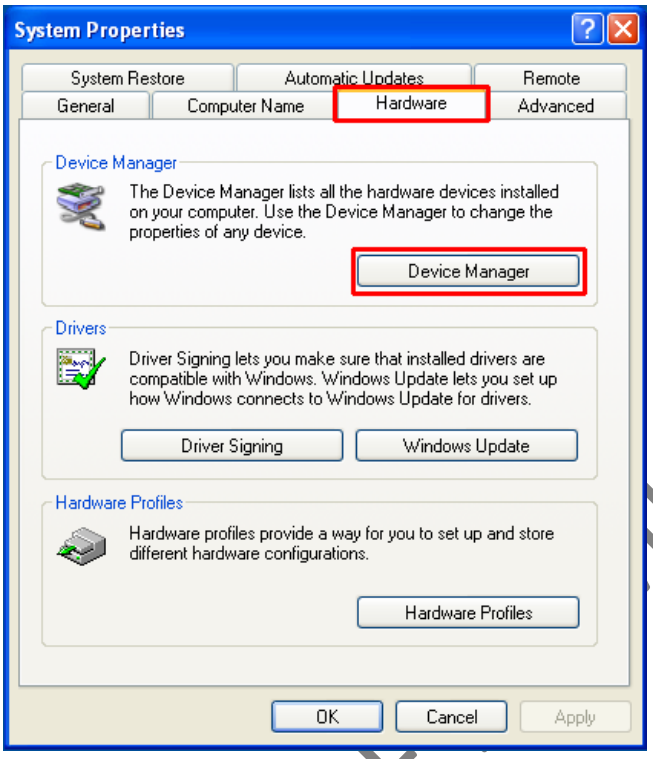


Figure 32 – System Properties Hardware Tab (Windows XP)

3. On Windows 7, click the *Device Manager* button.

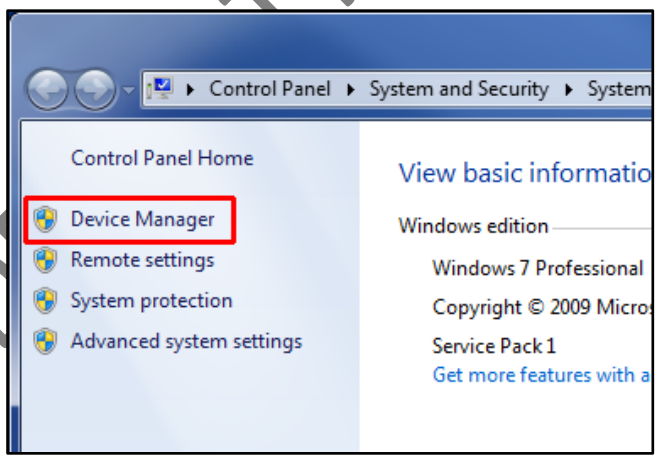


Figure 33 – System Properties Hardware Tab (Windows 7)

4. Expand the *Ports (COM & LPT)* section of the Device Manager list. The USB-to-Serial adapter should be listed, and the COM port number identified. In this example the adapter has been assigned the Virtual COM Port number *COM3*.

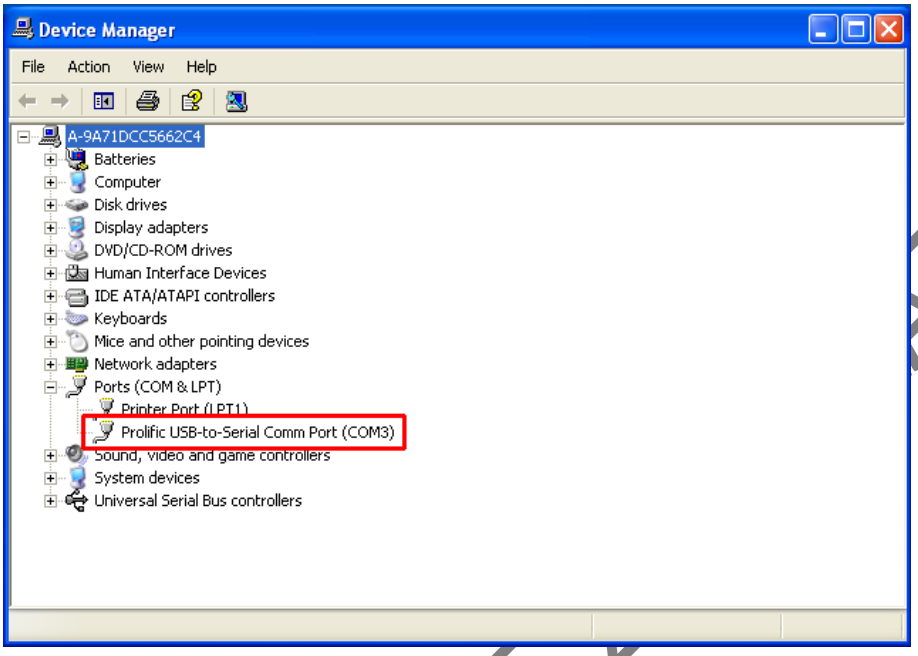


Figure 34 – Device Manager (Driver Installed)

5. If the device driver software has not been installed correctly, the adapter may be found with a warning marker in the *Other Devices* section of the device list. Install the device driver software, or refer to the documentation for the USB-to-Serial adapter.

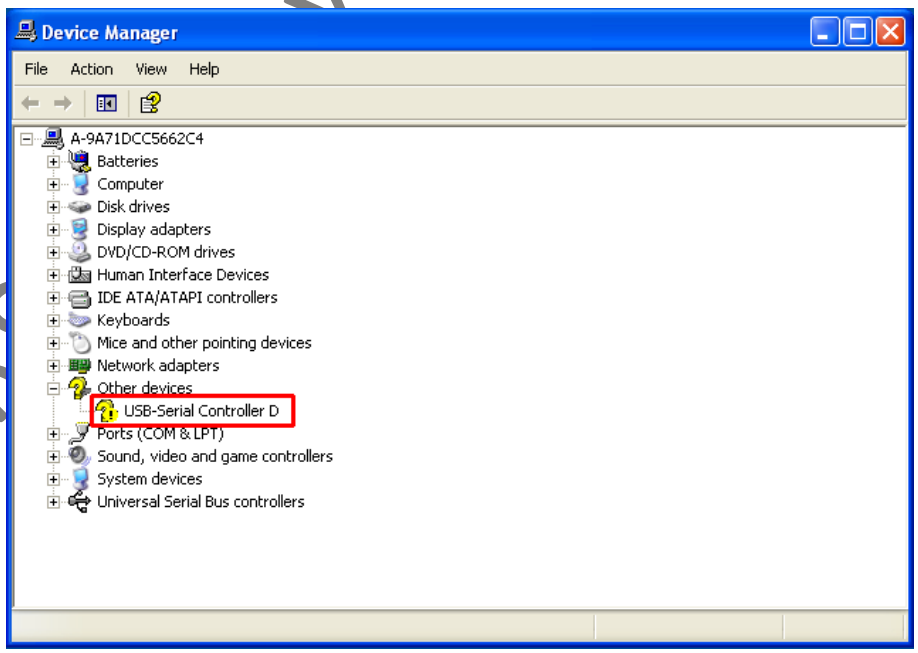


Figure 35 – Device Manager (Driver Missing)

COM Port Reconfiguration

Current versions of specific Simulator Software Packages require that the Virtual COM Port number assigned to a USB-to-Serial adapter is in a specified range[22]. If the adapter has been assigned a COM port number outside the supported range, the port must be reconfigured to lower value[23].

1. Open the Windows Device Manager as described above.
2. Expand the *Ports (COM & LPT)* section of the Device Manager list. The USB-to-Serial adapter should be listed, and the COM port number identified. In this example the Prolific adapter has been assigned the Virtual COM Port number *COM14*, which is beyond the range supported by Abel.

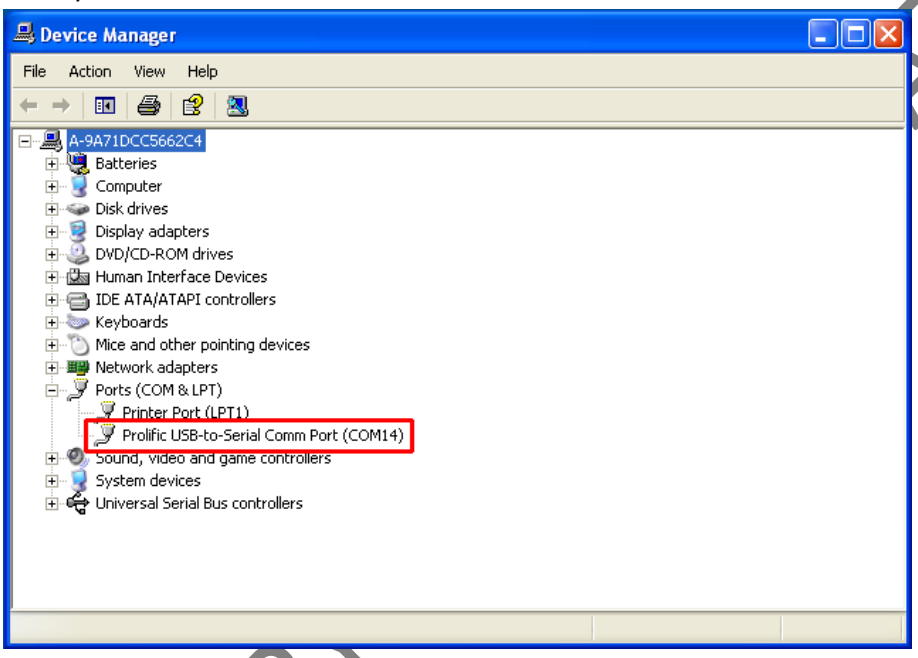


Figure 36 – Device Manager (Port COM14)

[22] Abel: COM1 – 8. Beltower: COM1 – 8 (32 in Beltower 2016).
[23] Virtual Belfry does not have this requirement.

3. Right-click the USB-to-Serial adapter entry, and select the Properties menu item.

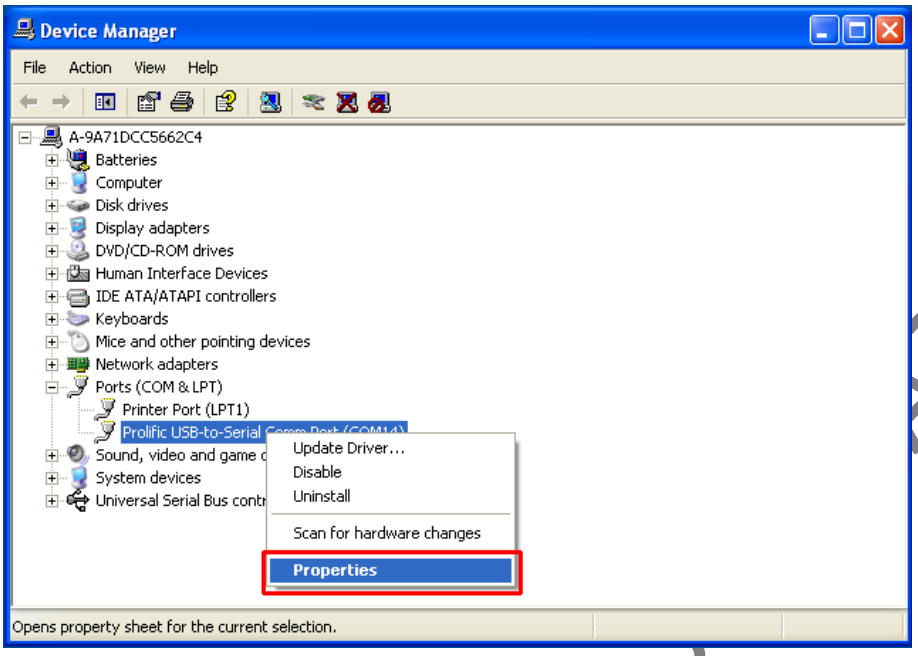


Figure 37 – Device Manager Context Menu

4. Click on the *Port Settings* tab, and then click the *Advanced...* button.

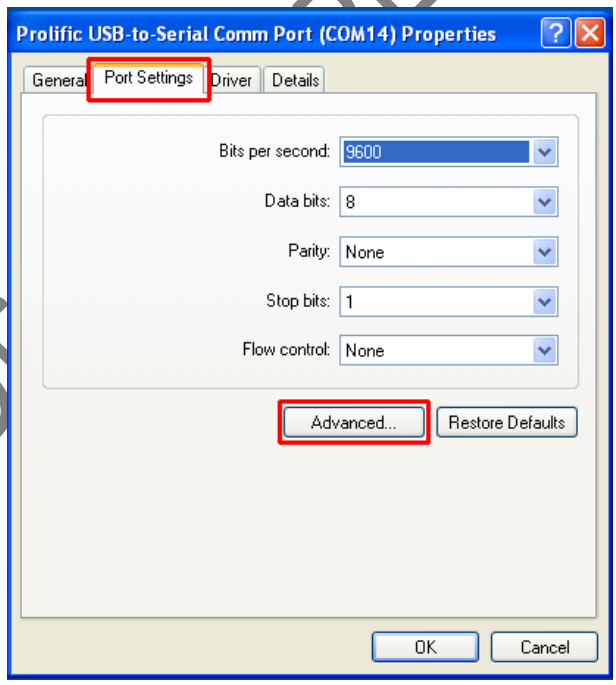


Figure 38 – COM Port Properties

5. Use the *COM Port Number* dropdown to select a COM port number in the range COM1 to COM8. Then click *OK*. Note that some values may be listed as "in use" if they have ever been assigned (for example, if other USB devices have been attached to the PC in the past). Use the Device Manager to determine which COM ports are actually in use.

