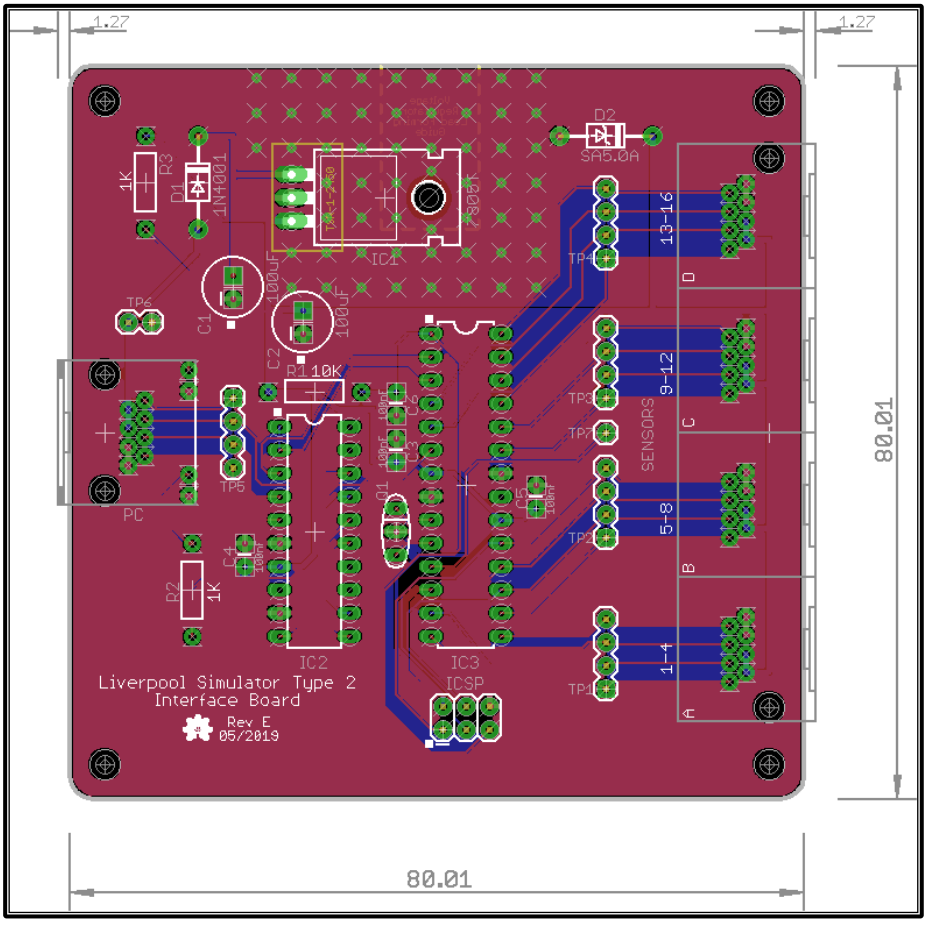
Type 2 Liverpool Ringing Simulator

06 – Technical Reference Guide



Author: Andrew Instone-Cowie
Date: 28 December 2019
Version: 1.0

Contents

Index of Figures

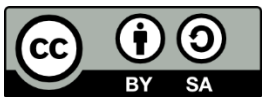
Index of Tables

Document History

Version	Author	Date	Changes
0.1	A J Instone-Cowie	26/12/2019	First Draft.
1.0	A J Instone-Cowie	28/12/2019	First Release.

Cover image: Type 2 Simulator Interface PCB Design.

Licence



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

Documentation Map

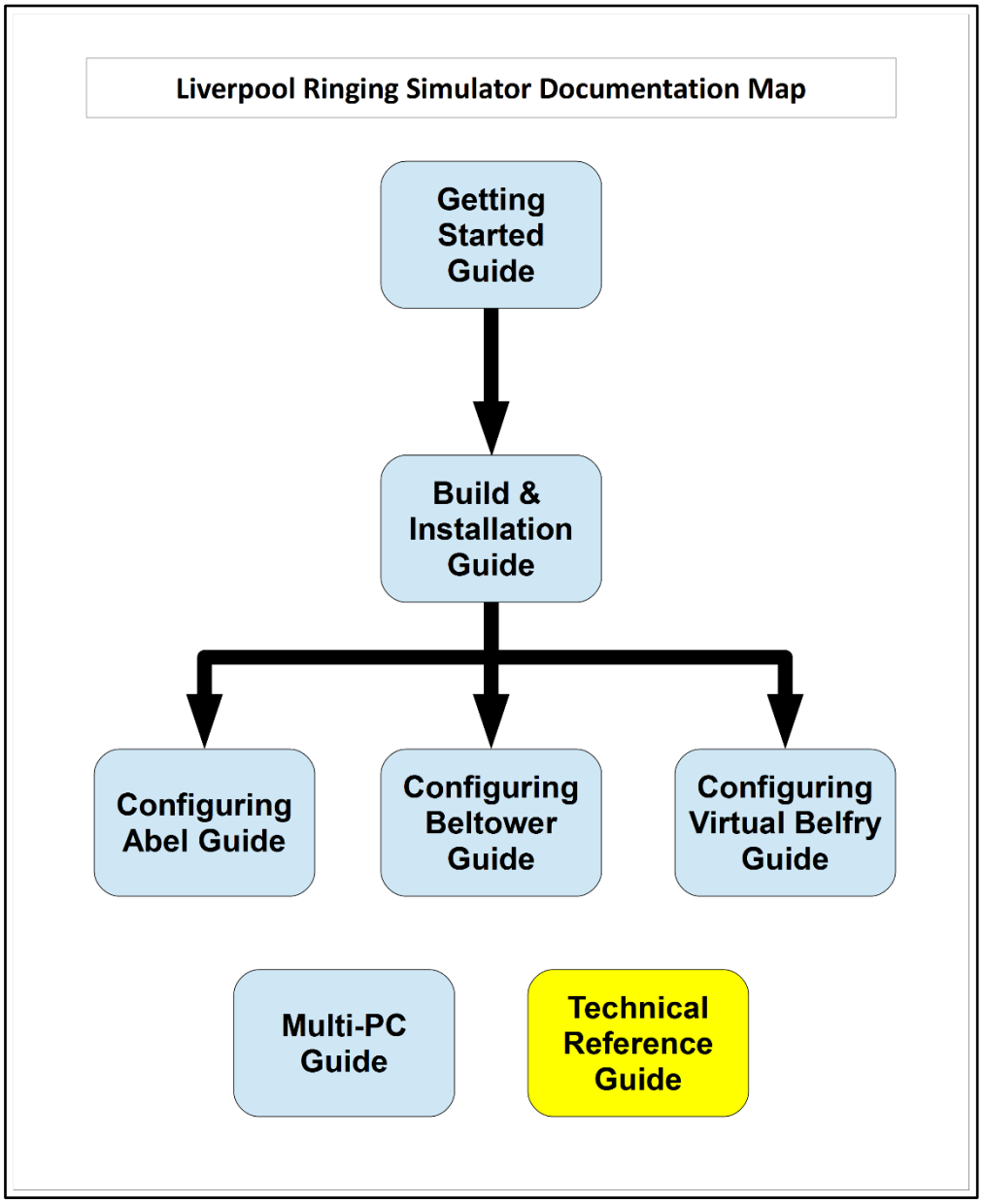


Figure 1 – Documentation Map

About This Guide

The Type 2 Liverpool Ringing Simulator allows sensors, attached to one or more real tower bells or teaching dumb bells, to be connected to a computer Simulator Software Package such as Abel[2], Beltower[3] or Virtual Belfry[4]. This allows you to extend and augment the teaching and practice opportunities in your tower.

This ***Technical Reference Guide*** contains background information on the design of the simulator hardware and firmware, and other reference material which may be useful.

Configuration guides for the main Simulator Software Packages are available separately, as are the detailed ***Build & Installation Guide*** and ***Multi-PC Guide***.

Please note that while advice and guidance is available, this is a Build-it-Yourself project. No pre-built hardware is available.

[2] http://www.abelsim.co.uk/
[3] http://www.beltower.co.uk/
[4] http://www.belfryware.com/

Typical Type 2 Simulator Installations

Basic Configuration

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

Multiple Sensor Modules in the belfry, one per bell, are connected to a Simulator Interface Module. A single data cable transmits the aggregated signals from the Simulator Interface Module to the Simulator PC in the ringing room. The same cable feeds power from a low voltage power supply in the ringing room, via a Power Module, back up to the Simulator Interface to power both Interface and Sensor Modules. The Type 2 simulator supports up to 16 sensors (only 12 shown on the diagram to save space).

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.

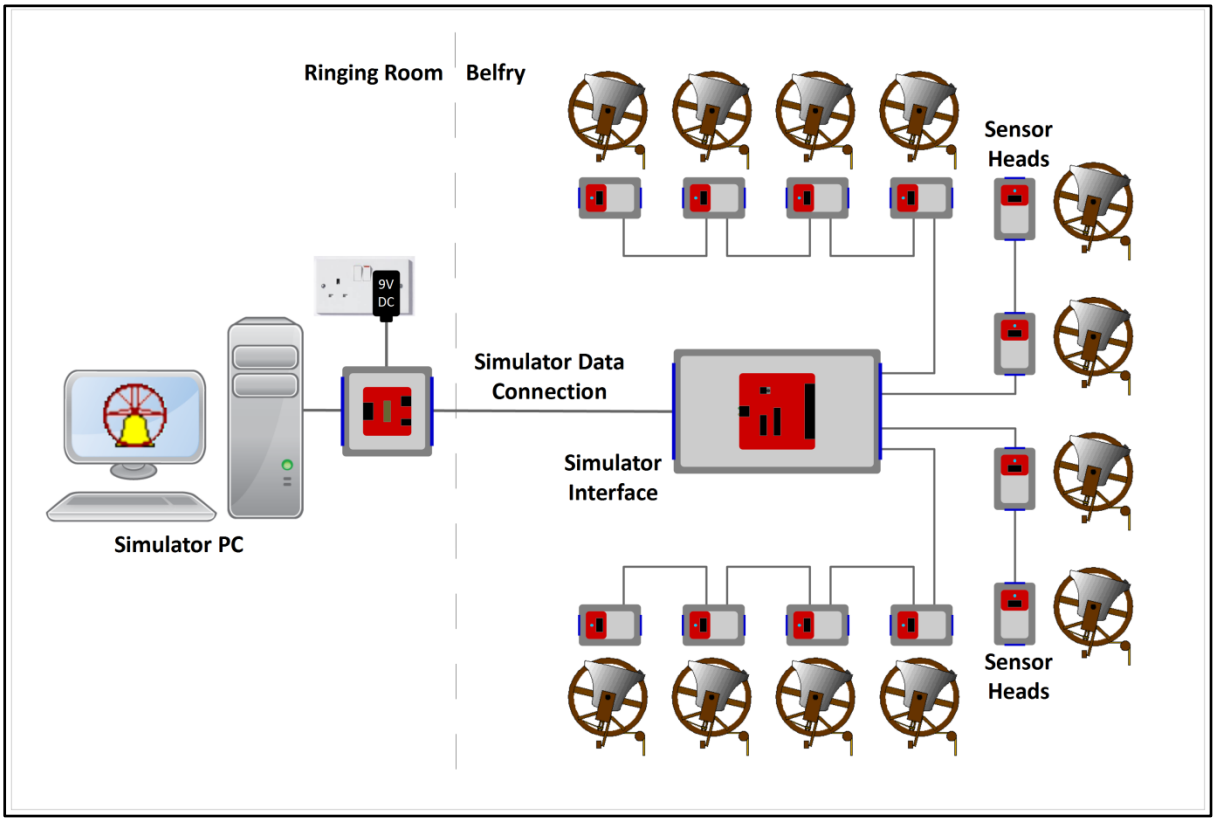


Figure 2 – Simulator General Arrangement

Second PC Module

As an option, multiple Simulator PCs may be used concurrently, using either a Second PC Module, or the Basic Serial Splitter Module.