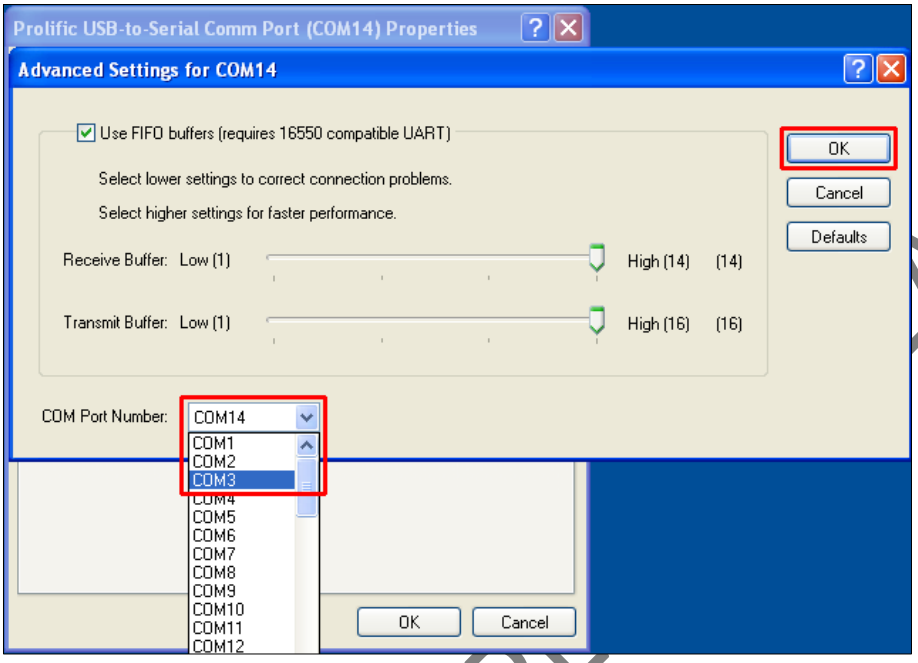


Figure 39 – COM Port Advanced Settings (Prolific)

6. The *Advanced Settings* window for the FTDI driver is more complex, but the reconfiguration method is the same.

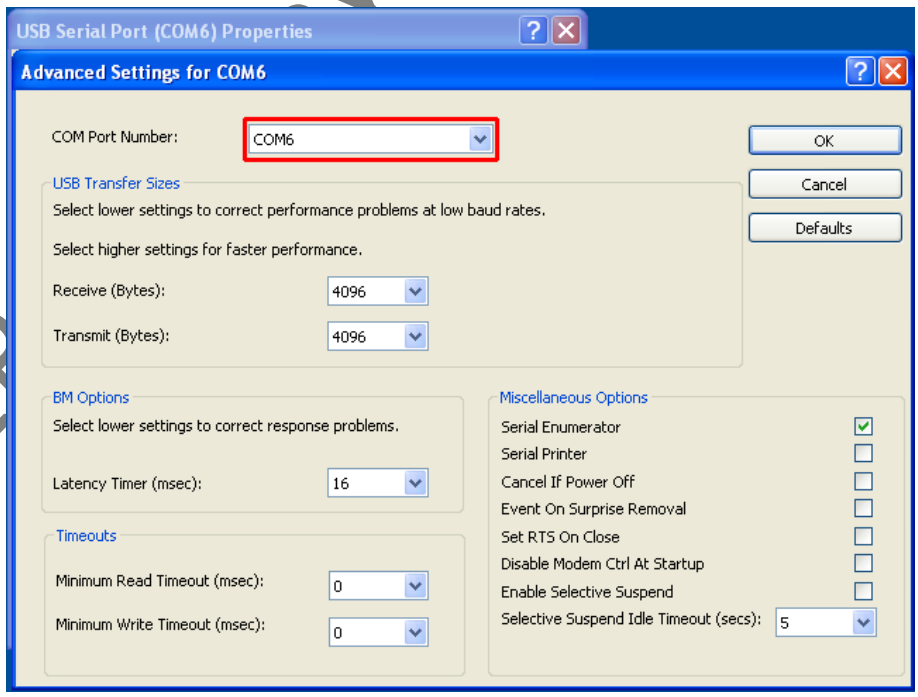


Figure 40 – COM Port Advanced Settings (FTDI)

7. It may be necessary to unplug the USB-to-Serial adapter, then plug it back before the adapter is correctly identified and initialised with the new COM port number.

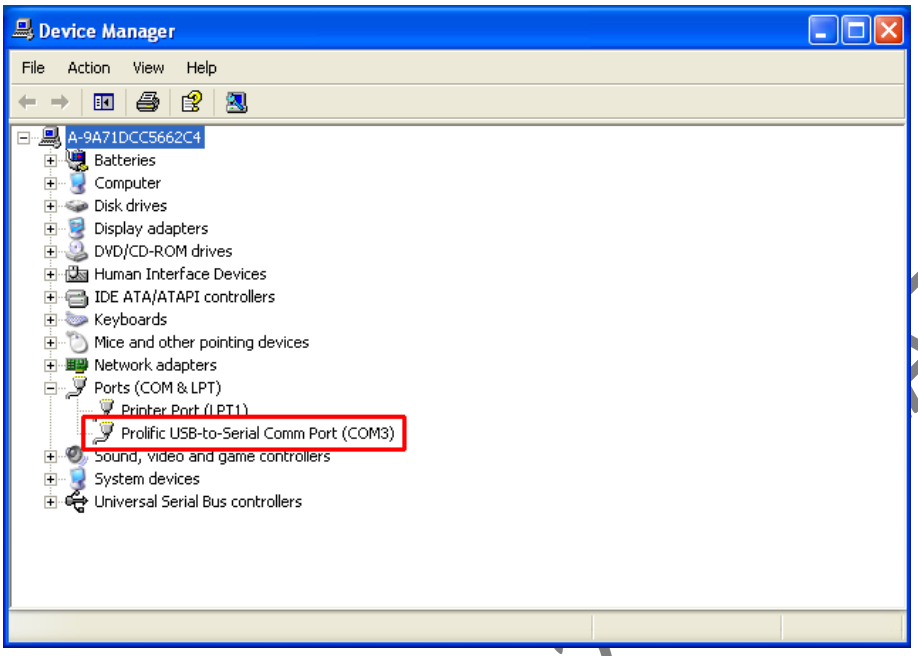


Figure 41 – Device Manager (Reconfigured Port COM3)

Simulator Software Package Configuration

Abel External Bells Configuration

Configuration of the Abel Simulator Software Package to use the Simulator Interface should also only need to be done once. All settings are saved in the Abel options file. This example is based on Abel 3.9.1.

To configure Abel to use the Simulator Interface, carry out the following steps. This manual described the minimum necessary to configure Abel to use the Simulator Interface, for full details on the overall configuration of Abel please refer to the product documentation.

1. Start Abel on the Simulator PC, and from the *Options* menu select *External Bells...* .
2. In the *External Bells* configuration window, click the *Discover Ports* button.

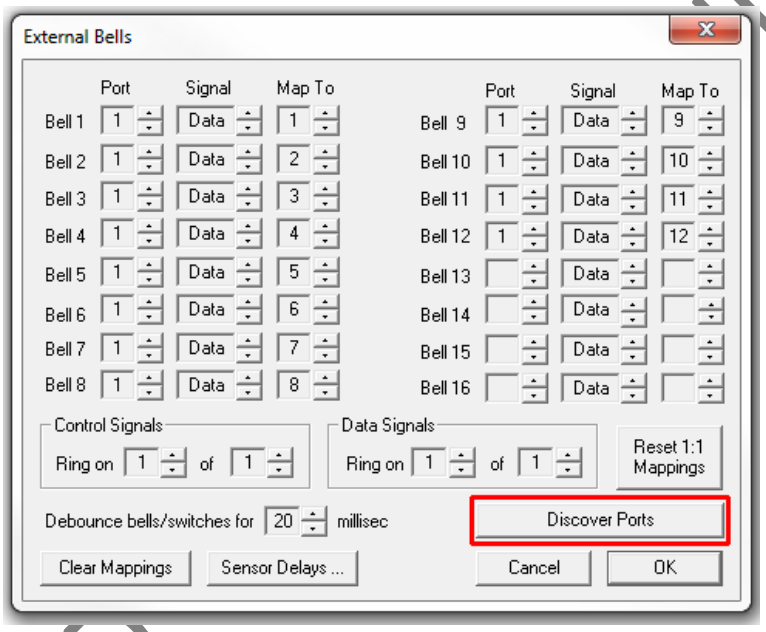


Figure 42 – Abel – Discover Ports

3. Verify that the serial port to be used is recognised by Abel, and then click *OK*. Note that Abel requires the serial COM port number to be between 1 and 8. Refer to the section above on USB-to-Serial adapters for instructions on reconfiguring port numbers.

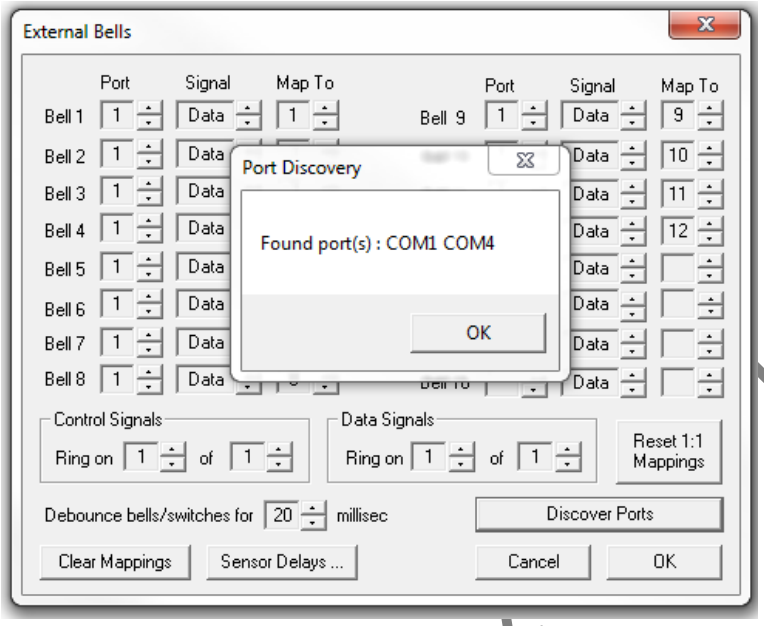


Figure 43 – Abel – Port Discovery

4. Set the *Port* field to the COM port number of the serial interface to be used. In this example the port *COM1* is used. This field should be set to the same value for all Simulator Sensors connected to the same Simulator Interface.

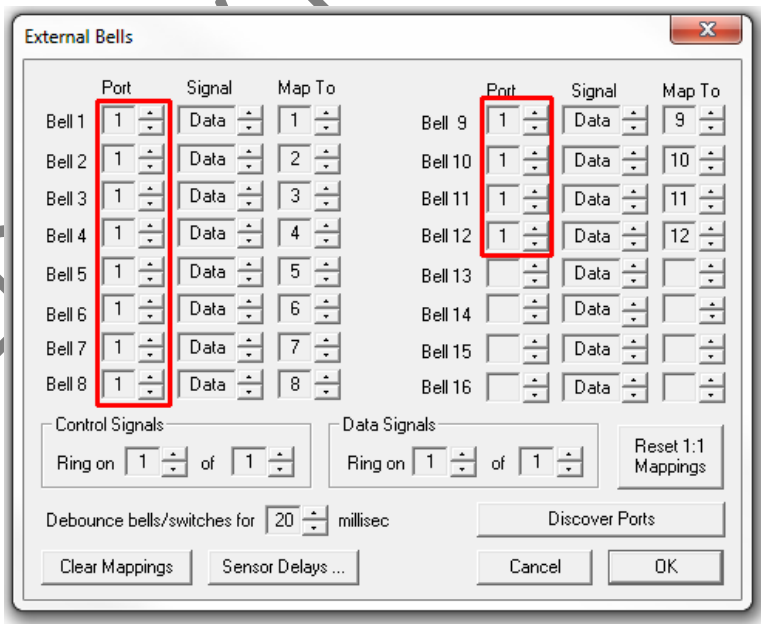


Figure 44 – Abel – Port Setting

5. Set the *Signal* field to *Data*, indicating that this external bell is connected via a Simulator Interface using the MBI protocol. Do this for all Simulator Sensors connected to a Simulator Interface[24].

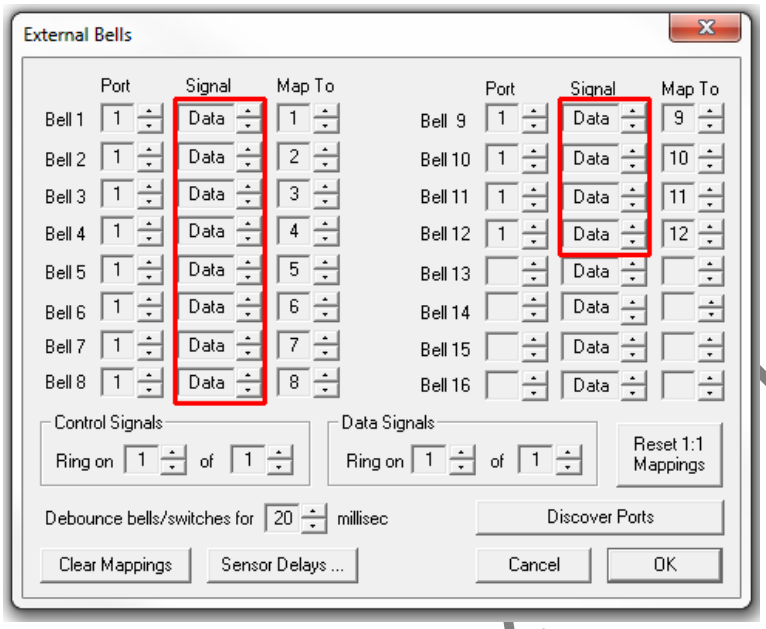


Figure 45 – Abel – Signal Setting

6. Set the *Map To* fields to map Bell 1 to 1, 2 to 2 and so on, for as many Sensor Heads are connected to the Simulator Interface. This does not mean that Sensor Head number 1 will always the simulated Treble (and so on); the use of the simulated bells can be changed in the main Abel screen – refer to the Abel documentation.

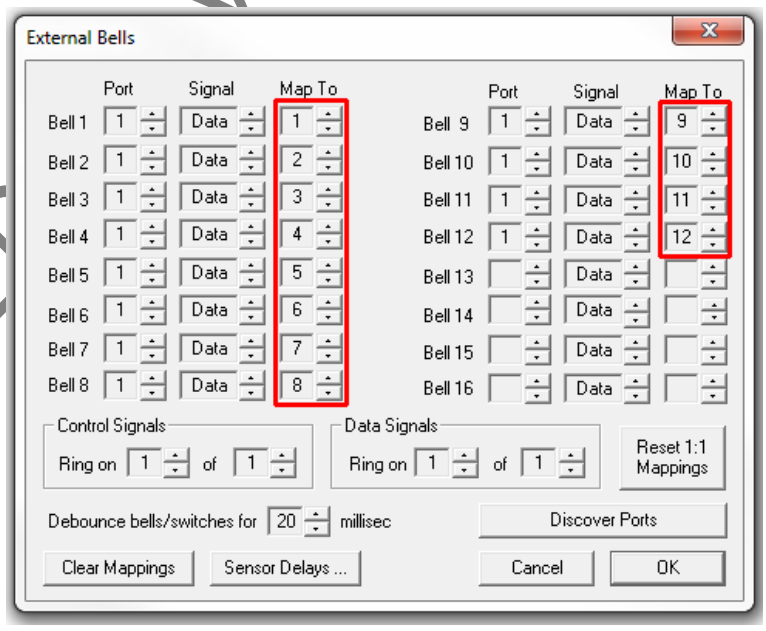


Figure 46 – Abel – Mappings

7. Leave the *Control Signals* and *Data Signals* set to *Ring on 1 of 1*, and the de-bounce value at 20ms.

[24] It is possible to use more than one Simulator Interface, on different COM ports, on the same Simulator PC.

8. The entries for Bells 13 – 16 should not be configured[25].
9. The resulting configuration should look similar to this:

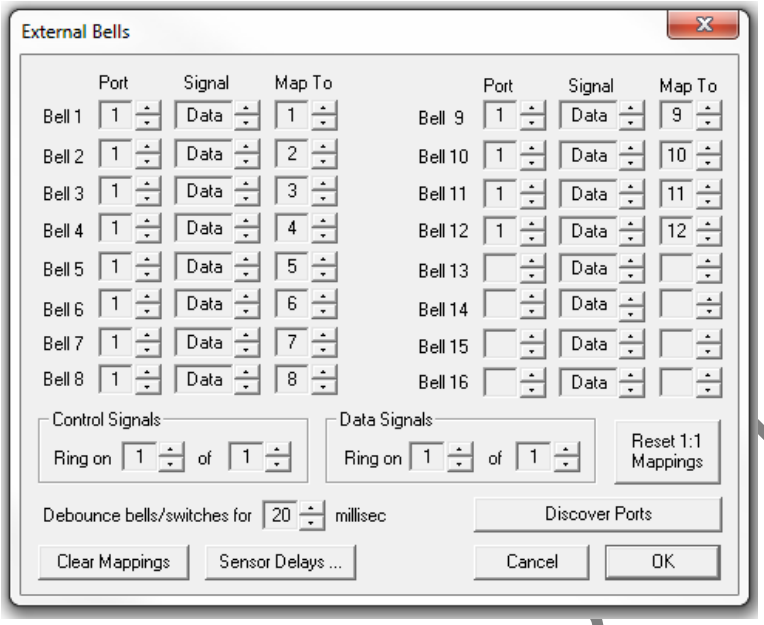


Figure 47 – Abel – Completed External Bells Configuration Example

10. Click on the *Sensor Delays…* button to configure the delay timers.

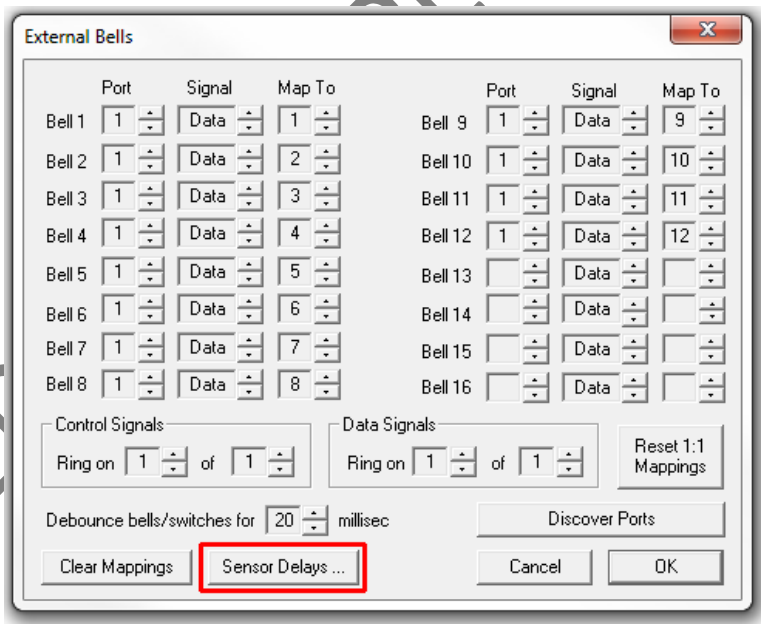


Figure 48 – Abel – Sensor Delays

[25] Unless a second Simulator Interface is connected.

11. In the *Sensor Delays* window, set the delay for each bell to an appropriate value, so that the simulated bell sounds as closely as possible to the same time as the real bell (this is best done with the real bell un-silenced), and leave the *Apply Delays in Software for Control Signals* checkbox unchecked. Note that in Abel the delay values are specified in 1/100ths of a second (centiseconds), not milliseconds. Click *Apply* to upload the values to the Simulator Interface, and then click *OK*.
    The following example shows a Simulator Interface with setting of delay timers for Liverpool Cathedral completed.

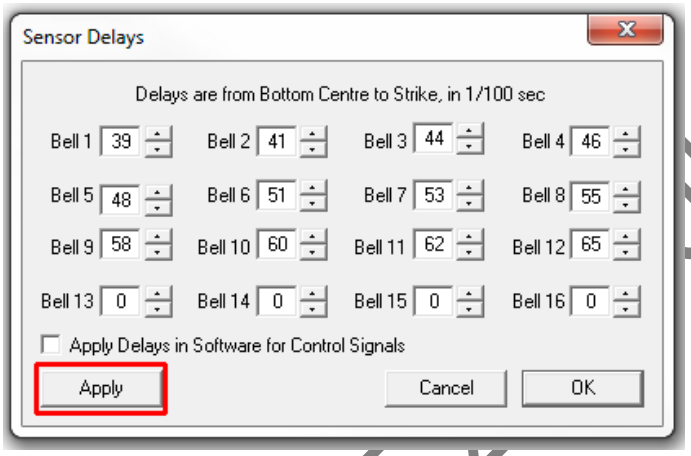


Figure 49 – Abel – Sensor Delays Dialogue

12. Click *OK* in the *External Bells* window to close it.
13. Save the new options by selecting *Save Options* from the *Options* menu. If the options have changed, Abel will prompt for this when the program is closed.
14. Abel should now be configured to use the Simulator Interface. Test each bell in turn and check that the simulated bells are correctly mapped to the real bells.

Beltower Sensors Configuration

Configuration of the Beltower Simulator Software Package to use the Simulator Interface should also only need to be done once. All settings are saved in the Beltower configuration file. This example is based on Beltower 2015 (12.29).

To configure Beltower to use the Simulator Interface, carry out the following steps. This manual described the minimum necessary to configure Beltower to use the Simulator Interface, for full details on the overall configuration of Beltower please refer to the product documentation.

1. Identify the COM port number of the serial interface to which the Simulator Interface is connected, if necessary by using Windows Device Manager as described above.
2. Start Beltower on the Simulator PC, and from the *Settings…* menu select *Sensors*.
3. In the *Input Mode* dropdown, select *Serial interface on COM…*, using the correct serial interface COM port number. Note that Beltower requires the serial COM port number to be between 1 and 8 (Beltower 2016 increases this limit to 32). Refer to the section above on USB-to-Serial adapters for instructions on reconfiguring port numbers.

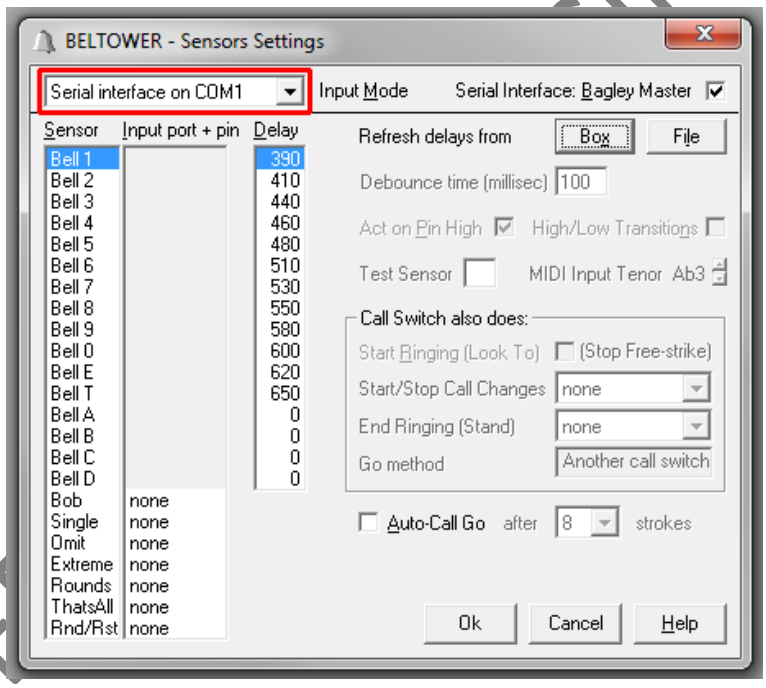


Figure 50 – Beltower – Input Mode

4. Set the delay for each bell to an appropriate value, so that the simulated bell sounds as closely as possible to the same time as the real bell (this is best done with the real bell un-silenced), and leave the *Apply Delays in Software for Control Signals* checkbox unchecked. Note that in Beltower the delay values are specified in 1/1000ths of a second (milliseconds), in increments of 10ms. Double-click each timer value to edit it.