A second Simulator PC can be connected via a Second PC Module. This module utilises the second transmitter in the Simulator Interface MAX233 RS-232 serial line driver IC, and a spare core in the Power/Data cable, to provide a separate data feed to the Second PC.

A typical configuration with a Second PC Module is illustrated in the following diagram (only a subset of sensors are shown to save space):

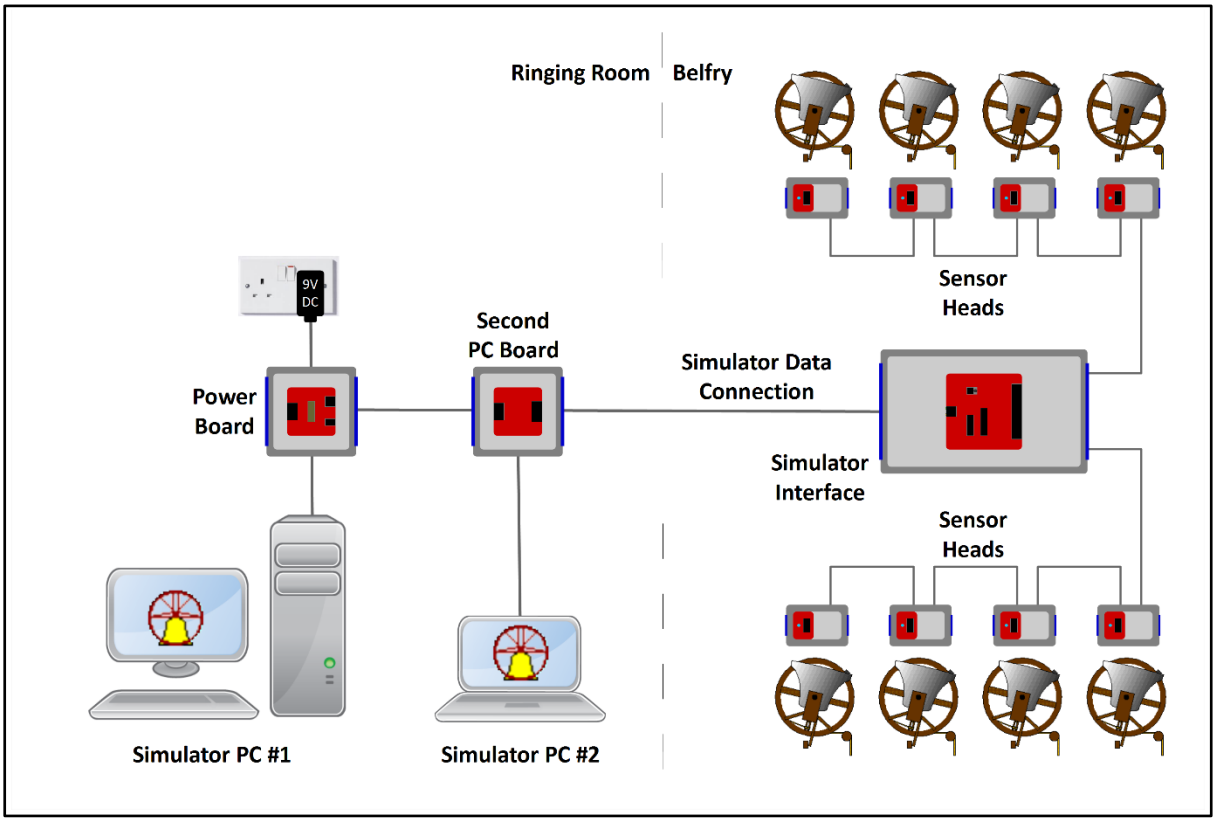


Figure 3 – Second PC Module General Arrangement

Basic Serial Splitter Module

The Basic Serial Splitter Module uses additional RS-232 receiver and line driver ICs to copy the stream of signals from the Simulator Interface to multiple PCs. The minimum configuration of the Basic Serial Splitter (the "Master Board") supports up to four PCs, but this can be linked to a second "Expander Board" which supports four more PCs.

Two Basic Serial Splitter Modules can be daisy-chained between the Power Module and the Simulator Interface Module on RJ45 cables, supporting a maximum of 16 Simulator PCs. The maximum configuration of two daisy-chained Basic Serial Splitters is illustrated in the following diagram (all sensors are omitted to save space):

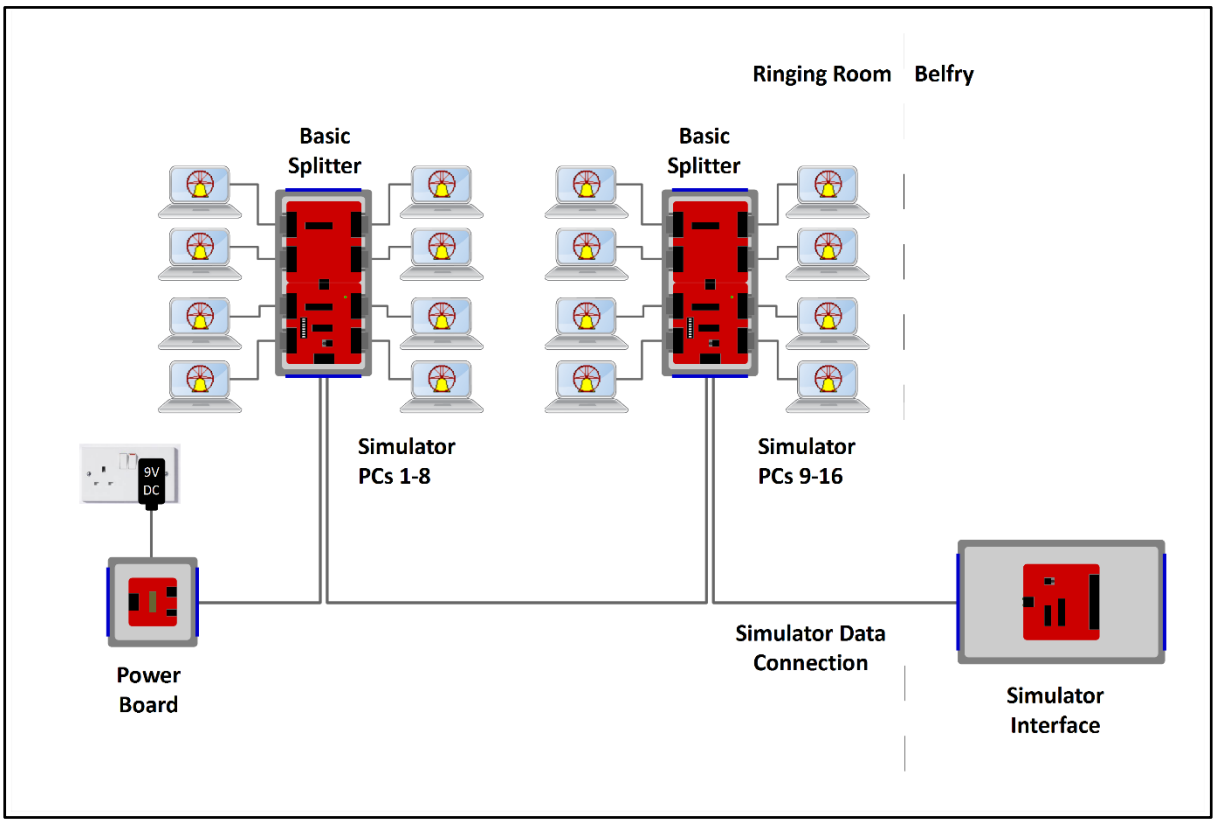


Figure 4 – Basic Serial Splitter Modules General Arrangement

All Simulator PCs receive a read-only copy of the same signals from the Simulator Interface. Each PC runs its own copy of the Simulator Software Package (or even of different software packages), and the Simulator Software on each PC is configured to filter out unwanted signals.

Any one PC connected to the Basic Serial Splitter can be used to send configuration data to the Simulator Interface, selectable by a switch on the splitter. (If two Basic Serial Splitters are daisy-chained together, only PCs connected to the Splitter nearest to the Power module can be used to configure the Interface.)

Simulator Overview

Bell Sensing Methods

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.

- Mechanical switches or contacts suffer from reliability problems, including mechanical fatigue and environmental damage (e.g. corrosion or ingress of dirt), and are not now widely used. Mechanical switches also typically suffer from a high degree of contact "bounce".
- Electro-Magnetic sensors, typically consisting of a glass-encapsulated reed switch attached to the bell frame and a small magnet attached to the wheel shroud, have also been used, but typically require very close alignment of switch and magnet. They can also suffer from fatigue and contact bounce.
- Inductive sensors, which have a fixed detector circuit attached to the bell frame, and a wire coil fitted to the wheel shroud, have also been used. In principle the absence of moving parts should make these a very reliable option. However a key component of the original design[5] is no longer available, and parts for an alternative design[6] are described as difficult to obtain.
- Aidan Hedley has produced a sensor design[7] using a Honeywell magneto-resistive sensor, activated by a small, powerful rare earth magnet mounted on the wheel shroud. The standard magneto-resistive sensor used by the Type 2 Liverpool Ringing Simulator is derived from this design.
- A wireless, accelerometer-based sensor has been developed and prototyped as the Belfree simulator[8]. This sensor is mounted on the wheel of the bell and communicates with the Simulator PC via a radio link. At the time of writing these not currently available.
- Optical sensor heads using visible light typically consist of a light source, usually a fixed red Light Emitting Diode (LED), mounted parallel to a photo-diode or photo-transistor detector, in an enclosure attached to the bell frame; and a reflector mounted on the wheel shroud. Visible light optical sensors are widely used, are very reliable, and this is the approach adopted by the Bagley MBI sensor heads[9]. Their use can be problematic in areas with high ambient light levels. Other designs include additional pulse-shaping or timing circuitry[10].
- Optical sensor heads using infra-red light are similar in general design, but have reduced susceptibility to ambient visible light levels. A sensor design using a low cost commercially available modulating infra-red detector has been developed as an alternative sensor for the Type 2 Simulator.

[5] http://www.abelsim.co.uk/doc/tbellex2.htm
[6] http://www.gremlyn.plus.com/ahme/prox_sen.html
[7] http://www.gremlyn.plus.com/ahme/mag_sen.html
[8] http://belfree.co.uk/
[9] http://www.ringing.demon.co.uk/sensors/sensors.htm
[10] http://www.abelsim.co.uk/doc/tbellex1.htm

Single vs Dual Trigger

There are generally two approaches used by simulators when detecting signals from sensors:

- The dual trigger approach uses two triggers (for example, optical reflectors) on the shroud of the wheel of each bell, one of which triggers at the moment that the bell would strike at handstroke, the other at the moment that the bell would strike at backstroke.
- The single trigger approach uses one trigger, which triggers as the bell passes through the bottom dead centre (BDC) 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.

The Liverpool Ringing Simulator project adopts the single trigger approach, on the grounds of simplicity of installation and compatibility with other similar systems. This is discussed in more detail in a later section of this guide.

Dual Triggers

The dual trigger approach uses two triggers (for example, optical reflectors or magnets) on the shroud of the wheel of each bell, one of which triggers 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 (from the trigger travelling past the sensor as the bell is on its way up to the balance), and then ignores the next signal pulse (as the first trigger travels back past the sensor as the bell is on its way down), and so on.

This approach is illustrated in the following diagrams. In Figure 5 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 6 shows the opposite stroke as the bell rises towards the backstroke position and passes through the handstroke strike point.