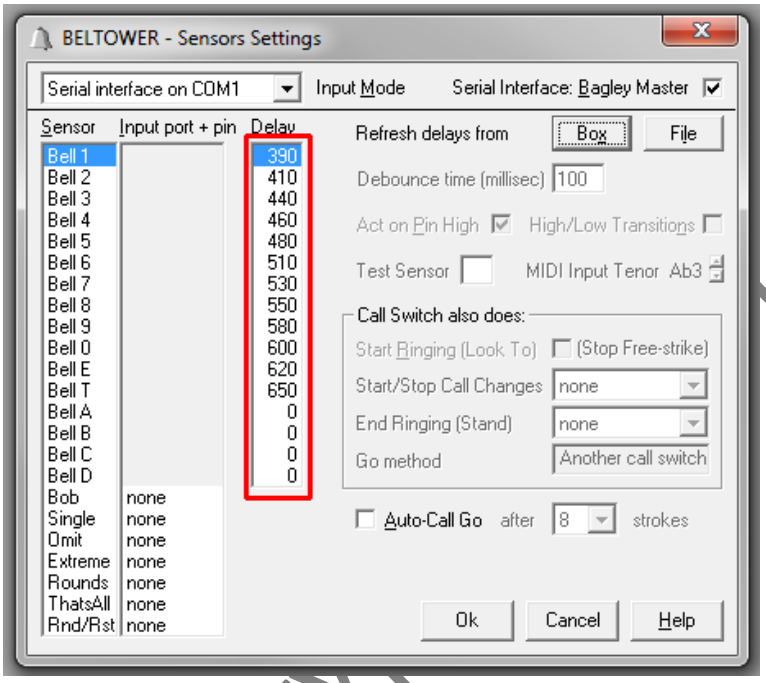


Figure 51 – Beltower – Sensor Delays

5. Ensure that the *Serial Interface: Bagley Master* checkbox is ticked.

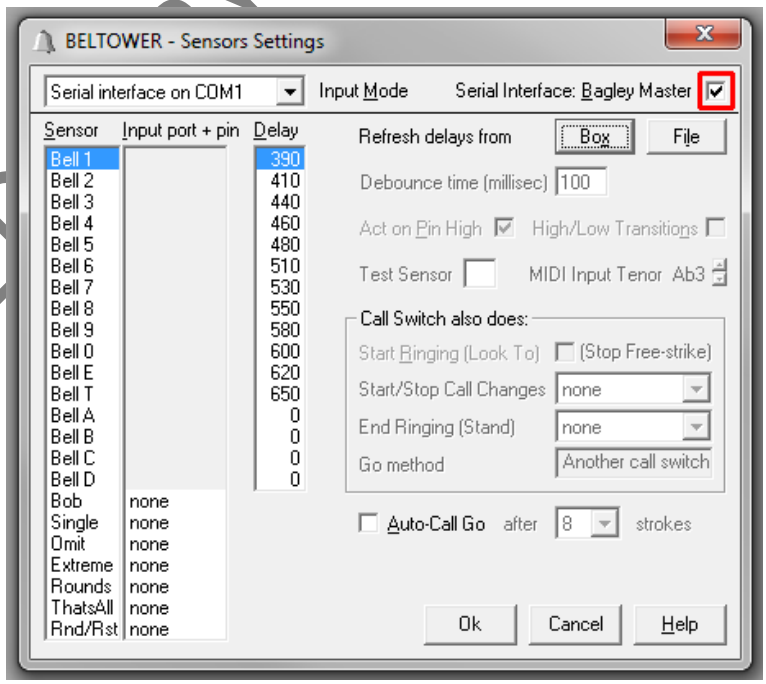


Figure 52 – Beltower – Master Mode

6. Click *OK* in the *Sensor Settings* window to upload the values to the Simulator Interface and close the window. The following example shows a Simulator Interface with setting of delay timers for Liverpool Cathedral completed.

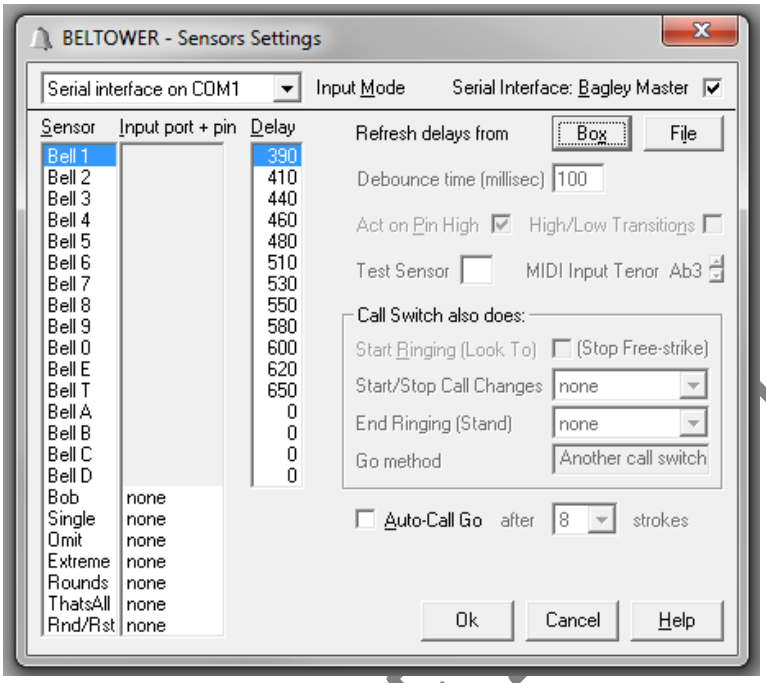


Figure 53 – Beltower – Completed Sensor Settings Example

7. Save the new options by selecting *Save Selections* from the *File* menu. If the options have changed, Beltower will also prompt for this when the program is closed.
8. Activating the sensor configuration is done in one of two different ways, depending on whether Beltower is being used in *Basic* or *Advanced* mode.
9. In *Basic* mode, select *Tower Bell Sensor(s)* from the dropdown shown when the application starts.

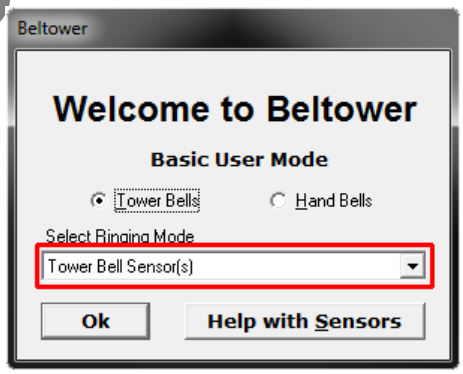


Figure 54 – Beltower – Basic Mode

10. When Beltower is running in *Basic* mode, the sensor configuration can be activated by selecting *Ring Options…* from the *Options* menu, then selecting *Tower Bell Sensor(s)* from the *Timing Options* dropdown, and clicking the *Initialize Free-Strike* button.

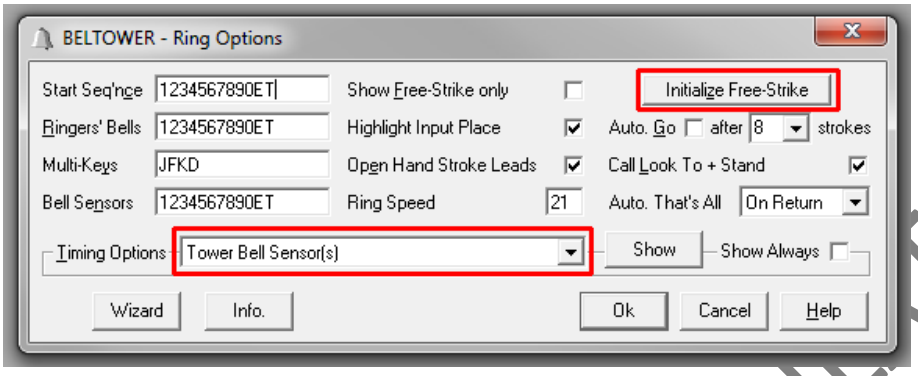


Figure 55 – Beltower – Basic Mode Options

11. When Beltower is running in *Advanced* mode, the sensor configuration can be activated by selecting *Ring Options…* from the *Options* menu, then checking the *External Sensors* radio button, and clicking the *Initialize Free-Strike* button.

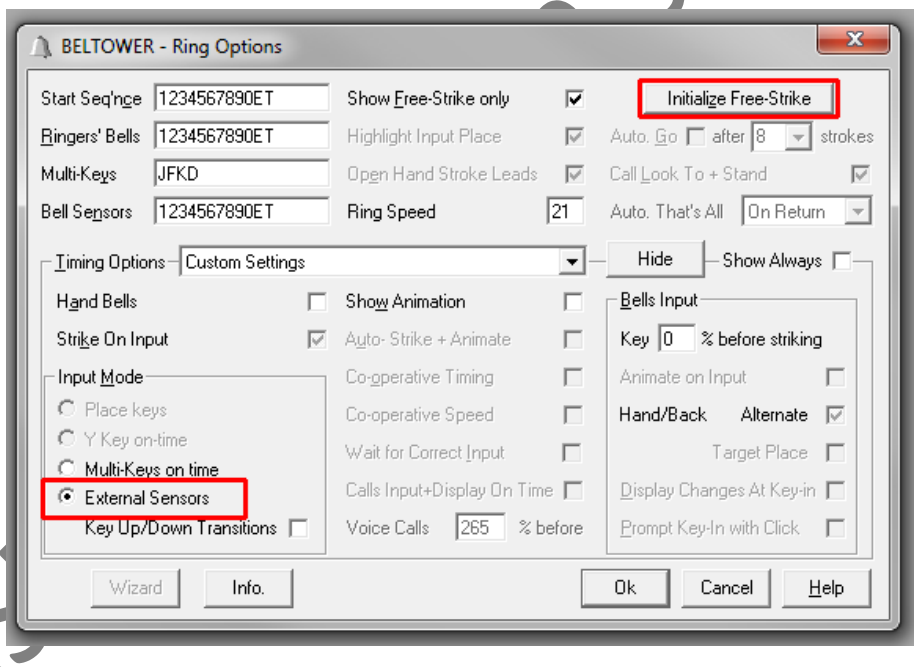


Figure 56 – Beltower – Advanced Mode Options

12. Beltower should now be configured to use the Simulator Interface. Test each bell in turn and check that the simulated bells are correctly mapped to the real bells.

Virtual Belfry Sensors Configuration

Configuration of the Virtual Belfry Simulator Software Package to use the Simulator Interface should also only need to be done once. All settings are saved by Virtual Belfry in the Windows Registry. This example is based on Virtual Belfry 3.4.

To configure Virtual Belfry to use the Simulator Interface, carry out the following steps. This manual described the minimum necessary to configure Virtual Belfry to use the Simulator Interface, for full details on the overall configuration of Virtual Belfry please refer to the product documentation.

1. Start Virtual Belfry on the Simulator PC, click on the *Sensors* vertical tab, then click the *Configure…* button.

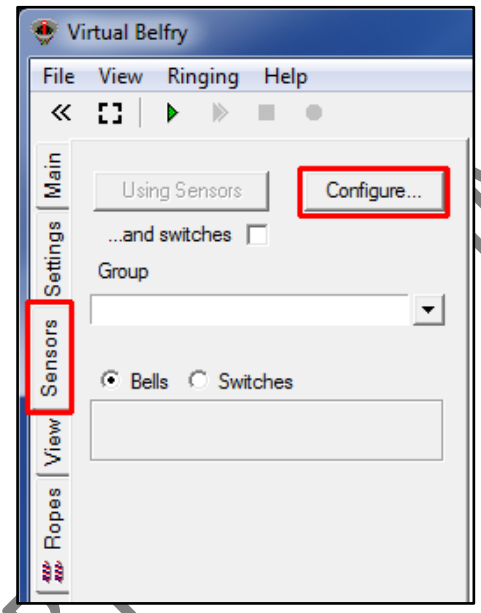


Figure 57 – Virtual Belfry – Main Window

2. In the *Configure Sensors* window, click the *New…* button to create a new sensor group.

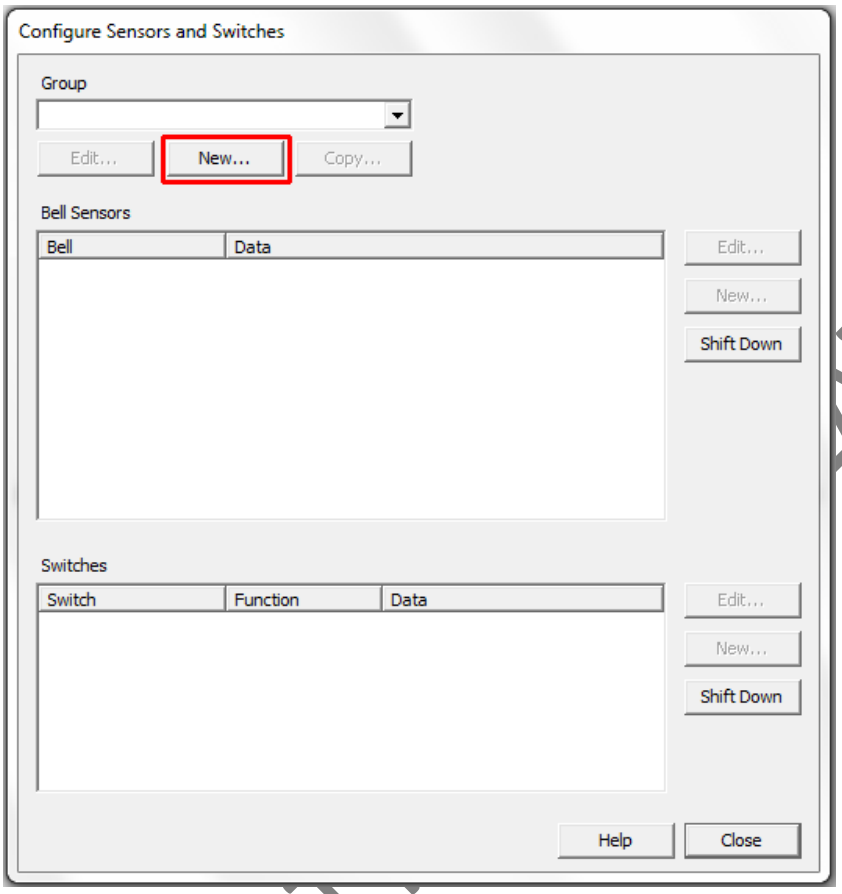


Figure 58 – Virtual Belfry – Add New Sensor Group

3. Give the sensor group a name, and then click *OK*.

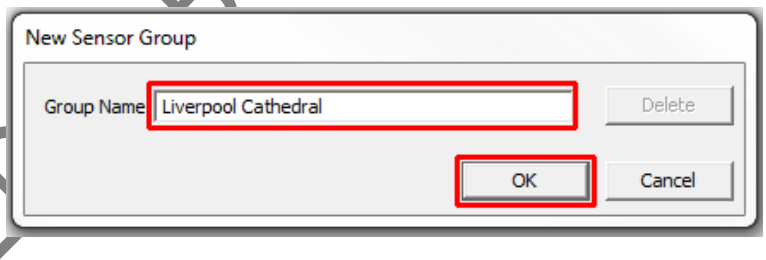


Figure 59 – Virtual Belfry – New Sensor Group

4. In the *Configure Sensors* window, select the new sensor group in the *Group* dropdown, and then click *New…* to create a new sensor.

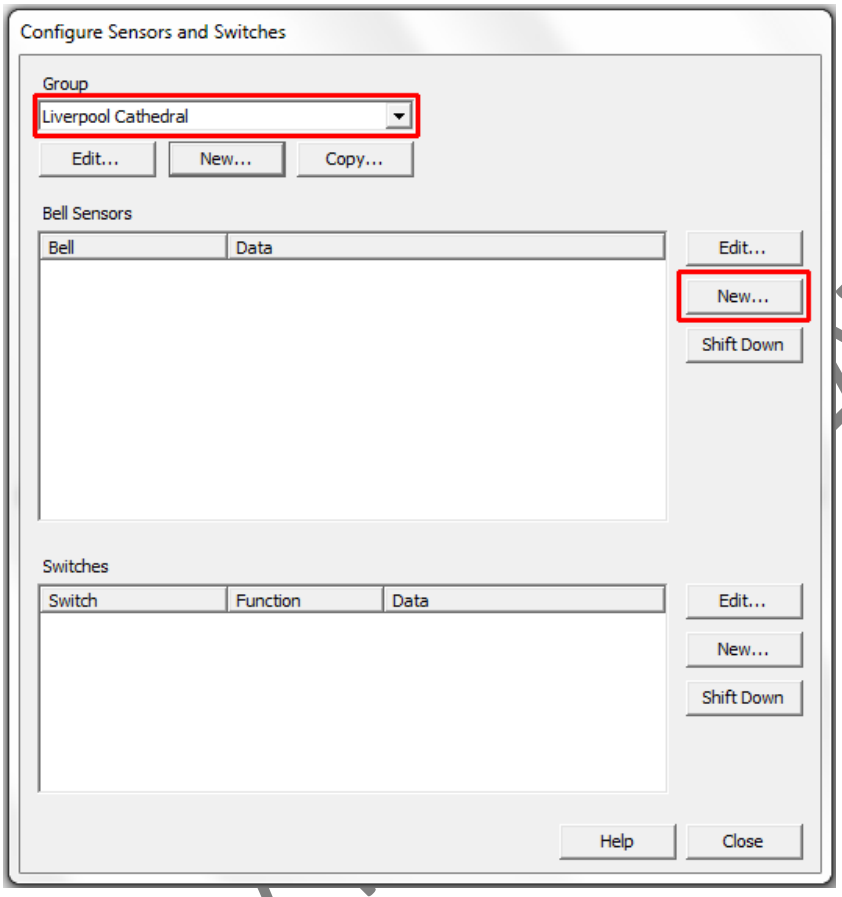


Figure 40 – Virtual Belfry – Add New Sensor

5. In the *New Sensor* window, configure the first sensor as follows:
   - In the *Bell* field, enter a unique name for the sensor, in this example *Sensor #1*.
   - In the *Type* dropdown, select *Bagley Multi-Bell Interface*.
   - In the *Port* field, enter the full name of the COM port to be used, in this example *COM1*. Virtual Belfry displays a dynamically updated list of available COM ports in the large box at the top right of the window.
   - In the *Signal* field, select the Simulator Interface signal that corresponds to this sensor, in this case channel *1*.
   - In the *Wait* field, enter the delay for this bell to an appropriate value, so that the simulated bell sounds as closely as possible to the same time as the real bell (this is best done with the real bell un-silenced). Note that in Virtual Belfry the delay values are specified in 1/1000ths of a second (milliseconds).
   - Tick the *All delays written to, and handled by, the interface* checkbox.
   - Set the *Ignore* field to *20ms*, and the signal number fields to *1 of 1*.

- Click *OK* to close the *New Sensor* window.

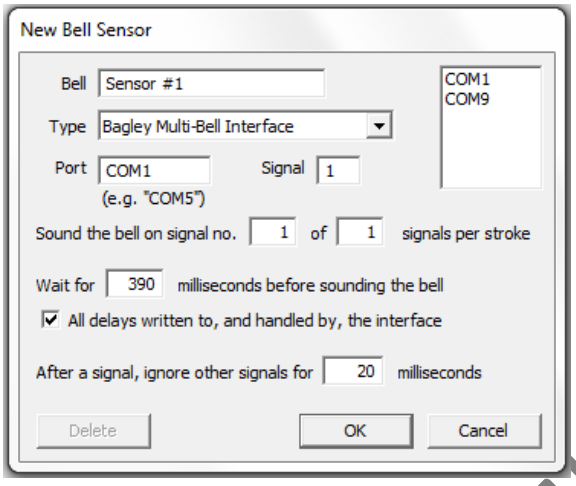


Figure 61 – Virtual Belfry – First New Sensor

6. Repeat steps 4 and 5 for the second and each successive sensor. The only values which should be different are the *Bell*, *Signal* and *Wait* fields.

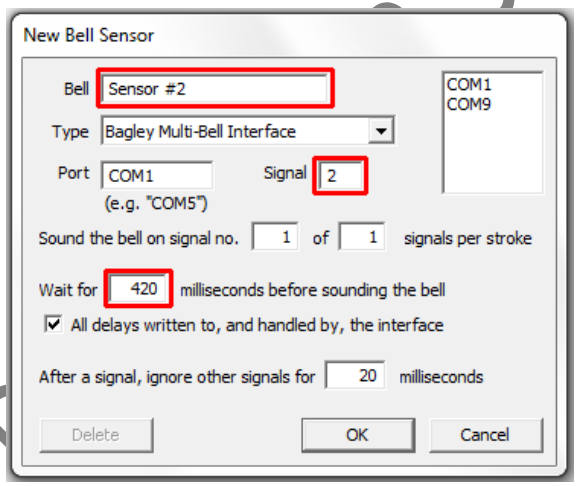


Figure 62 – Virtual Belfry – Subsequent Sensors

7. Click *Close* in the *Configure Sensors* window to upload the values to the Simulator Interface and close the window. The following example shows a completed sensor group with delay timers set for Liverpool Cathedral.

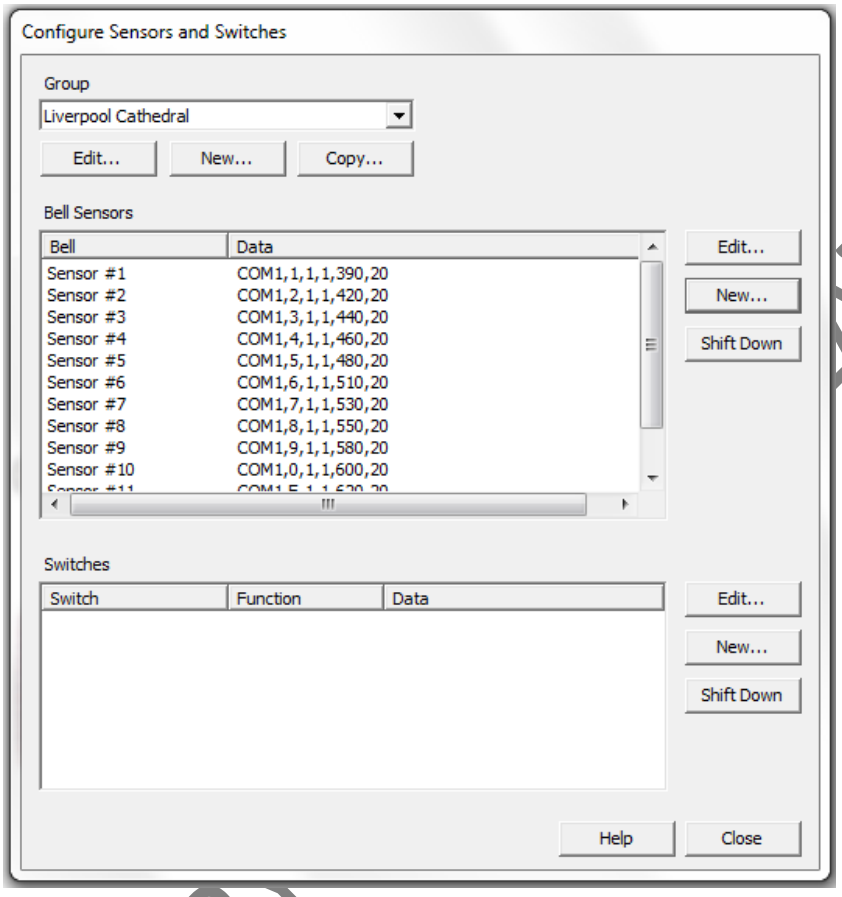


Figure 63 – Virtual Belfry – Completed Sensor Configuration Example

8. To activate the sensor group for silent practice, click on the *Using Sensors* button on the *Sensors* vertical tab, and tick the checkboxes for the bells in use.

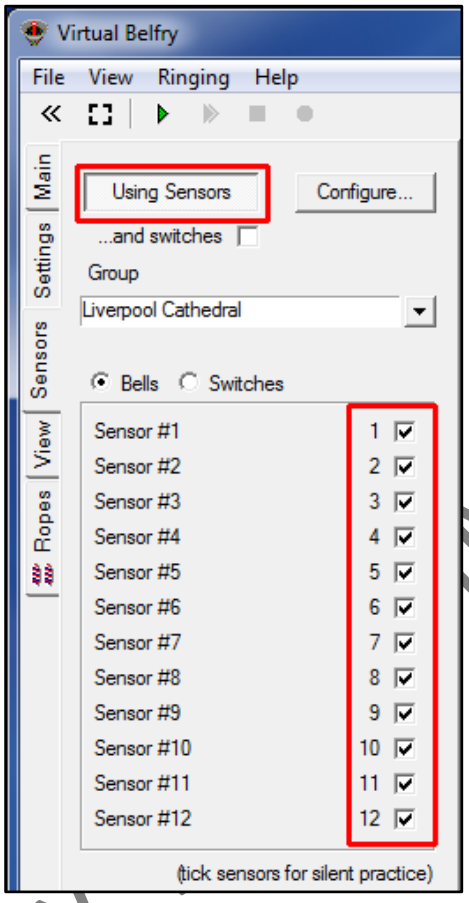


Figure 64 – Virtual Belfry – Using Sensors

9. Virtual Belfry should now be configured to use the Simulator Interface. Test each bell in turn and check that the simulated bells are correctly mapped to the real bells. The *Monitor* button on the *Sensors* vertical tab opens a monitor pane in which sensor output can be observed. New data is added at the top of the pane.

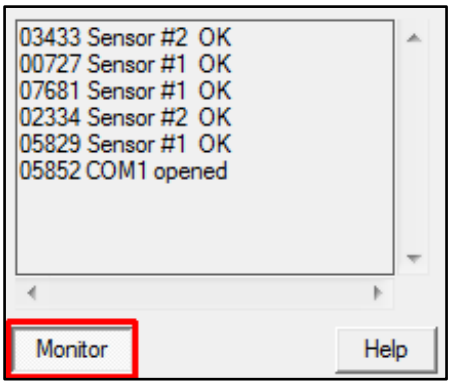


Figure 65 – Virtual Belfry – Monitor Function

10. Sensor groups may be copied and edited, facilitating switching between multiple different sensor configurations.

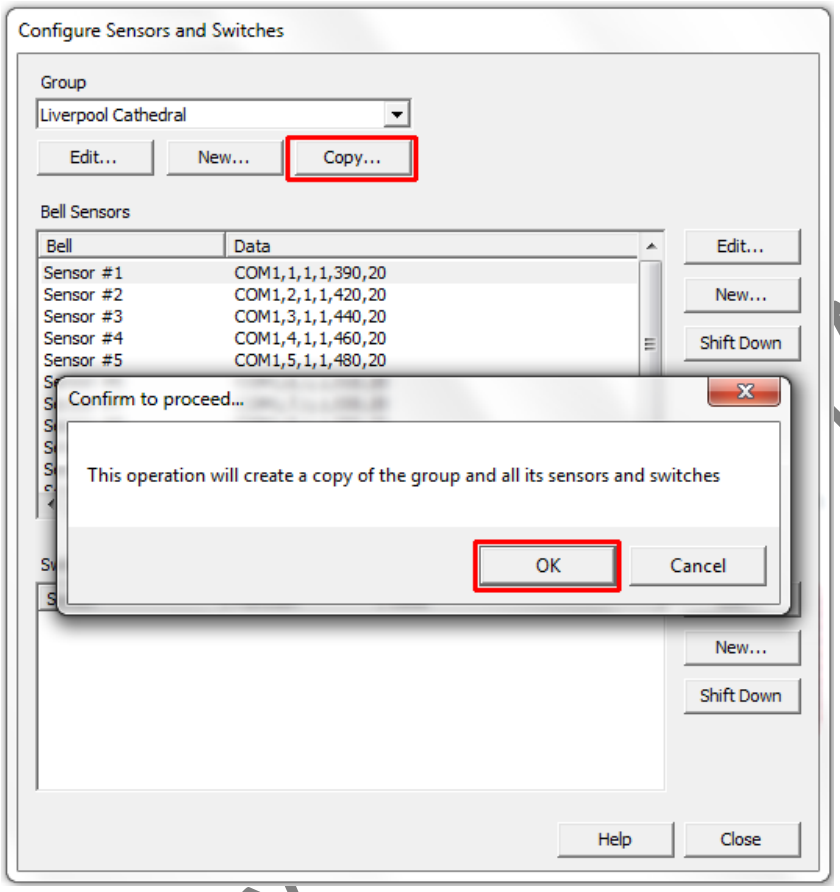


Figure 66 – Virtual Belfry – Copy Sensor Group

Ringing Subsets of Bells

Where a simulator is installed on a complete ring of bells, it may be desired to ring a subset of bells, for example the front 6 of a heavy 8, the front 8 of a ring of 12, or the back 8 or 10 of a ring of 12. The following examples show how this may be done using Abel, similar approaches are possible with the other Simulator Software Packages.