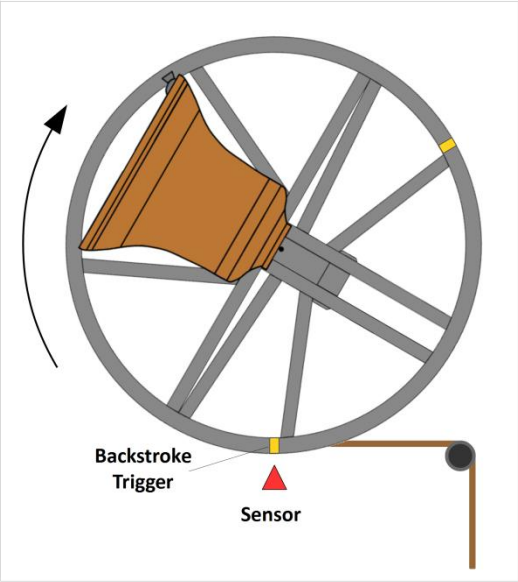


Figure 5 – Dual Triggers – Backstroke Strike Point

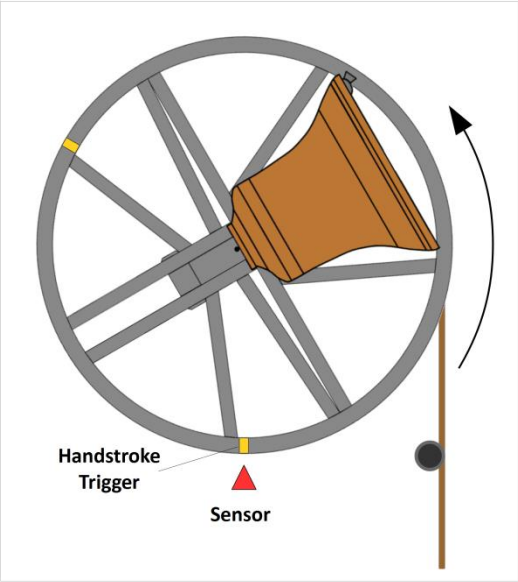


Figure 6 – Dual Triggers – Handstroke Strike Point

The sensor is shown at the BDC position only for clarity. The sensor may be located anywhere around the wheel, provided that 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 single trigger approach uses one trigger, which triggers as the bell passes through the bottom dead centre (BDC) 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 7 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 8 shows the opposite stroke as the bell rises towards the backstroke position and passes through the handstroke strike point.

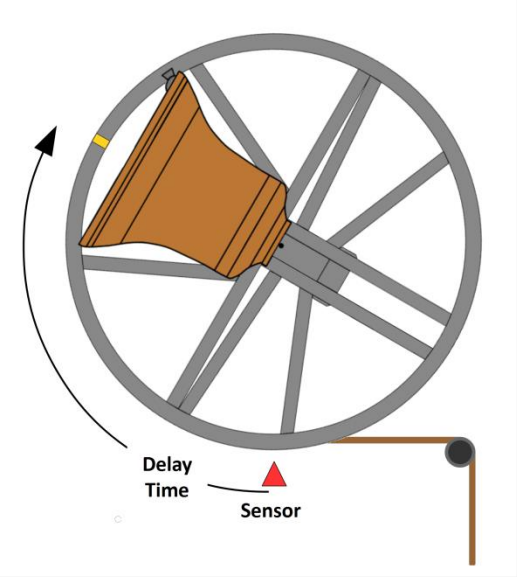


Figure 7 – Single Trigger – Backstroke Strike Point

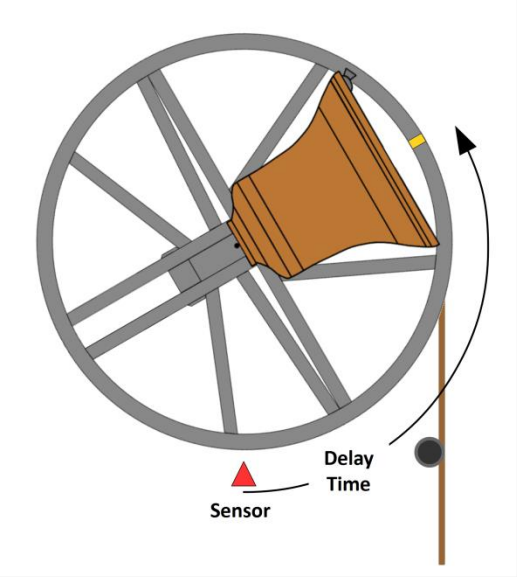


Figure 8 – Single Trigger – Handstroke Strike Point

The sensor is again shown at BDC only for clarity. The sensor may be located anywhere around the wheel, provided that 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[11] 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.

[11] http://www.jrnorris.co.uk/strike.html

The Type 2 Liverpool Ringing Simulator adopts the single trigger approach, on the grounds of simplicity of installation, and compatibility with other similar systems.

Simulator Interface Module Hardware

Overview

The Simulator Interface Module hardware consists of three main functional blocks:

- A 5V regulated power supply, which provides power for the interface electronics and for the sensor.
- A digital microcontroller, which detects incoming signals from the sensors, and then sends aggregated signals to the Simulator PC in the form of a stream of ASCII characters.
- A RS-232 serial line driver, which converts the TTL-level signals from the microcontroller to higher voltage RS-232 signals, for transmission over longer distances to the Simulator PC.

All the components of these functional blocks are mounted on a single custom PCB. The same core hardware (and firmware) can connect to any number of Sensors Modules from one to 16.

Hardware Options

Over recent years several low-cost electronic prototyping platforms have been brought to the market, with a number of different architectures, and suitable for a wide range of experimental and embedded functions. These include single board computers running feature-rich operating systems, such as the Raspberry Pi and Beagle Board, and simpler microcontroller-based products such as the Arduino family, PICAXE, ARM Cortex and others.

New devices and variants are released frequently.

The Arduino Uno development platform (based on the Microchip (formerly Atmel) ATmega328P microcontroller[12]) was originally selected as the prototyping platform for the Liverpool Cathedral Simulator project, on the grounds of:

- Simplicity & low cost
- Established user base & wide community support
- Wide availability of additional prototyping peripheral hardware (“shields”)

The use of the same microcontroller family has been continued for the Type 2 Liverpool Ringing Simulator, allowing the same well-supported and freely available tool chain to be used for code development and deployment.

Hardware Interrupts

The choice of the ATmega328P imposes an important constraint on the firmware design for the Simulator Interface, in that the microcontroller has limited support for hardware interrupt inputs. This is described in more detail in a later section of this guide.

[12] https://www.microchip.com/wwwproducts/en/atmega328p

Microcontroller Hardware

The Simulator Interface is based on the same Microchip (formerly Atmel) ATmega328P microcontroller as the Arduino Uno used in the first prototype interfaces. Using the same microcontroller makes it very easy to port the Simulator Interface firmware to the new hardware.

- The Simulator Interface[13] makes use of an external 8MHz resonator as a clock source. Although this is less accurate than a crystal oscillator, it is still adequate for the purposes of the Simulator Interface, and this reduces the overall component count and complexity of the interface board by omitting crystal load capacitors.
- Although the Simulator Interface uses the ATmega328P microcontroller and development tool chain, it does not use the Arduino boot loader program. The ATmega328P is programmed via an ICSP header using a programmer (the *Arduino-as-ISP* method is suggested and is described in more detail in the ***Build & Installation Guide***.)

Serial Line Driver

- The Simulator Interface uses the Maxim MAX233 RS-232 serial line driver[14]. The MAX233 has the advantage over other line driver ICs of requiring no external charge pump capacitors, also reducing the overall component count. It also provides two RS-232 transmitters, simplifying support for multiple Simulator PCs.

Transient Voltage Suppression

The simulator uses Transient Voltage Suppression (TVS) diodes to provide a degree of protection against induced over-voltages on the power supply lines, for example resulting from local lightning strikes (similar devices are included in the Sensor Module designs).

Bi-directional TVS diodes are used on the RS-232 signal lines, and uni-directional TVS diodes are used on logic and power supply lines.

[13] From Rev E onwards.
[14] http://www.maximintegrated.com/en/products/interface/transceivers/MAX233.html

Microcontroller Configuration

Microcontroller I/O Pins

The following table shows the ATmega328P pin assignments for the Type 2 Simulator Interface:

Table 1 – Microcontroller I/O Pin Assignments

Arduino I/O Pin[15]	ATmega328P Physical Pin	Function
0	2	Serial Port Receive
1	3	Serial Port Transmit
2	4	Sensor #12 (T)
3	5	Sensor #10 (0)
4	6	Diagnostic LED
5	11	CRO Timing Pin (enabled in development code only)
6	12	Sensor #1
7	13	Sensor #2
8	14	Sensor #3
9	15	Sensor #4
10	16	Sensor #5
11	17	Sensor #6
12	18	Sensor #7
13	19	Sensor #8
14 (A0)	23	Sensor #9
15 (A1)	24	Sensor #11 (E)
16 (A2)	25	Sensor #13 (A)
17 (A3)	26	Sensor #14 (B)
18 (A4)	27	Sensor #15 (C)
19 (A5)	28	Sensor #16 (D)

[15] Pin numbers as referenced in the Arduino programming environment.

Microcontroller Fuse Settings

The ATmega328P microcontroller has several configuration registers known as *fuses*. These are used to control the behaviour of the microcontroller[16]. Fuse values are contained in the file *boards.txt* described in the section on uploading firmware below, and are only written during the *burn bootloader* phase of programming.

Warning: Setting incorrect fuse values can render the microcontroller unusable or prevent further reprogramming without specialist hardware.

The appropriate set of fuses is selected in the Arduino IDE by selecting the correct board type in the Liverpool Ringing Simulator Boards set.

Table 2 – ATmega328P Fuse Settings – Board Rev A to D

Fuse	Value	Notes
Low	0xE2	8MHz internal oscillator, maximum clock startup delay. (SUT0, CKSEL3, CKSEL2, CKSEL0)
High	0XDF	Enable serial programming, boot reset vector disabled. (SPIEN)
Extended	0xFD	Brown-out detection level 1 (2.7V). (BODLEVEL1)

For more details of these fuse settings and their functions, refer to the ATmega328P data sheet, or see the online fuse calculator at: http://eleccelerator.com/fusecalc/fusecalc.php?chip=atmega328p&LOW=E2&HIGH=DF&EXTENDED=FD.

Table 3 – ATmega328P Fuse Settings – Board Rev E onwards

Fuse	Value	Notes
Low	0xCF	8MHz external resonator, maximum clock startup delay. (SUT0, SUT1)
High	0XDF	Enable serial programming, boot reset vector disabled. (SPIEN)
Extended	0xFD	Brown-out detection level 1 (2.7V). (BODLEVEL1)

For more details of these fuse settings and their functions, refer to the ATmega328P data sheet, or see the online fuse calculator at: http://eleccelerator.com/fusecalc/fusecalc.php?chip=atmega328p&LOW=CF&HIGH=DF&EXTENDED=FD.

Note that unlike the Type 1 Simulator Interface, EEPROM contents are cleared on firmware load (high fuse bit 3, EESAVE, is not set, giving 0xDF instead of 0xD7). This is to be confident of having a workable default configuration even if the interface has not yet been configured by the user.

[16] Fuse tutorial: http://www.martyncurrey.com/arduino-atmega-328p-fuse-settings/

Power Supply

DC power is supplied to the Simulator Interface over the Power/Data Cable at a higher voltage than required, to overcome the effects of losses due to the resistance of the Power/Data Cable. Incoming power is fed to a conventional linear 5V regulator on the Simulator Interface PCB. An alternative to a conventional linear regulator is described in the ***Build & Installation Guide***, for installations with higher current requirements.

Typical Current Requirements

The Simulator Interface Module and Sensor Modules all operate at 5V DC.

- The Simulator Interface current consumption is approximately 25mA.
- The current consumption of the standard Magneto-Resistive Sensor Modules is approximately 8mA per module in the triggered state (i.e. with the LED lit).
- The current consumption of the Infra-Red Sensor Modules is approximately 25mA per module in the triggered state.
- The total current requirement with a full set of 16 Magneto-Resistive Sensor Modules is therefore approximately 153mA.
- The total current requirement with a full set of 16 Infra-Red Sensor Modules is approximately 425mA.

These values are well below the 1A maximum current rating of the specified 5V voltage regulator, however heatsinking may be required for configurations with many Infra-Red Sensor Modules, as noted below.

Microcontroller Package Ratings

The ATmega328P microcontroller has a source/sink current limit of 40mA per pin, and a total device package current limit of 200mA. The majority of the microcontroller pins will be connected to sensor inputs, and these were measured to sink approximately 130 µA each when triggered, so neither of the microcontroller limits should present a problem. Other pins drive the diagnostic LED and the RS-232 line driver, and in total these do not draw more than a few milliamps.

Voltage Regulator

The Simulator Interface uses a standard linear voltage regulator to provide a regulated 5V DC supply. The voltage regulator requires at least 7V DC at its input pin in order to maintain a stable 5V DC output. The voltage regulator is fed via a polarity protection diode, which drops approximately 0.7V, giving a minimum input voltage requirement of 7.7V at the Simulator Interface input.

Cable Voltage Drop

CAT5 Ethernet cable is specified[17] as having a DC loop resistance of <0.188 Ω/m.

The effective resistance of the power cores of a 25m cable (considering that the simulator configuration doubles up both power and ground cores) is therefore estimated to be approximately 2.35Ω. At a peak supply current of 425mA, this would result in a voltage drop of approximately 1V along the 25m cable.

[17] https://www.macinstallations.com/cat-5-cabling-explained/

Taking into account the voltage drop across the polarity protection diode, the theoretical minimum supply voltage to the cable is approximately 8.7V in order for the voltage regulator to maintain a stable 5V DC output. In practice a multi-voltage DC plug-in power supply unit rated at 1000mA with the output set to 9V has been found adequate for a cable run up to 25m.

For exceptionally long cable runs a 12V DC supply could be used, but there is no advantage to doing so for shorter runs, and the heat dissipation of the voltage regulator will increase, with a requirement for additional heatsinking.

Heat Dissipation

At peak load, with a full complement of 16 Infra-Red Sensor Modules, the voltage regulator would be dissipating a minimum[18] of approximately 0.85W. The power dissipation will be higher with shorter cable runs or a higher supply voltage. The data sheet for the voltage regulator indicates that this will require a heatsink.

An alternative approach, described in the ***Build & Installation Guide***, is to replace the linear voltage regulator with a switched buck regulator. These are available as drop-in replacements for the standard TO-220 regulator package and eliminate the need for a heatsink altogether.

Fusing

A fuse rated at 800mA has been included in the design of the Type 2 Simulator. This is located on the Power Module board.

[18] Assuming the minimum required input voltage of 7V at the voltage regulator.

Hardware Compatibility

The Type 2 Liverpool Ringing Simulator has been designed to be compatible, as far as possible, with existing simulator hardware, to allow operation with unmodified Simulator Software Packages which support the MBI protocol. This is described in more detail in later section of this guide. The following hardware 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 inputs 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[19].

Unlike the Type 1 design, the simplified cabling used by the Type 2 Simulator Interface means that the sensor inputs do not use the same 4-pin GX16-4 panel mount connectors as the Bagley MBI. Integrating existing sensors into the Type 2 Simulator may be achieved using the Generic Sensor module described in the ***Build & Installation Guide***.

[19] Sensors which operate with reverse logic (normally LOW, going HIGH when triggered) may be accommodated by amending the firmware to look for LOW to HIGH transitions.

Connector Pin-Outs

Sensor Module Connector

The Sensor Modules use RJ45 connectors for the downstream cabling connection towards the Simulator Interface, and upstream connections to other Sensor Modules in the same chain.

- Each Sensor Module presents its output signal on pin 8 of the downstream *Interface* connector. Each Sensor Module takes any upstream sensor signals appearing on pins 4, 6 and 8 of the *Next Sensor* connector, and loops them through to pins 2, 4 and 6 respectively of the *Interface* connector.
- Pin 2 of the Next Sensor connector is not used, and hence connecting more than four sensors in one chain is not possible.
- Power pins are doubled up to mitigate the effects of voltage drop in longer cable runs. Pins 5 and 7 are used for the +5V supply to sensors, and pins 1 and 3 are 0V. These pins are looped through between the *Interface* and *Next Sensor* connectors.

The wiring of the female PCB connector is shown in the following diagram:

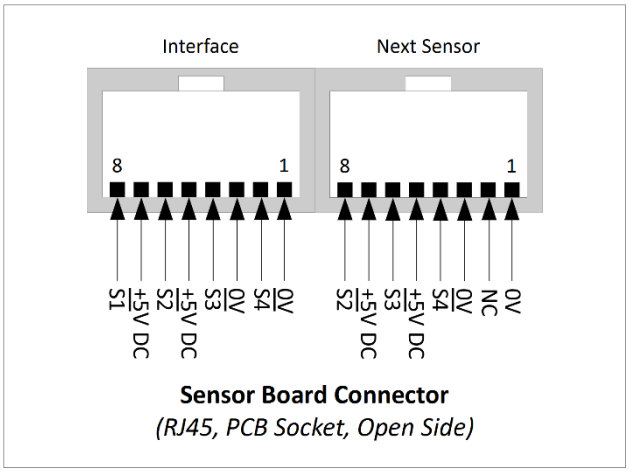


Figure 9 – Sensor Module Connector

The daisy-chain inter-connection of the sensors is illustrated in the following diagram:

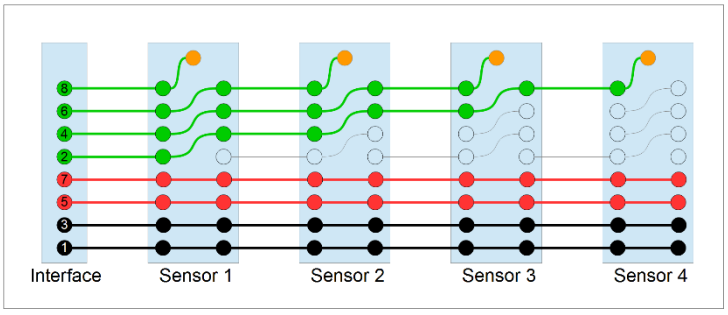


Figure 10 – Sensor Chain Inter-Connection

Simulator Interface Module Sensor Connector

The Simulator Interface Module uses RJ45 connectors for the upstream cabling connection to the Sensor Modules. A 4-gang connector is used, allowing the connection of up to 16 sensors.

The wiring of each female PCB connector is shown in the following diagram:

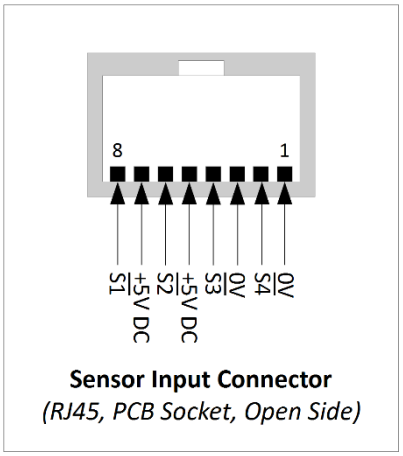


Figure 11 – Simulator Interface Module Sensor Connector

The layout of the 4-gang RJ45 PCB connector is shown in the following diagram:

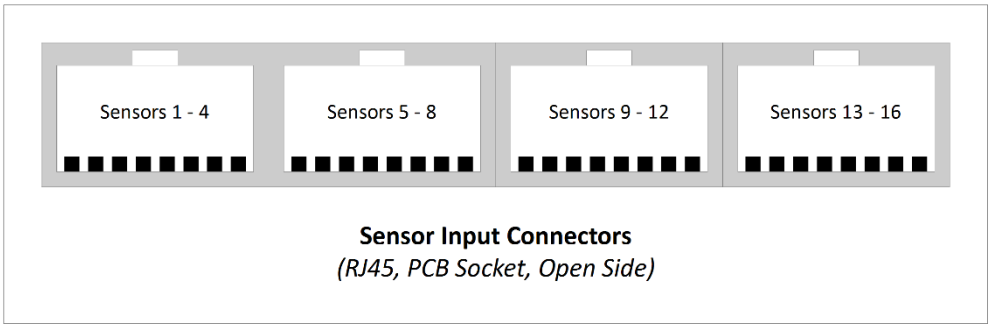


Figure 12 – Simulator Interface 4-Gang Connector

Simulator Interface Module Power/Data Connector

The Simulator Interface Module uses an RJ45 connector for the downstream cabling connection to the Power Module (or Multi-PC Modules, where these are used).

- Data is transmitted downstream towards the Power Module on pin 2 (and from board Rev D, also on pin 6, using the second RS-232 transmitter in the MAX233), and upstream configuration data is received from the Simulator PC on pin 4.
- Power pins are doubled up to mitigate the effects of voltage drop in longer cable runs. Pins 5 and 7 are used for the nominal +9V supply to the interface, and pins 1 and 3 are 0V.

The wiring of the female PCB connector is shown in the following diagram:

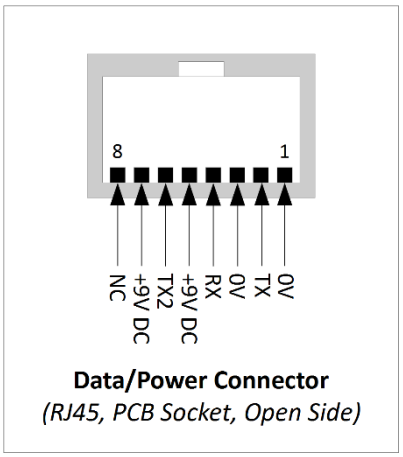


Figure 13 – Simulator Interface Module Power/Data Connector

Power Module Power/Data Connector

The Power Module uses an RJ45 connector for the upstream cabling connection to the Simulator Interface Module (or Multi-PC Modules, where these are used).

- Data is received from the upstream Simulator Interface Module on pin 2, and upstream configuration data is sent to the interface on pin 4.
- In a basic single PC configuration, any data received on pin 6 is discarded. Pin 6 is connected only to a TVS surge protection diode.
- Power pins are doubled up to mitigate the effects of voltage drop in longer cable runs. Pins 5 and 7 are used for the nominal +9V supply to the interface, and pins 1 and 3 are 0V.
- Pin 8 is not connected. This is required for correct operation when the Basic Serial Splitter Module is used.

The wiring of the female PCB connector is shown in the following diagram:

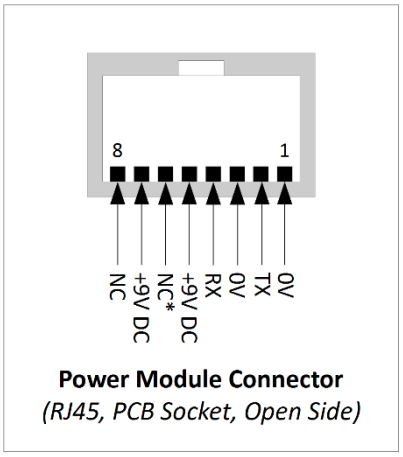


Figure 14 – Power Module Power/Data Connector

Second PC Module Connector

The Second PC Module uses RJ45 connectors for the upstream cabling connection to the Simulator Interface, and the downstream connection to the Power Module.

- Power pins 1, 3, 5 and 7, and the spare pin 8 are looped through from the *Interface* connector to the *Power Board* connector.
- Pin 4 is used to send data (marked RX – configuration data received from the point of view of the upstream interface module) and is also looped through; this pin is not connected to the 9-pin serial connector on the Second PC Module.
- Sensor data received on Pin 2 of the *Interface* connector (marked TX – sensor data transmitted from the point of view of the upstream interface module) is routed to the 9-pin serial connector on the Second PC Module.
- Sensor data received on Pin 6 of the *Interface* connector (marked TX2) is looped through to pin 2 of the *Power Board* connector.
- Connecting more than two Second PC modules is therefore not possible; only the two most upstream modules will receive data from the Simulator Interface Module.