Ringing the Light Bells

Sensors Heads should be connected with the treble connected to Channel 1 of the Simulator Interface, sequentially down to the tenor.

The MBI Protocol, as currently implemented, supports a maximum of 12 bells on a single Simulator Interface. There is therefore no additional sensor installed on accidentals such as Sharp 2nd or Flat 6th bells. However, when Abel is configured as above to simulate, say, 12 bells, simply ringing the front 8 bells will sound out of key.

- In order to ring, for example, the front 8 of a 12 (with 2♮ bell) and have them sound in key, it is necessary to change the configuration of Abel. Loading one of the 8-bell or 7-bell method files (*8bell.mcf/7bell.mcf*) from the *File | Open…* menu will cause Abel automatically to shift

the sound options to a ring of 8 in the correct key, starting from the treble, and automatically adjusting the 2nd up a semitone[26].

- Likewise, loading a 6-bell or 5-bell method file (*6bell.mcf/5bell.mcf*) from the *File | Open…* menu will cause Abel to shift the tuning to a true light 6, starting from the treble.
- Reloading one the 12-bell or 11-bell method files (*12bell.mcf/11bell.mcf*) restores the tuning to a ring of 12 in the correct key.

Ringing the Back Bells

To ring, for example, the back 8 or back 10 of a ring of 12 for silent practice (where all bells are being rung by ringers and not simulated by the Simulator), is straightforward, and requires only that the simulator is set up as above for 12-bell ringing.

Ringing the back 8 or back 10 with a mixture of real and simulated ringers is more complicated, because Abel will always try to assign bell sensors from Channel 1 upwards. Hence loading the 7- or 8-bell method files causes Abel to switch to the front 8 bells, as noted above, not the back 8, and loading the a 9- or 10-bell method files causes it to attempt to switch to the front 10.

To work around this problem, set Abel up as a ring of 8 or 10, and save the configuration in a new options file by selecting *Save Options As…* from the *Options* menu. It will be necessary to change the mappings in the *External Bells* window to map the Sensor Heads for the real back bells to simulated bells 1 to 8 or 1 to 10.

The following screenshot illustrates an *External Bells* mapping for the back 8 of a ring of 12. These settings are also saved in the Abel Options file.

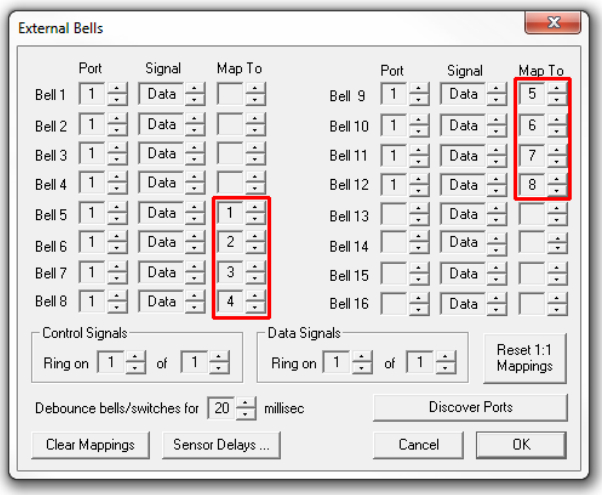


Figure 67 – Abel External Bells Dialogue (Back 8)

[26] One consequence of this behaviour is that it is quite possible to ring subsets of bells correctly in key on the simulator, which would not be possible on the real bells; for example the front 10 of a ring of 12, or the front 6 of a ring of 8.

The custom configuration can then be loaded by selecting *Open Options File…* from the *Options* menu, or by specifying it as a command line option to a desktop shortcut.

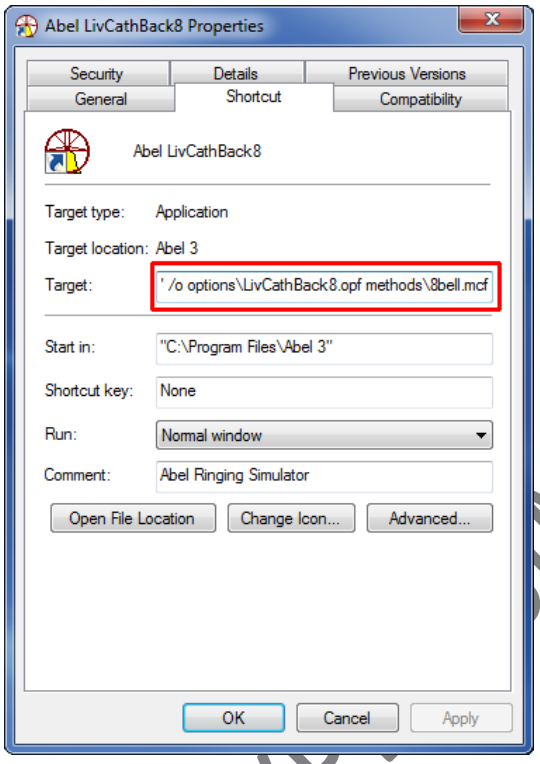


Figure 68 – Windows Shortcut Creation Dialogue

(The *Target* field reads in full, *C:\Program Files\Abel 3\Abel3.exe" /o options\LivCathBack8.opf methods\8bell.mcf*)

A similar approach may be used to create configuration options files for the back 10 of a ring of 12, or the back 6 of a ring of 8, and so on.

Delay Timer Calibration

For accurate simulation of the real bells, the single reflector approach requires that the delay timer for each bell is set so that the Simulator Interface sends the strike signal to the Simulator at exactly the point at which the real or untied bell would have struck. This delay time is specific to each bell, but for most bells is about 0.5s.

For odd-struck bells, this time may be different between handstroke and backstroke. The Simulator Interface uses only a single timer value for each bell, so for badly odd-struck bells this will be a compromise value.

Two methods of setting the timer values have been used: A simple acoustic method, and an approach using an electronic sound sensor. Both require the ringing of open bells.

Acoustic Method

To configure the Simulator Interface timers using the acoustic method:

1. Start the Simulator Software Package on the Simulator PC.
2. Ring each bell in turn, open, and compare the sound of the bell and the simulated sound from the Simulator.
3. If the real bell sounds before the Simulator, reduce that bell's delay timer value.
4. If the Simulator sounds before the real bell, increase that bell's delay timer value.
5. Repeat this process until the sound of the real bell and the sound from the Simulator are as close to coincident as possible.
6. Repeat for each of the other bells in turn.

This approach is very simple, and is recommended for almost all installations.

Electronic Method

During development of the Liverpool Cathedral Simulator project, an experimental electronic method for determining delay timer values was trialled. This involved temporarily replacing one of the optical Sensor Heads with a sound operated sensor placed in the belfry.

- Debugging options in the Simulator Interface firmware were then used to compare the strike times reported by the sound sensor and the optical sensor for each bell.
- Comparison of these two values could be used to determine the interval between the bell passing bottom dead centre (optical sensor) and the actual strike point (sound sensor), and thus the delay timer value that should be configured in the Software Simulator Package.
- However, when this method was conducted in practice, the prolonged reverberation of the bell caused multiple re-triggering of the sound operated sensor. Sensitivity adjustment of the sensor was difficult, and a limited number of good observations were obtained.
- This approach is therefore not recommended for general use when setting up a simulator, but may be useful for analysing the odd-struckness of the bells.

BDC-to-Strike Intervals

The following chart shows the average time differences obtained during development between the triggering of the standard infra-red optical Sensor Head and the sound operated sensor for each bell at Liverpool Cathedral. Separate average values are shown for handstroke, backstroke, and both strokes combined. The total sample size was 93 blows.

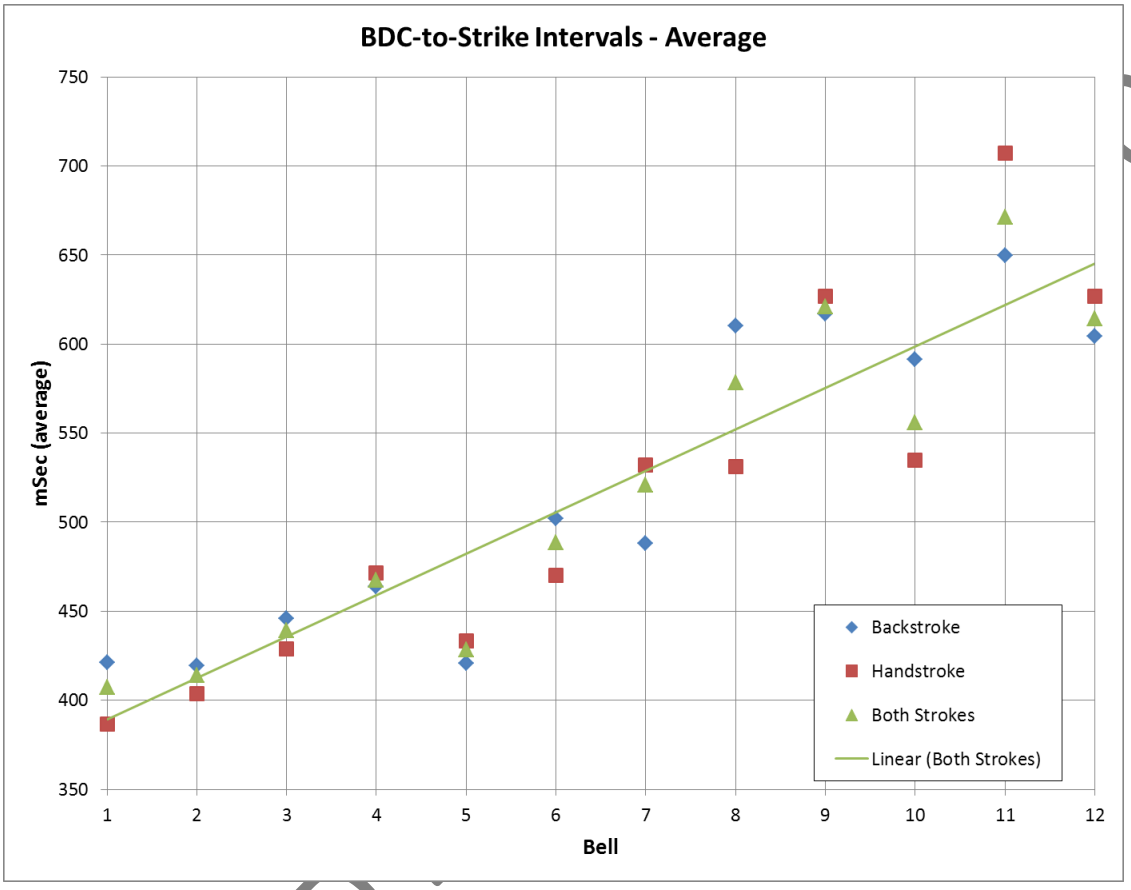


Figure 68 – Liverpool Cathedral Odd-Struckness Chart

One Bell Simulator Interface

Hardware Overview

The One Bell Simulator Interface is an integrated sensor and interface unit for a single bell, designed to be housed in a small standard Sensor Head enclosure, as illustrated in the following photograph:

Figure 70 - One Bell Simulator Interface (Optical Sensor)

- The hardware is described in detail in the accompanying One Bell Simulator Interface Hardware Manual. The illustration above shows a One Bell Simulator Interface with a standard optical detector, but alternative configurations are possible.
- The One Bell Simulator Interface is based on an ATtiny85 microcontroller, and runs a reduced and modified version of the firmware described previously.
- The firmware and operational differences of the One Bell Simulator Interface are summarised in this section of the manual. Reference to the firmware source code is recommended. This manual is based on v1.3 of the One Bell Interface firmware.