The wiring of the female PCB connector is shown in the following diagram:

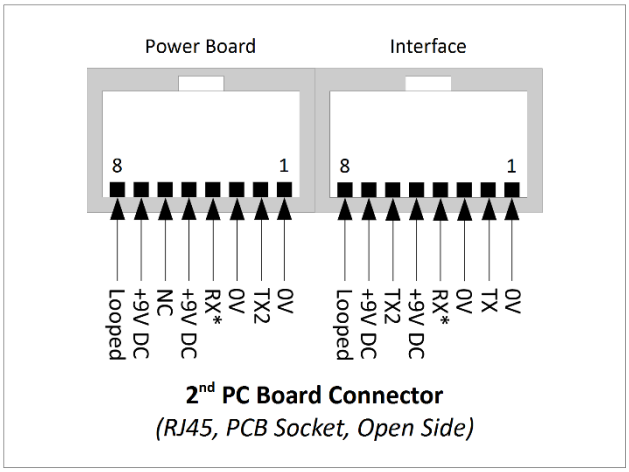


Figure 15 – Second PC Module Connector

Basic Serial Splitter Module Connector

The Basic Serial Splitter Module uses RJ45 connectors for the upstream cabling connection to the Simulator Interface Module, and the downstream connection to the Power Module.

- Power pins 1, 3, 5 and 7 are looped through from the *Interface* connector to the *Power Board* connector.
- Pin 4 is used to send data (marked RX – configuration data received from the point of view of the upstream interface module) and is also looped through; where two Basic Serial Splitter Modules are daisy-chained, the RS-232 transmitter of the upstream splitter module is disabled. This mechanism is discussed further below.
- Sensor data received on Pin 2 of the *Interface* connector (marked TX – sensor data transmitted from the point of view of the upstream interface module) is routed to the PCs connected to the Basic Serial Splitter Module.
- Sensor data received on Pin 6 of the *Interface* connector (marked TX2) is looped through to pin 2 of the *Power Board* connector. This data may be used a second, downstream Basic Serial Splitter Module.
- Pin 8 on each connector is used to enable or disable the RS-232 transmitter of the upstream splitter module where more than one Basic Serial Splitter Module is used.
- Connecting more than two Basic Serial Splitter Modules is therefore not possible; only the two most upstream modules will receive data from the Simulator Interface Module, and the transmitters of all but the most downstream module will be disabled.

The wiring of the female PCB connector is shown in the following diagram:

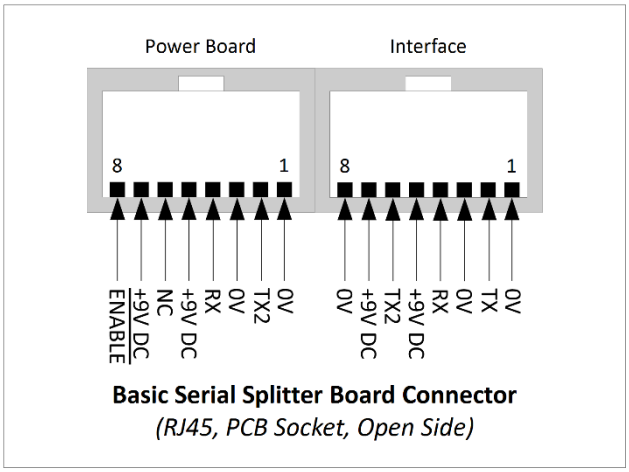


Figure 16 – Basic Serial Splitter Connector

Transmitter Disabling

Two Basic Serial Splitter Modules may be daisy-chained to support a maximum of 16 Simulator PCs.

- Where a single Basic Serial Splitter Module is used, only one Simulator PC can be used to send configuration data to the Simulator Interface Module. The active PC is selected using the 8-position DIP switch on the Basic Serial Splitter Module.
- Where two Basic Serial Splitter Modules are daisy-chained together, only one module can be used to send configuration data to the Simulator Interface Module (the active PC on that module being selected with the DIP switch, as above).
- Only the RS-232 transmitter on the downstream of the two modules is enabled; the transmitter on the upstream module is disabled.
- The disabling of the upstream transmitter is automatic and cannot be configured.

The circuit controlling the transmitter disabling function is illustrated in the following diagram:

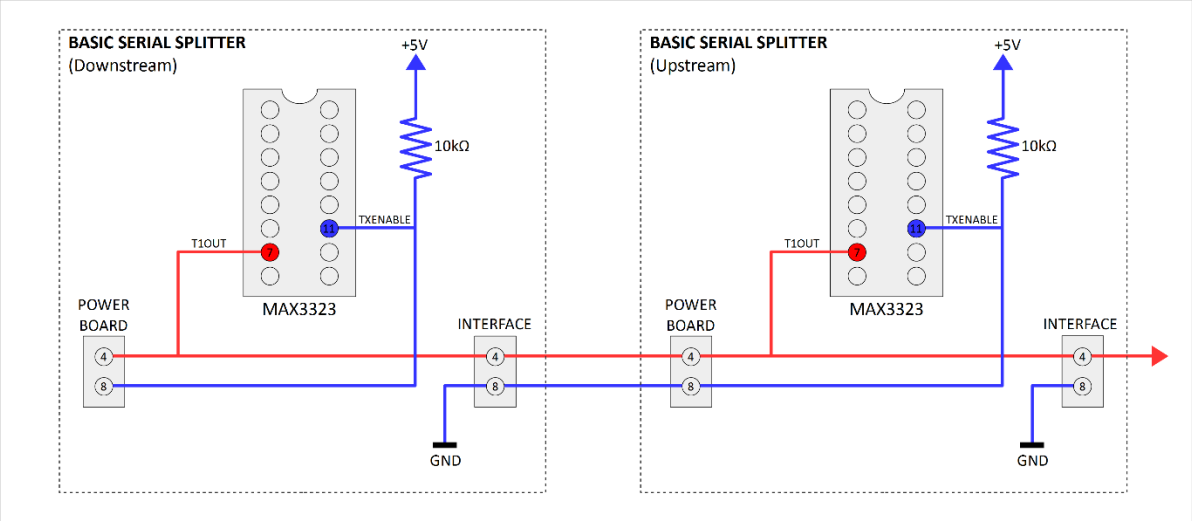


Figure 17 – Basic Serial Splitter Transmitter Control

The transmitter disabling function operates as follows:

- The Basic Serial Splitter Modules uses the MAX3323 RS-232 serial line driver IC, which has the capability to disable the serial transmitter via a control pin, making the RS-232 level transmit T1OUT[20] pin passive and high impedance[21].
- When a single Basic Serial Splitter Module is used, the TXENABLE pin is held high via a 10kΩ pull-up resistor.
- The TXENABLE pin is also connected to pin 8 of the *Power Board* connector. This is pin is not connected on the Power Module, so the TXENABLE pin is still held high. This enables the RS-232 transmitter.
- Pin 8 on the *Interface* connector is connected to ground (0V). This pin is not used on the Simulator Interface Module, so this has no effect.

[20] Pin names as defined in the MAX3323 data sheet.
[21] The RS-232 receiver can also be disabled, but this functionality is not used in the Basic Serial Splitter Module.

- When a second Basic Serial Splitter is connected, pin 8 on the *Power Board* connector of the upstream module is connected to the *Interface* connector of the downstream interface. This has the effect of pulling low the TXENABLE pin of the upstream module's MAX3323.
- This disables the upstream MAX3323's RS-232 transmitter and makes the T1OUT transmit pin high impedance. PCs connected to the upstream module cannot transmit data to the Simulator Interface Module, regardless of the DIP switch setting.
- The TXENABLE pin of the downstream module's MAX3323 is still held high by its pull-up resistor. As above, this pin is connected to pin 8 of the *Power Board* connector, but this is not connected on the Power Module, so the TXENABLE pin is still held high. The downstream module's RS-232 transmitter remains enabled.

Simulator Interface Module Firmware

Sensor Characteristics

Over the years several 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 above.

- For the purposes of the Type 2 Simulator Interface Module, 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 Type 2 Simulator Interface Module 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.
- 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 Module firmware is such that the maximum apparent 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 expiry of the “guard timer”.

Firmware Design

Interrupt Driven vs Polling

As noted in the hardware section above, 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 Modules and sending serial data to the Simulator PC as appropriate. After all bells have been processed, the code loops round and the process starts again.

In the current version of code (v3.5), with the ATmega328P driven by an external 8MHz resonator, each iteration of the main polling loop takes approximately 400µs when configured for 16 sensors, which is fast enough for the disadvantages of polling not to be significant. Enabling debug mode and the CRO timing pin increases the loop time by a few percent, but as these are not intended for active simulator use this is also not significant.

The timing of the main loop using the CRO timing pin is illustrated in the following CRO capture:

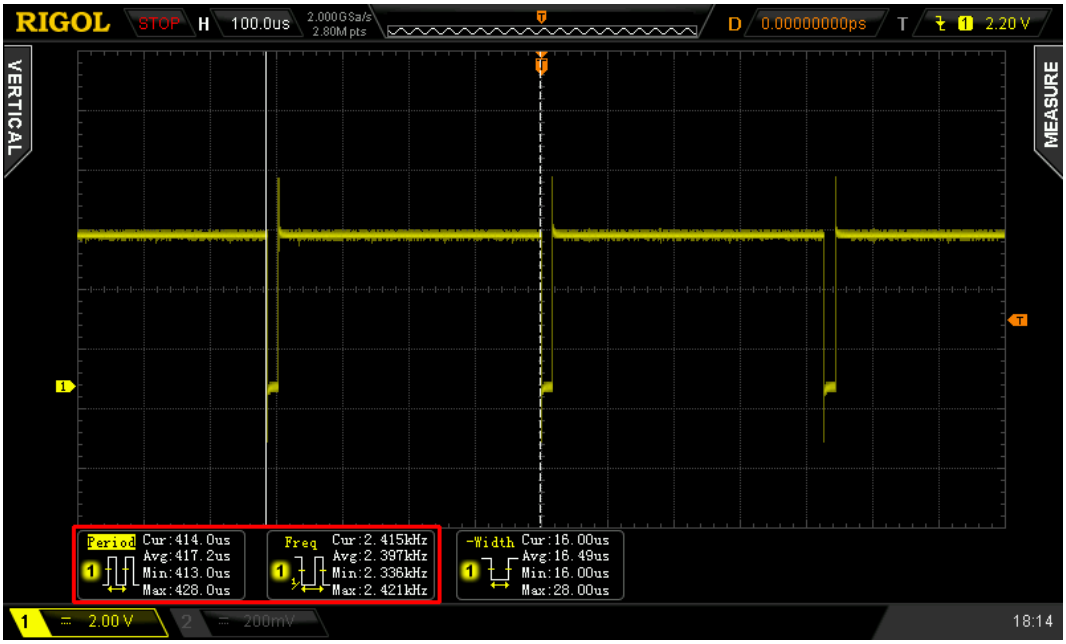


Figure 18 – Simulator Interface Main Loop Timing

Processing of serial input from the Simulator, in the form of CLI commands, is triggered by the standard Arduino *serialEvent()* function.

State Machine Operation

The Simulator Interface firmware maintains a set of 16 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 for Guard Timer, Sensor Disabled, or Test Mode.*

These changes of state are illustrated in the following diagram:

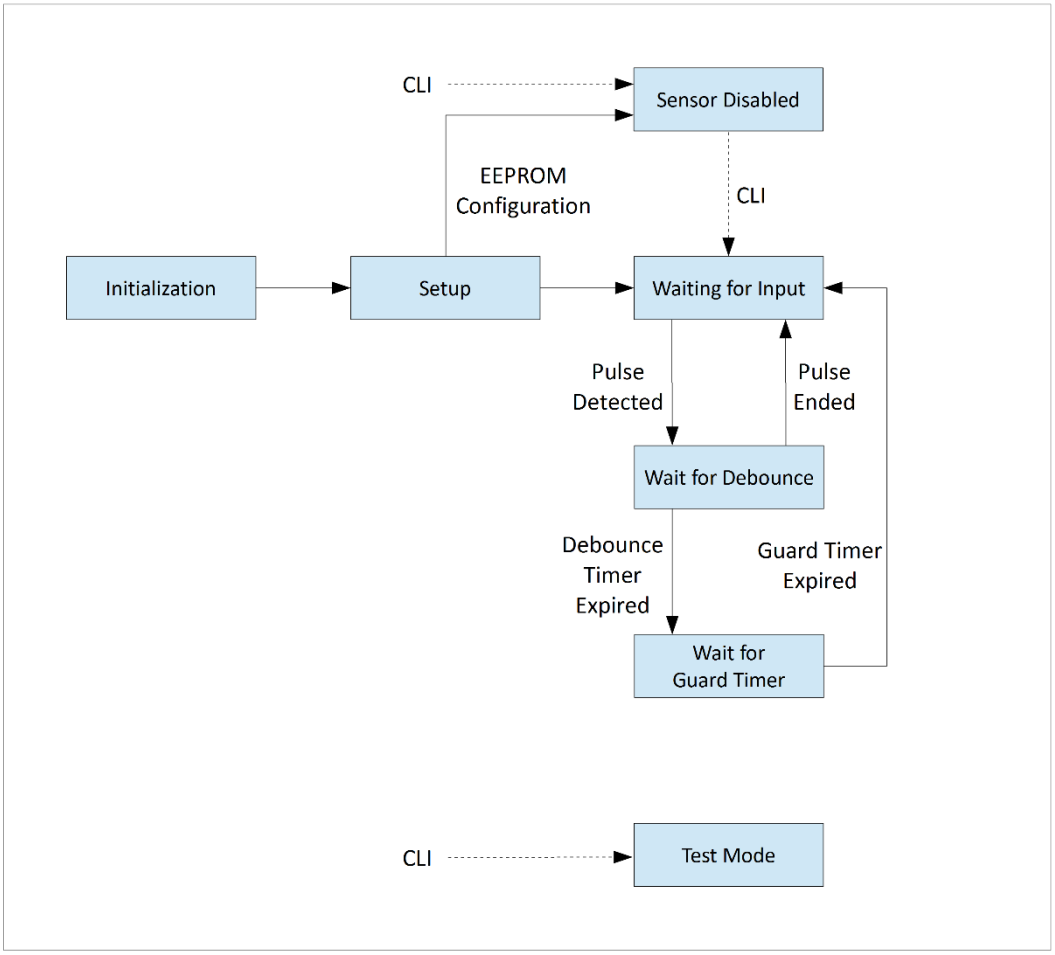


Figure 19 – State Machine Transitions Illustration

Waiting for Input

When a channel'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 channel, 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 connected Sensor Module.

- 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 pulse is detected then the code 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, 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 2ms, but this can be configured through the Simulator Interface CLI.

Waiting for Debounce

When a channel'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 channel.

- 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 sends the appropriate character over the serial interface to the Simulator PC, and switches the state machine into the *Waiting for Guard Timer* state.

Waiting for Guard Timer

When a channel's state machine is in the *Waiting for Guard Timer* state, the firmware code compares the value of the Guard Timer 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 Guard Timer* state.
- If the timeout has been reached or has been passed, the code switches the state machine back into the *Waiting for Input* state, and the cycle starts again. The default value for the Guard Timer is 10cs (100ms), but this can be configured through the Simulator Interface CLI.
- The channel input pin is not read in the *Waiting for Guard Timer* state (other than for debugging purposes), effectively debouncing the end of the sensor pulse.

Disabled Sensor

When a channel'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.

Note that the number of channels to be scanned on each iteration is determined dynamically with reference to the highest numbered enabled channel: For example, if the highest numbered enabled channel is channel 10, only channels 1 to 10 will be scanned, shortening the loop time. If the highest numbered enabled channel is channel 16, then channels 1 to 16 will be scanned.

Test Mode

The *Test Mode* state implements a very simple test function, ignoring sensor inputs and generating simple test patterns 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.

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 both the *millis()* and *micros()* functions for its main timing and delay calculations (*micros()* is used only by debugging code). These functions return 32-bit unsigned long data types; consequently the value returned by *micros()* will increase and then overflow back to zero after approximately 50 power-on days, and the value returned by *micros()* will overflow approximately every 70 minutes.

These overflows would disrupt simulator processing, as state machines may become "stuck" waiting for an overflowed timer to expire.

- In the case of the *micros()* timer, which is used for the sensor debounce timer, the Simulator Interface firmware attempts to catch this overflow, and looks for timer expiry times which are an unreasonably long time into the future.
- For the *millis()* timer, the current version of Simulator Interface firmware will require a power cycle or reset to clear the problem. This issue may be addressed in a future version of the firmware. 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.

Sensor De-Bouncing

The Simulator Interface firmware reacts to the start of a pulse from a sensor. In practice the output from some sensor types may be noisy and 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:

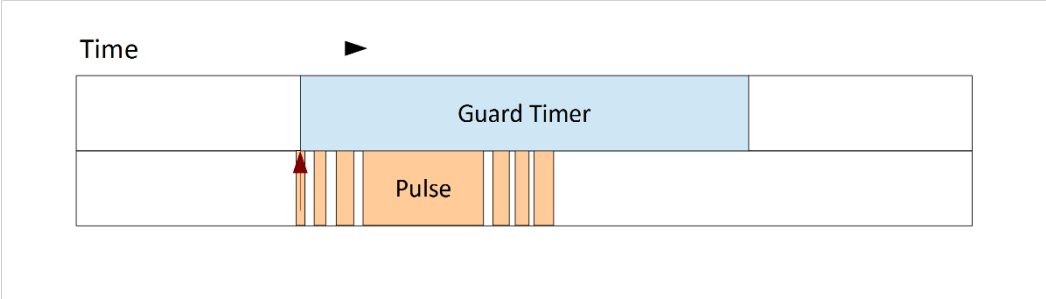


Figure 20 – Sensor De-Bounce Illustration

The current version of the Simulator Interface code reacts to the first observed transition in the output of the sensor (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 debounce timer (default 2ms, 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. The debounce timer is specified in milliseconds, but calculated in microseconds for improved consistency. The resolution of the microsecond timer is 8µs.

When the state machine has switched into the *Waiting for Guard Timer* 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 guard timer, this effectively de-bounces the input. Typically, the input pulse from an optical sensor will be a few milliseconds in duration, and the guard 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 Modules 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 Module 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 PC, as defined in the MBI Protocol specification.

The following debugging features are available in firmware v3.5:

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 16 channels.
- The Debug Mask setting may be saved in non-volatile EEPROM[22].
- 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 space separated data fields:

- The channel number (1-16),
- The mapped ASCII character (1-90ETABCDWXYZ),
- The guard timer expiry offset time 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 guard timer.

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

Note that the pulse length timing code is not entirely robust: the assumption is made that the first Sensor Module pulse will be shorter than the guard timer. If the first pulse has not completed by the time the guard 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.

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 space separated data fields:

- The channel number (1-16),
- The mapped ASCII character (1-90ETABCDWXYZ),
- The debounce timer expiry offset time in microseconds,
- 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 debounce timer will generate output on a separate line, with the following data fields:

- The channel number (1-16),
- The mapped ASCII character (1-90ETABCDWXYZ),
- The debounce timer expiry offset time in microseconds,
- The identifier “D”.

Debug Show LED

When the *Debug Show LED* flag is set, the diagnostic LED output will be enabled for all bells for which debugging is enabled. By default, the 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 must handle legitimate MBI Protocol commands and delay data sent by the Simulator Software, in order to differentiate these from CLI input (even though these data are not used by the Type 2 Simulator Interface). 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 adopts the following strategy when handling serial input.

- The first byte received is examined. If this byte is a recognised MBI Protocol command (See Appendix B), it is discarded. No responses are sent to MBI commands.
- 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 13 bytes (12 values plus the specified terminator byte) within a defined timeout interval (one second in the current v3.5 firmware).
- If 12 values plus the correct terminator byte have been received within the timeout period, these are assumed to be delay values, and all 13 bytes are discarded.
- 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 diagnostic LED is used to indicate that a CLI command has been received, or that an error occurred. The CLI command are documented in Appendix D below.

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.

As noted above, the number of channels to be scanned on each iteration is determined dynamically with reference to the highest numbered enabled channel.

Memory Footprint

The 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 v3.5 on an ATmega328P is as follows:

- 12,742 / 32,768 Bytes Flash Memory
- 718 / 2,048 Bytes SRAM
- 23 / 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 v3.5 of the Interface firmware can run on older ATmega128 microcontrollers with 16kBytes of Flash Memory and only 1kByte of SRAM.

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 Modules 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 Module during normal ringing.
- The Sensor Pulse Duration: The length of the pulse from each Sensor Module 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 Interface Module 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 approximately 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 PC, 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 PC[23].

(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.)

[23] http://blog.arduino.cc/2011/10/04/arduino-1-0/

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 400µs, giving a resolution of less than 0.017% 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 typical, the theoretical expected duration of the signal pulse from the Sensor Module is a function of the width of the trigger (e.g., 25mm wide reflective tape or 20mm diameter magnets 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[24]. 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 may activate before the trigger is fully axially aligned with the module (and will not deactivate until after the trigger has ceased to be fully axially aligned). In addition to this, the detection zone on many sensors is effectively conical, and at the wheel is therefore somewhat larger than the physical diameter of the detector.

[24] http://www.cl.cam.ac.uk/~fhk1/Bells/equations.pdf

The following diagram illustrates this effect:

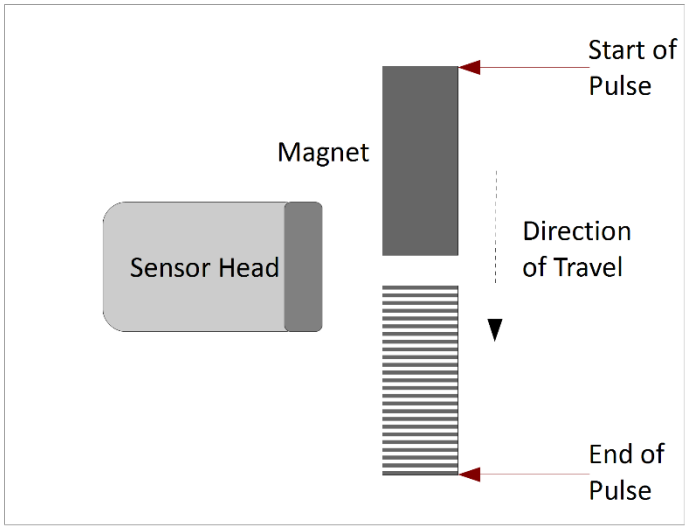


Figure 21 – Sensor Pulse Duration Illustration

In the diagram above, the Sensor Module signal pulse starts before the trigger is fully aligned with the sensor (top red arrow), and ends after the trigger has cease to be fully axially aligned (bottom red arrow). Thus, the effective width of the trigger, and the duration of the resulting pulse, are increased.

This is supported by evidence: An analysis of the theoretical and observed sensor pulse durations from the optical Sensor Heads of the original Liverpool Cathedral Simulator installation can be found in Appendix C. The sensor shielding tube has 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 in the Simulator Interface results in a number of potential problems:

Missed Sensor Signals

If the Simulator Interface firmware polling interval is longer the pulse emanating from a Sensor Module, 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:

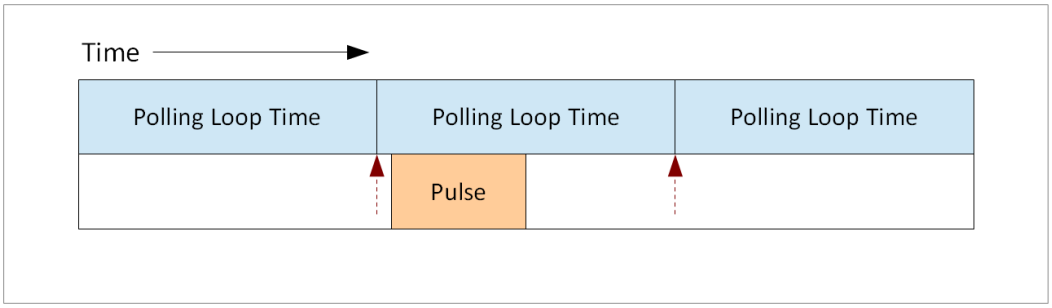


Figure 22 – Missed Sensor Signals Illustration

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 pulse begins and completes without ever being detected.

Given a typical pulse duration of approximately 6ms, and a maximum polling loop duration of approximately 400µs, each pulse therefore lasts approximately 15 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 Module (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 would be sent to the Simulator PC.