Firmware Differences

The One Bell Simulator Interface firmware has the following principal design differences from the main Simulator Interface firmware:

- The Interface hardware is based on the ATtiny85 microcontroller, driven by the internal 8MHz oscillator. The firmware is reduced in complexity and functionality to fit in the limited flash and SRAM capacity of this microcontroller.
- The memory footprint of the One Bell Simulator Interface firmware v1.3 on an ATtiny85 microcontroller is as follows:
    - 7,396 / 8,192 Bytes Flash Memory
    - 222 / 512 Bytes SRAM
    - 21 / 512 Bytes EEPROM

- The ATtiny85 has no dedicated serial UART, therefore the Arduino *SoftwareSerial* library[27] is used for processing of serial I/O from the Simulator. *SoftwareSerial* does not support the *serialEvent()* function, so the software serial port is polled on each program loop instead.
- The One Bell Interface firmware supports only a single input channel, and this is always Channel 1. (The Interface hardware is intended to be integrated into a single enclosure with the sensor, as shown above.)
- The *Debug Show LED* flag is not supported, and the number of active channels is not configurable. These features only make sense on multi-channel interfaces.
- The Interface uses a single LED for diagnostic purposes.
- The Interface runs only at 2400bps. The facility to configure other speeds for diagnostic purposes is not supported.
- The Sensor Enable/Disable and Test Mode functions are not supported.
- The VTSerial library is omitted, to reduce the overall size of the firmware.

Operational Differences

The One Bell Simulator Interface firmware has the following principal operational differences from the main Simulator Interface firmware:

- The Interface configuration CLI is less sophisticated, does not use colour or terminal control.
- The following CLI commands are not supported in the One Bell Interface firmware:
  - "H" – display the help screen for the Simulator CLI. This command is omitted to reduce the overall size of the firmware.
  - "#" – configure the number of active channels. This command is not required because there is only one input channel.
  - "E" – enable/disable sensors. This command is not required because there is only one input channel.
  - "M" – configure the active Debug Mask. This command is not required because there is only one input channel.
  - "P" – configure serial port speed for diagnostics. This command is not required because there is only one input channel, and the volume of debug data is unlikely to overrun the serial port buffer.
  - "L" – display the debug help screen for the Simulator CLI. This command is omitted to reduce the overall size of the firmware.
  - "T" – enter test mode. This command is omitted to reduce the overall size of the firmware.
- The following CLI commands behave differently in the One Bell Interface firmware:
  - "B" – set de-bounce timer. Repeated use of this command cycles the value of the de-bounce timer from 1ms to 20ms. This command has been changed to reduce the overall size of the firmware.
  - "S" – save settings. This command saves only the simulator type and de-bounce timer to EEPROM. The number of channels is fixed at one and is not configurable, and the enable and debug masks are not supported.

[27] https://www.arduino.cc/en/Reference/softwareSerial

- "D" – enable debug mode. Repeated use of this command cycles the value of the debug flags around all possible combinations. This command has been changed to reduce the overall size of the firmware.
- The *Debug Show LED* debug flag is not supported. This feature only makes sense on multi-channel interfaces.

- When mapping Interface channels in Simulator Software Packages, the One Bell Simulator Interface is always Channel 1, and sends ASCII character "1" to the Simulator PC.

The following screenshots show examples of the simplified One Bell Interface firmware CLI:

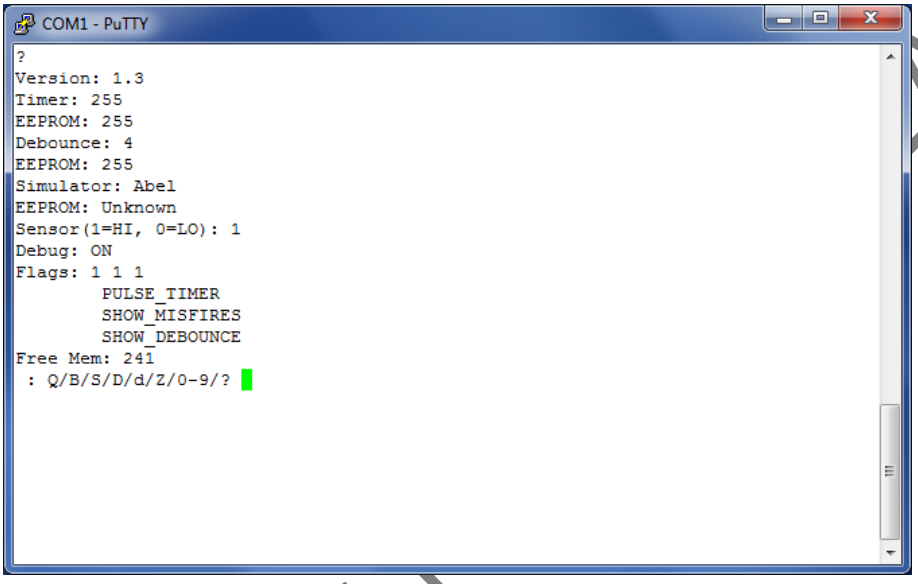


Figure 71 – One Bell Interface – Settings

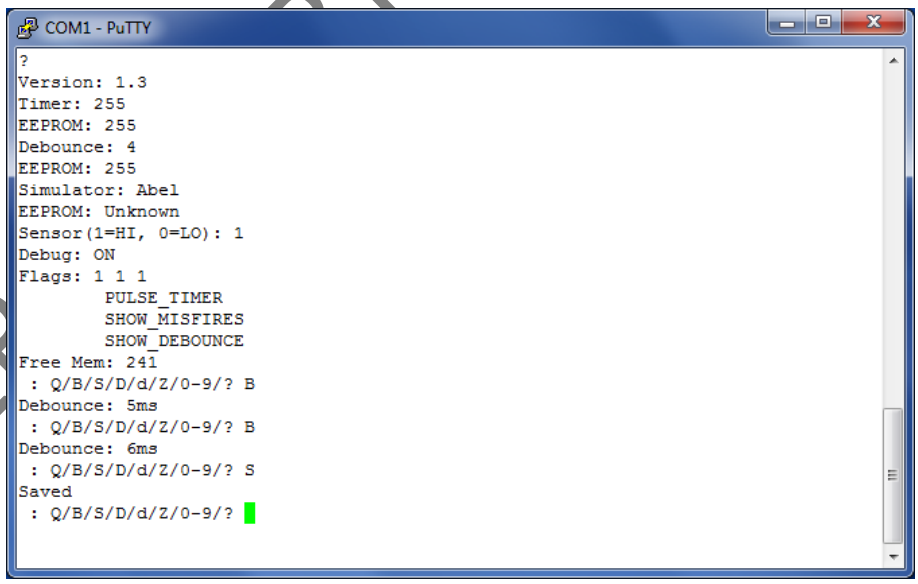


Figure 72 – One Bell Interface – De-Bounce Timer

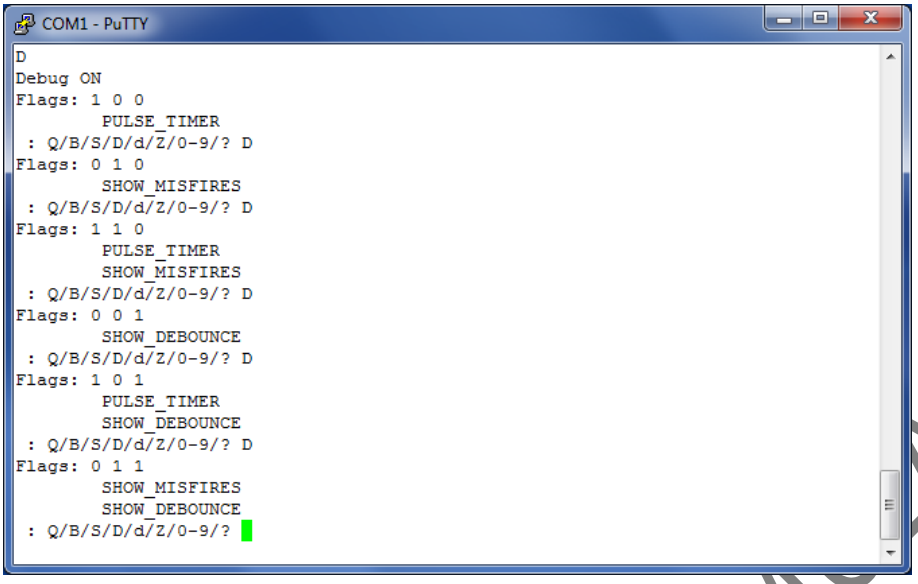


Figure 73 – One Bell Interface – Debug Settings

Appendices

Appendix A: MBI Protocol Description

The following description of the MBI Protocol is derived from information supplied by David Bagley and Chris Hughes.

The MBI aggregates signals from a number of Sensor Heads, one per bell, into a stream of ASCII characters sent over a RS-232 serial data link to the Simulator. This link operates at 2400 bps, 8 data bits, 1 stop bit, no parity.

Within the MBI, each bell has a delay timer value configured which corresponds to the time between the bell passing through the bottom dead centre of its swing, and the time the clapper would normally strike the bell. These delay values are uploaded by the Simulator and stored by the MBI in non-volatile memory.

When a signal from a Sensor Head is detected, the MBI applies the delay timer (appropriate to that bell), and at the expiry of that timer the MBI sends the corresponding single ASCII character to the Simulator, which then produces the simulated sound of that bell. The characters are defined using the usual change ringing convention of “0”, ”E” & “T” for bells 10, 11 & 12. At present a maximum of 12 bells are supported by the protocol.

The MBI Protocol also includes provision for connecting up to 4 external command switches, for which the characters sent are “W”, “X”, “Y” & “Z”. This behaviour is not currently supported by Bagley MBI interface hardware, but Abel has support for the protocol functionality.

The MBI also receives serial input from the Simulator. The command set defined by the protocol is described in Appendix B. Not all features of the command set are used by all Simulator applications.

Appendix B: MBI Protocol Commands

The following table lists the commands defined in the MBI Protocol.

Table 3 – MBI Protocol Commands

Command	Function
0xFD	The 0xFD command is used to detect the presence of a Simulator Interface. The expected response is for the Interface to return one byte, hexadecimal value 0xFD. • This command does not appear to be used by tested versions of either Abel, Beltower or Virtual Belfry.
0xFE	The 0xFE command is used to retrieve the current timer delay versions from the Simulator Interface. The expected response is for the Interface to return 13 bytes, one byte for each bell in the order 1 to 12, containing the current delay value in centiseconds expressed as a hexadecimal value, and a trailing termination byte, hexadecimal value 0xFF. • This command is used by the tested version of Beltower.

	• This command does not appear to be used by the tested version of Abel or Virtual Belfry.
0xNN … 0xFF	Timer delay data is sent to the Simulator Interface by the Simulator Software Package as a series of 13 bytes, one byte for each bell in the order 1 to 12, containing the new delay value in centiseconds expressed as a hexadecimal value, and a trailing termination byte, hexadecimal value 0xFF. All 12 values are always sent, even if fewer than 12 bells are configured. In this case the data is padded with 0x00 bytes. • This command is used by tested versions of Abel, Beltower and Virtual Belfry.

Appendix C: Metrics Tables

Inter-Blow & Inter-Row Interval

This appendix contains a more detailed analysis of the inter-blow and inter-row intervals for a large sample of rings of bells. A total of 4096 records of peals and quarter peals were downloaded from the *Ringing World Bell Board* database using the experimental XML API.

This data was then sanitised to remove all hand bell performances, any performances for which the duration or number of changes was not specified, and to convert the duration to consistent units. This left a total of 3080 performances comprising 54.7 Million blows for analysis.

The following table shows the results of this analysis:

Table 4 – Inter-Blow & Inter-Row Intervals

Bells[28]	Total Performances	Total Blows	Inter-Row Interval (s)	Inter-Blow Interval (ms)	
12	111	4452144	2.93	244	Max
			1.96	164	Min
			2.41	200	Mean
10	265	8554130	2.84	284	Max
			1.67	167	Min
			2.22	222	Mean
8	1203	25199176	2.69	336	Max
			1.25	157	Min
			2.10	262	Mean
7	9	81396	2.26	323	Max
			2.05	292	Min
			2.18	312	Mean

[28] Including the cover bell, if any.

6	1341	15074688	2.68	447	Max
			1.06	177	Min
			2.01	334	Mean
5	116	1168285	3.00	600	Max
			1.12	224	Min
			1.99	398	Mean
4	23	149088	2.26	566	Max
			1.57	393	Min
			1.88	471	Mean
3	12	45432	2.10	698	Max
			1.67	556	Min
			1.95	651	Mean
All	3080	54724339	3.00	698	Max
			1.06	157	Min

Sensor Pulse Duration

The following table shows the small pendulum period, wheel radius, estimated wheel rim speed and theoretical pulse duration for each bell at Liverpool Cathedral, based on a reflector tape 25mm wide; and the observed pulse duration measured during development by the Simulator Interface firmware in debug mode, using standard infra-red detectors.

The wheel rim speed is calculated as $4\pi R/T$, where R is the radius of the wheel, and T is the pendulum period of the bell for small oscillations.

Table 5 – Sensor Pulse Durations – Liverpool Cathedral

Bell	Period *T* (s)	Wheel Radius *R* (m)	Estimated Wheel Rim Speed (m/s)	Theoretical Pulse Duration (ms)	Observed Pulse Duration (Range) (ms)
1	1.60	0.81	6.38	3.9	5.7 (5.6 - 6.2)
2	1.60	0.84	6.58	3.8	6.2 (6.1 - 6.3)
3	1.60	0.86	6.78	3.7	6.0 (5.9 - 6.1)
4	1.70	0.86	6.38	3.9	6.2 (6.0 - 6.3)
5	1.75	0.89	6.38	3.9	6.4 (6.3 - 7.2)
6	1.80	0.89	6.21	4.0	5.9 (5.7 - 6.2)
7	2.00	0.99	6.22	4.0	6.3 (5.8 - 6.4)
8	2.05	1.07	6.54	3.8	7.7 (7.5 - 7.8)
9	2.25	1.14	6.38	3.9	6.6 (6.4 - 6.7)
10	2.40	1.22	6.38	3.9	6.5 (6.1 - 6.7)
11	2.40	1.32	6.92	3.6	6.7 (6.5 - 6.9)
12	2.60	1.50	7.24	3.5	6.1 (6.0 - 6.3)

The effect of the "overlap" effect described in the main text on the observed pulse durations is evident when the theoretical and observed duration figures are compared.

Similar measurements of period and wheel radius have been made on a number of other smaller bells, and these yield similar results. This suggests that, in this regard at least, the Cathedral bells are unexceptional and the sensing approach adopted may be re-used elsewhere.

The following table shows the results of these investigations. No simulators are currently fitted to these bells and no actual pulse duration measurements have been made.

Table 6 – Theoretical Sensor Pulse Durations – Other Towers

Bell	Period *T* (s)	Wheel Radius *R* (m)	Estimated Wheel Rim Speed (m/s)	Theoretical Pulse Duration (ms)
St John the Baptist, Tuebrook				
1	1.60	0.80	6.28	4.0
5	1.70	0.88	6.47	3.9
8	2.00	0.98	6.16	4.1
St Helen, Sefton				
1	1.50	0.71	5.95	4.2
4	1.60	0.77	6.01	4.2
7	1.80	0.87	6.04	4.1

Appendix D: CLI Command Reference

The following table lists the additional CLI commands supported by the Simulator Interface firmware, and their functions, as of v2.4.

- Not all commands are available in v1.3 of the the One Bell Simulator Interface firmware. Commands not available are marked †.
- The behaviour of some commands is different in v1.3 of the One Bell Simulator Interface firmware. These commands are marked ‡.

Table 7 – CLI Command Reference

Command	Function
Q	Set simulator type quirks mode (A/B/R/V/X) Configure the interface for Abel, Beltower, Ringleader, Virtual Belfry or Generic simulators. At present the only function of this command is to disable the diagnostic LEDs for Beltower, to circumvent[29] timing issues observed with delay timer configuration in Beltower 2015. See also command "S".
#	Set the number of Active Channels (1 –> 12) † This command prompts for the number of input channels used by the Simulator Interface (always starting at 1), and disables input on any unused channels. See also command "S".
B	Set the de-bounce timer (1ms -> 20ms) ‡ This command prompts for the value of the de-bounce time, in milliseconds. See also command "S".
E	Enable/disable sensors † This command prompts for the enable mask bit to be set or unset for each input channel. This determines which channels will generate output, and can be used for example to bypass a faulty Sensor Head. See also command "S".
S	Save settings to EEPROM ‡ This command saves the value set by the "Q", "#", "B" and "E" commands to non-volatile memory. If Debug Mode is enabled, the Debug Mask ("M") is also saved.
P	Set the serial port speed † The serial port speed may be set to 2400, 4800 or 9600bps with this command. The new speed is stored in EEPROM and an interface reset is required to make it active. This command is provided for debugging use only, all tested Simulator Software packages support 2400bps only.
D	Turn debug mode ON, or set debug flags ‡ If Debug Mode is off, this command turns it on. If Debug Mode is on, this command prompts the user to enable or disable each Debug Flag. These are documented in Table 2 above. This output is intended to be accessed via a terminal emulator, and is not suitable for use by the Simulator. Debug mode should be disabled before starting the

[29] A better solution to this problem would be to make LED signalling an asynchronous function of the Simulator Interface firmware.

	Simulator Software Package.
d	Turn debug mode OFF
M	Set or unset the debug mask bits † This command is available only when Debug Mode is turned on, and prompts for the debug mask bit to be set or unset for each input channel. This determines which channels will generate debug output, and can be used to reduce the volume of debug messages when troubleshooting a specific channel or Sensor Head. See also command "S".
Z	Set Active timers to default (500ms) (Debug Mode only) This command is available only when Debug Mode is turned on, and sets all the bell delay timers to a default 500ms. This is useful for Sensor Head calibration. The non-volatile EEPROM timers are not changed, and resetting the interface will cause the active timers to be restored back to the EEPROM values.
0 … 9	Print debug markers (Debug Modes only) This command is available only when Debug Mode is turned on, and sends the text "DEBUG MARKER *N*" to the serial port, where *N* = 0 to 9. This is useful for identifying areas of interest in the output when logging debug output with a terminal emulator.
H	Display CLI help text †
T	Enable test mode †
L	Display CLI debug mode help text (Debug Mode only) †
?	Display current settings This command sends a detailed set of configuration and operational values to the serial port for review.

Appendix E: Diagnostic LED Codes

The following tables list the meanings of the codes shown by the flashes of the red and yellow diagnostic LEDs on the Simulator Interface. This list is correct as of firmware v2.4.

- The One Bell Simulator Interface firmware uses single LED (by convention coloured red) for all diagnostic codes.
- Not all codes are used by v1.3 of the One Bell Simulator Interface firmware. Commands not used are marked †.
- Both LEDs flash slowly in unison when test mode is enabled †.

Red LED

The following table lists the signal codes displayed on the red LED of the Simulator Interface.

Table 8 – Red LED Signal Codes

Long	Short	Meaning
1	0	The red LED gives one long flash every time the Sensor Head connected to Channel 1 is triggered. The LED lights when the signal from the Sensor Head is received, and extinguishes when the corresponding signal is sent to the Simulator over the Simulator Data Connection. (The length of this flash is therefore equal to the setting of the Channel 1 delay timer.) The *Debug Show LED* debug flag enables this behaviour for all channels for which debugging has been selected.

Yellow LED

The following table lists the signal codes displayed on the yellow LED of the Simulator Interface.

Table 9 – Yellow LED Signal Codes

Long	Short	Meaning	CLI Command
N	*M*	Announces the firmware version on power-on.	-
-	-	On after power-on: Port speed is not 2400bps, or one or more sensors are disabled. †	-
0	1	CLI key press acknowledgement.	?, H, L, 0-9, D (flags)
1	1	MBI Protocol command 0xFD received.	-
1	2	MBI Protocol command 0xFE received.	-
1	3	MBI Protocol new delay timers (and terminator) received.	-
2	1	Simulator Type set[30].	Q
2	2	Number of active channels set. †	#
2	3	De-bounce timer set. †	B
2	4	Enable mask set. †	E

[30] In the current firmware version this code is never actually displayed if Beltower is selected, as the action of setting Beltower mode disables all LED activity for timing reasons, and this happens before the code is signalled.

2	5	Settings saved to EEPROM.	S
2	6	Debug mode on.	D (enabling)
2	7	Debug mode off.	d
2	8	Default active delay timers set.	Z
2	9	Serial port speed set in EEPROM. †	P
2	10	Debug mask set. †	M
3	0	Invalid character or timeout on serial interface, data discarded.	-

Appendix F: Useful Links

Table 10 – Useful Links

Description		Link
Software		
GitHub	Software Repository	https://github.com/Simulators
Abel	Simulator Software	http://www.abelsim.co.uk
Beltower	Simulator Software	http://www.beltower.co.uk
Virtual Belfry	Simulator Software	http://www.belfryware.com
Hardware		
OSH Park	PCBs	https://oshpark.com/profiles/AndrewIC
David Bagley	MBI, Sensors	http://www.ringing.demon.co.uk
Belfree	Wireless Ringing System	http://belfree.co.uk/
Atmel	MIcrocontrollers	http://www.atmel.com
Maxim	Serial Line Drivers	http://www.maximintegrated.com
Information		
Aidan Hedley	Bellringing Simulation	http://www.aidanhedley.me.uk
John Norris	“When does a bell strike?”	http://www.jrnorris.co.uk/strike.html
Frank King	Equations of Motion of a Free Bell and Clapper	http://www.cl.cam.ac.uk/~fhk1/Bells/equations.pdf
Tools		
Arduino	Electronic Prototyping	http://arduino.cc
ATtiny on Arduino	Electronic Prototyping	https://github.com/damellis/attiny
PuTTY	Free Terminal Emulator	http://www.chiark.greenend.org.uk/~sgtatham/putty/
PortMon	Serial Port Monitor	https://technet.microsoft.com/en-us/library/bb896644.aspx
FTDI	USB-Serial Adapter Drivers	http://www.ftdichip.com/Drivers/VCP.htm
Prolific	USB-Serial Adapter Drivers	http://www.prolific.com.tw/US/ShowProduct.aspx?p_id=225&pcid=41
Audacity	Sound Editing Package	http://audacity.sourceforge.net/

Appendix G: A Quarter Peal of Cambridge Surprise Minor

The following chart was constructed from debug output generated by a prototype Simulator Interface during development of the Liverpool Cathedral Simulator, and shows the relative time between blows of a sensor on the 3rd, which was being rung as the treble to a quarter peal of Cambridge Surprise Minor of the light (24cwt) six. This output showed that triggering of the standard Sensor Head was reliable, with no missed signals (which would have been indicated as very high values) or spurious triggers (very low values).

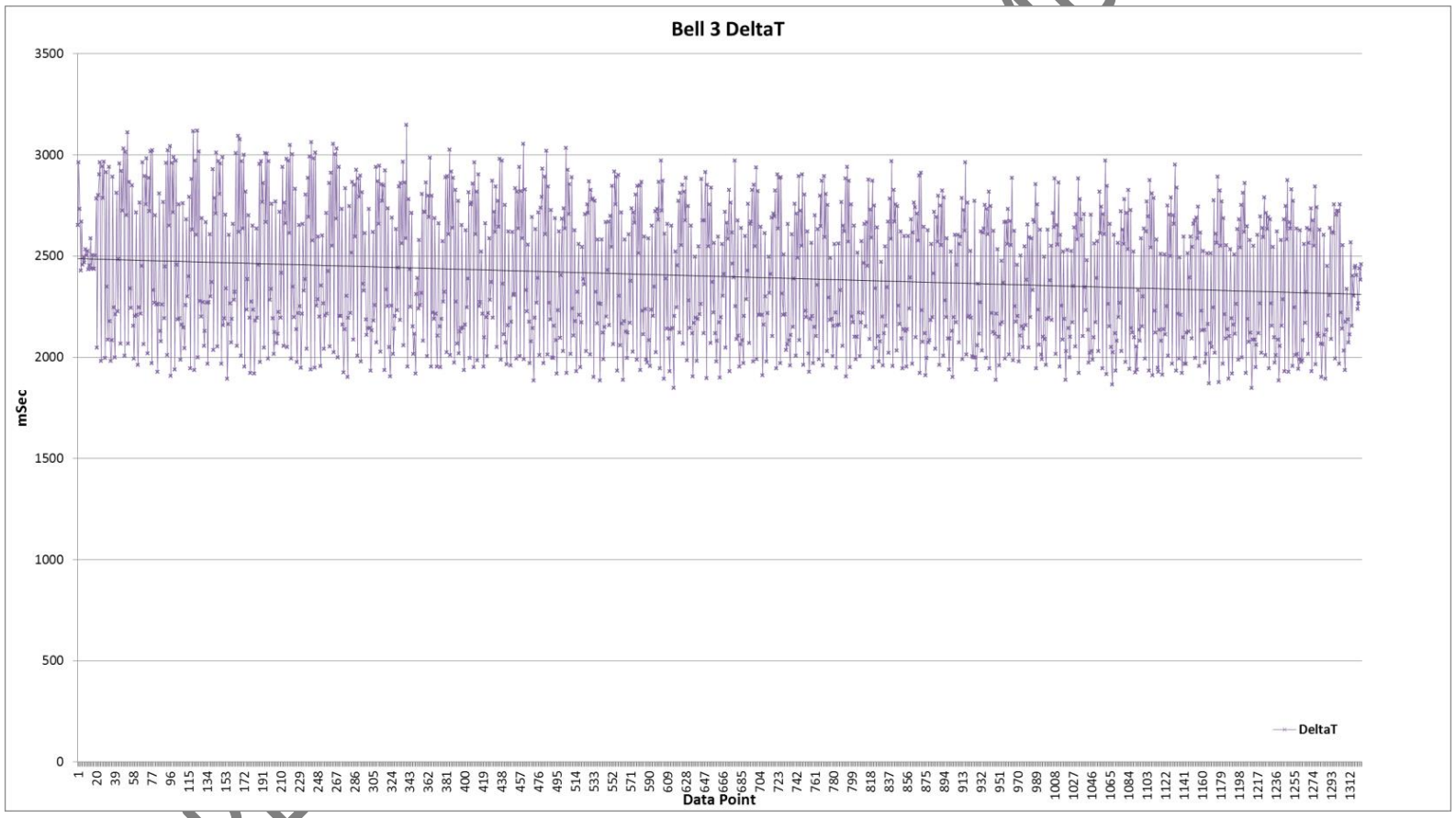


Figure 74 – Quarter Peal Sensor Head Test Timings