The following diagram illustrates this problem:

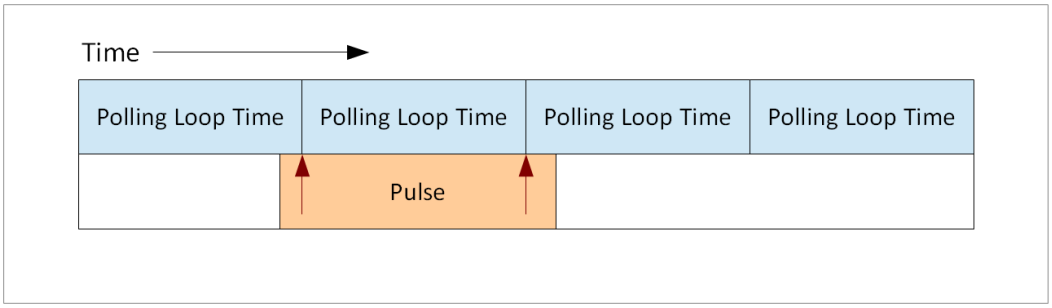


Figure 23 – Duplicated Sensor Signals Illustration

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

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, the pin is still LOW, but 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 Module 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 400µs 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 400µs (the duration of the polling loop).
- This would 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:

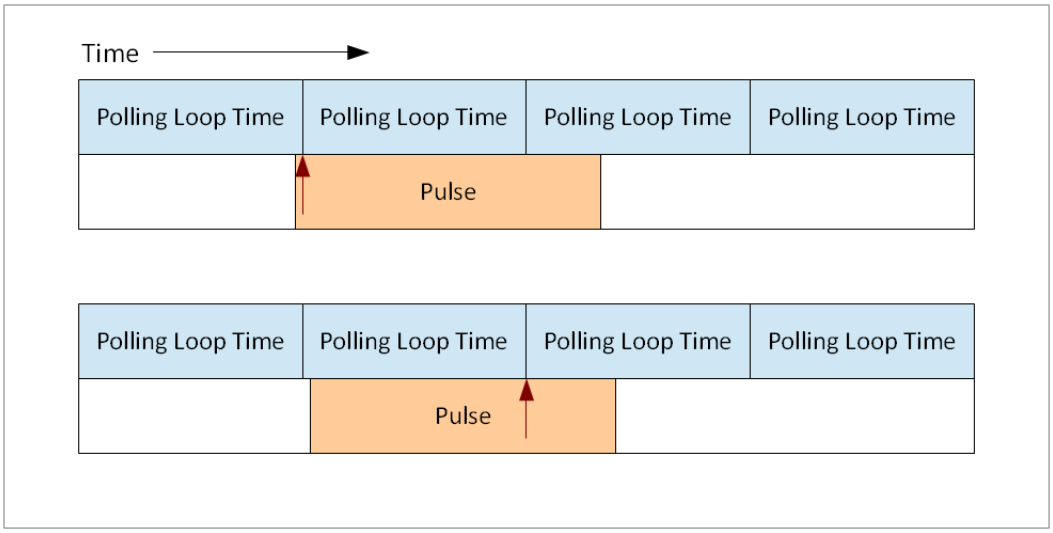


Figure 24 – Variable Odd-Struckness Illustration

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

Given a typical inter-row interval of say 2.3s, and a maximum polling loop duration of approximately 400µs, the variable odd-struckness contributed due to this problem is therefore between zero and ±0.009% 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 contribute approximately several tens of microseconds to the loop time, and could be omitted entirely from production code.

Latency

The latency of a simulator system is the delay between the bell triggering the sensor at BDC, and the arrival of the corresponding signal at the Simulator PC.

The latency of the simulator system is not critical in BDC detecting systems, provided it is relatively small (much less than the time between BDC and strike), is reasonably consistent, and is symmetrical about the BDC position. Latencies of up to several tens of milliseconds may be accommodated by tuning the delay time applied by the Simulator PC.

The latency of the simulator consists of the sum of the following:

- The debounce timer,
- The polling loop time,
- The RS-232 serial speed.

The default debounce timer is set to 2ms. As discussed above, the polling approach results in some variation of detection time, and the polling loop time is approximately 400µs. Moving the state machine from *Waiting for Input* to *Waiting for Debounce*, and then detecting the end of the debounce timer, involves two polling events. The average latency from the start of a sensor pulse to the start of the serial transmission should therefore be approximately[25] 2.4 ±0.2ms.

The serial line speed is 2400bps. Each ASCII character transmission consists of 10 bits (8 data bits, preceded by one start bit and followed by one stop bit). The time taken to transmit one character is therefore approximately 4.2ms and is constant.

The theoretical total latency of the simulator is therefore approximately 6.6 ±0.2ms. Any additional latency arising in the Simulator Software Package is beyond the scope of this discussion, but may be compensated for by tuning the delay time applied by the software.

The typical latency of the Simulator Interface is illustrated in the following annotated CRO capture:

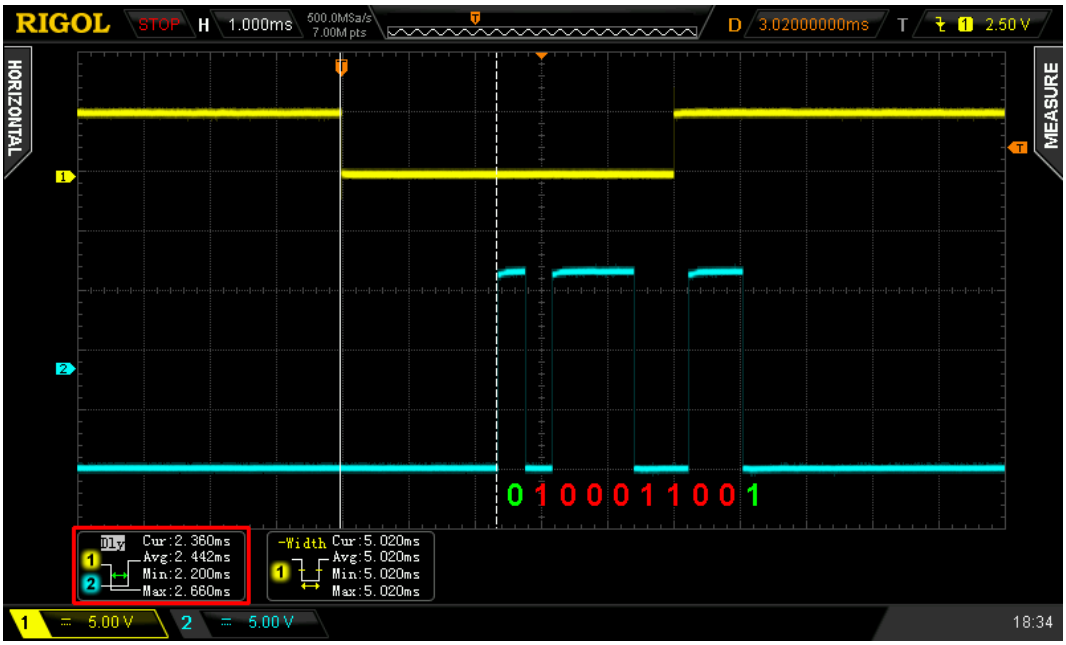


Figure 25 – Simulator Interface Latency

- The upper yellow trace represents a simulated sensor pulse measured on a Simulator Interface Module input pin. The duration of the pulse is approximately 5ms, and the pin transitions from +5V to 0V at the start of the pulse.
- The lower blue trace represents the RS-232 serial transmission, measured on the serial transmit pin of the same interface.
- On the serial line, logic 0 is a high voltage (approximately +8V), logic 1 is low (approximately -8V), and the bus idles at the logic 1 level[26]. The start bit is logic 0, the stop bit is logic 1, and hence the end of the stop bit cannot be distinguished from the idle level.

[25] This estimate does not consider the resolution of the microsecond timer, 8µs.
[26] Note that these are for the RS-232 serial interface. The TTL serial interfaces between the microcontroller and the line driver operates between 0V and +5V, with inverted logic.

- The character transmitted is ASCII “1” (0x31), binary 00110001, and is transmitted LSB first. Reading from left to right, the total bit stream is therefore “0100011001” (10 bits: a logic 0 start bit, ASCII “1” LSB first, and a logic 1 stop bit).
- The delay between the start of the sensor pulse and the start of the serial transmission is measured over multiple pulses as being between 2.200ms and 2.660ms, average 2.442ms. The latency start to end of the serial transmission can be estimated as approximately 4.2ms. These values are very close to the theoretical predictions.

Software Development & Compatibility

Development Environment

The Simulator Interface firmware has been developed on the following versions of operating systems, development tools and Simulator Software packages. Later versions of tools are expected to be compatible:

- Microsoft Windows 7 SP1 32-bit
- Arduino IDE 1.6.12
- PuTTY 0.70
- PortMon 3.03

Source Code 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.

Simulator Software Compatibility

Tested Configurations

The Type 2 Liverpool Ringing Simulator Interface has been tested successfully with Simulator Software Packages up to the following versions:

- Abel 3.10.1a
- Beltower 2017 (12.35)
- Virtual Belfry[27] 3.5

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.

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.

[27] 3.1b is the minimum version of Virtual Belfry required for proper handling of interfaces of this type.

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[28] and FTDI[29] 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

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

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.

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

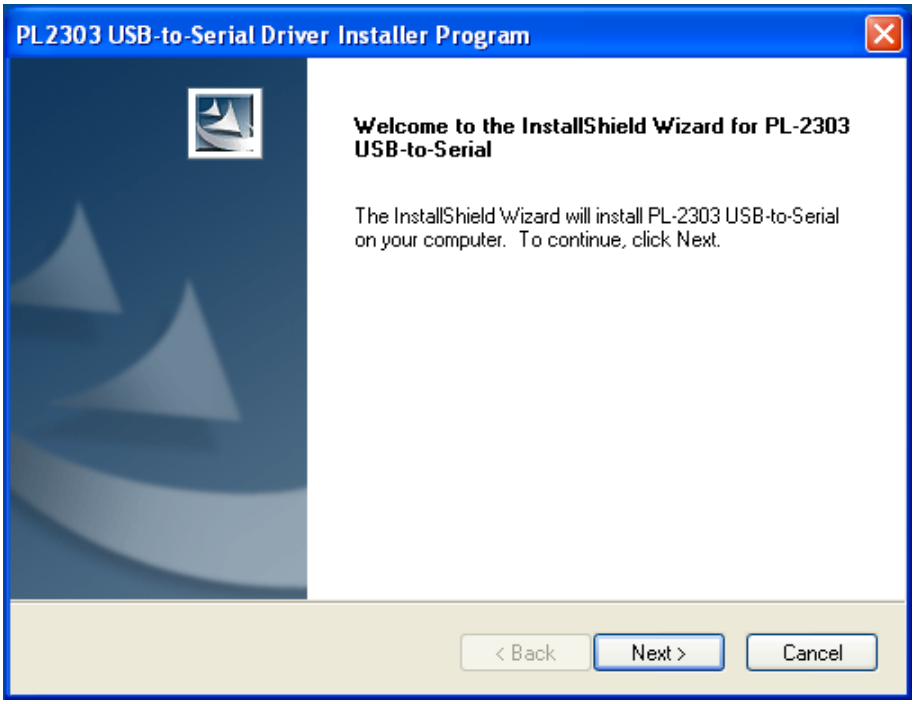


Figure 27 – Prolific Driver Installation

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

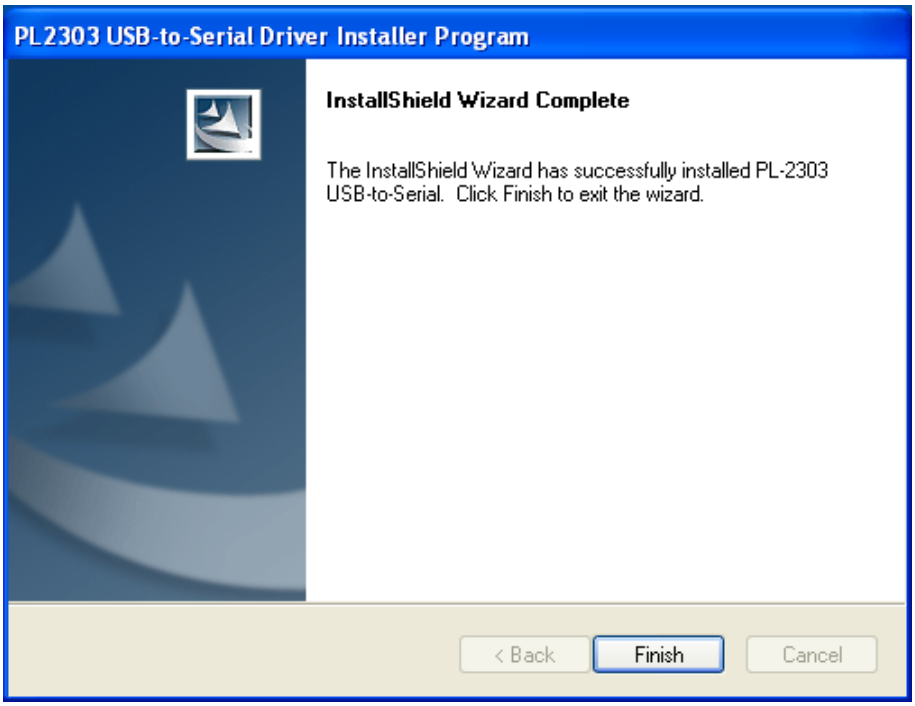


Figure 28 – Driver Installation Complete

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.

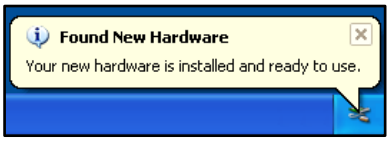


Figure 29 – Found New Hardware Message

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.

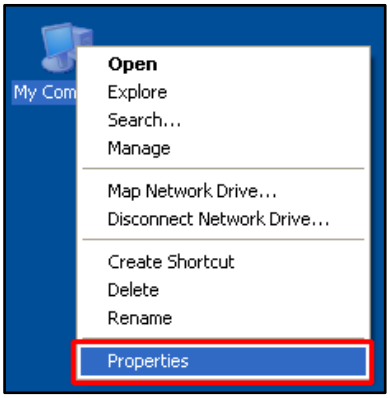


Figure 30 – My Computer Context Menu (Windows XP)

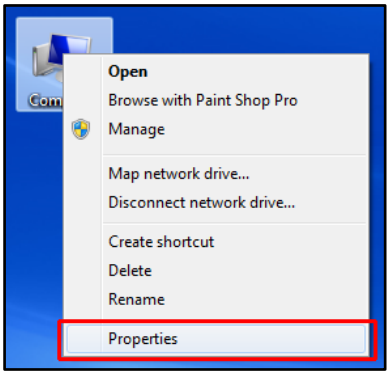


Figure 31 – Computer Context Menu (Windows 7)

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

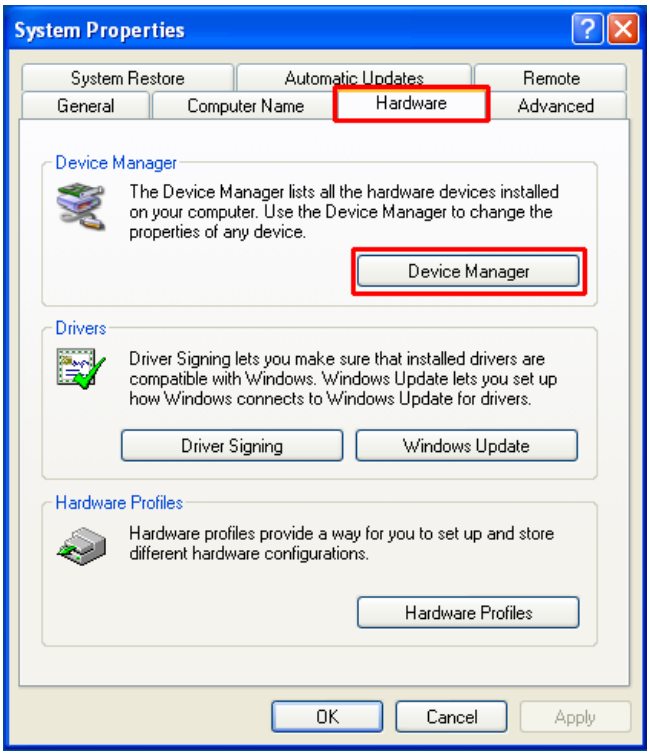


Figure 32 – System Properties Hardware Tab (Windows XP)

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

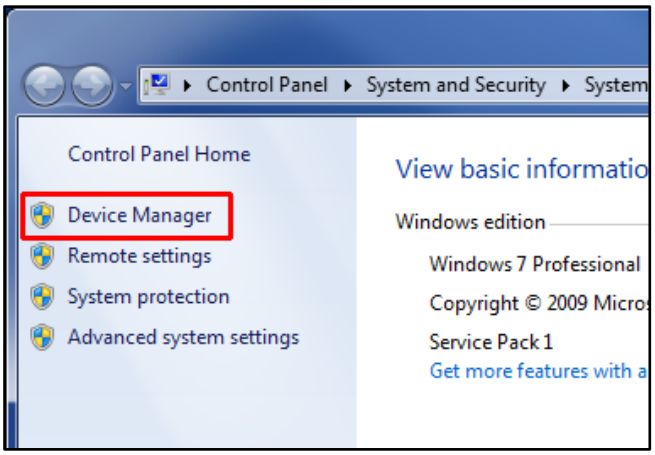


Figure 33 – System Properties Hardware Tab (Windows 7)

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*.

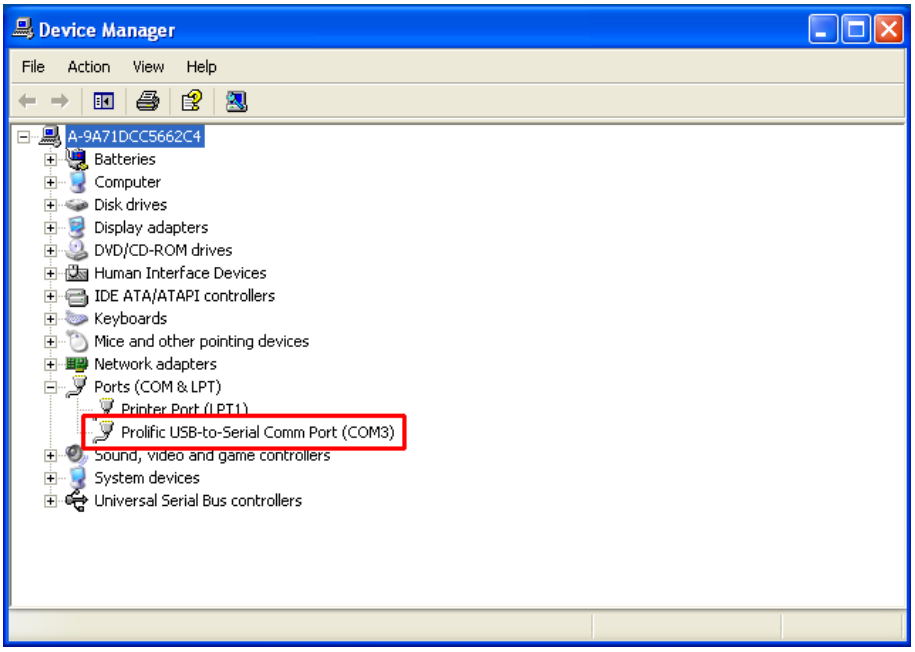


Figure 34 – Device Manager (Driver Installed)

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.

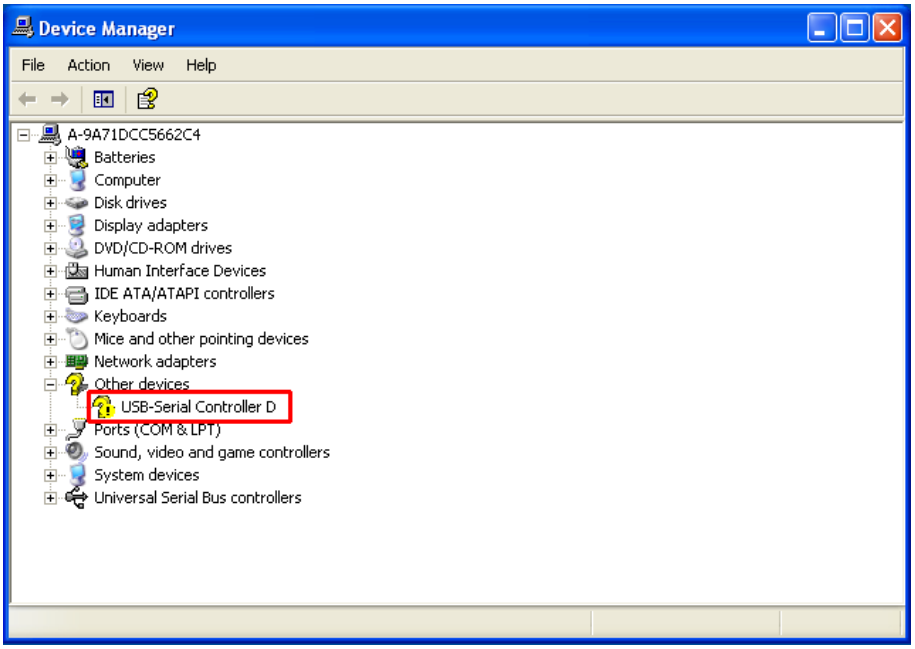


Figure 35 – Device Manager (Driver Missing)

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[30]. If the adapter has been assigned a COM port number outside the supported range, the port must be reconfigured to lower value[31].

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.

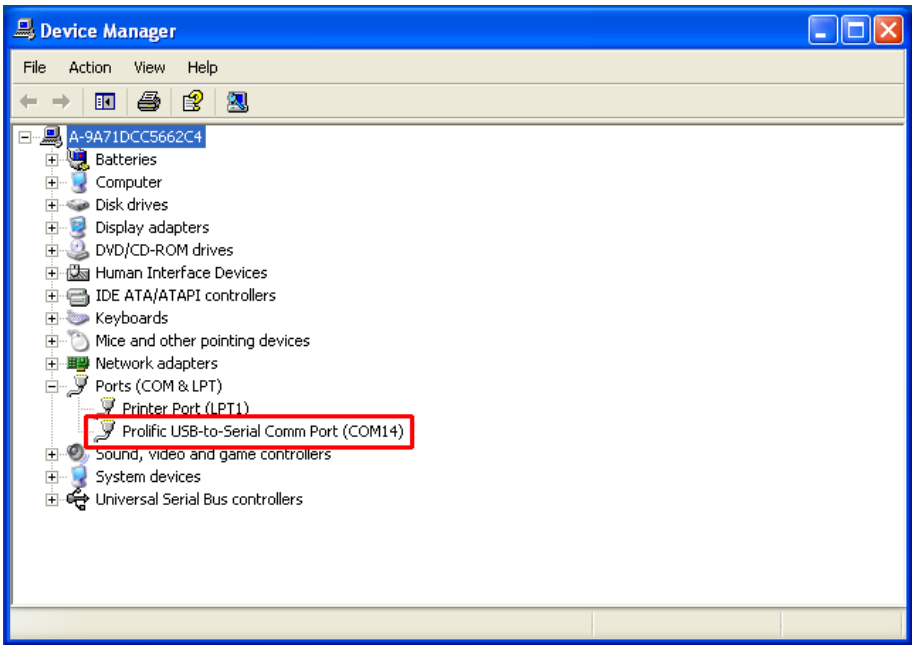


Figure 36 – Device Manager (Port COM14)

[30] Abel: COM1 – 8. Beltower: COM1 – 8 (32 from Beltower 2016).
[31] Virtual Belfry does not have this requirement.

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

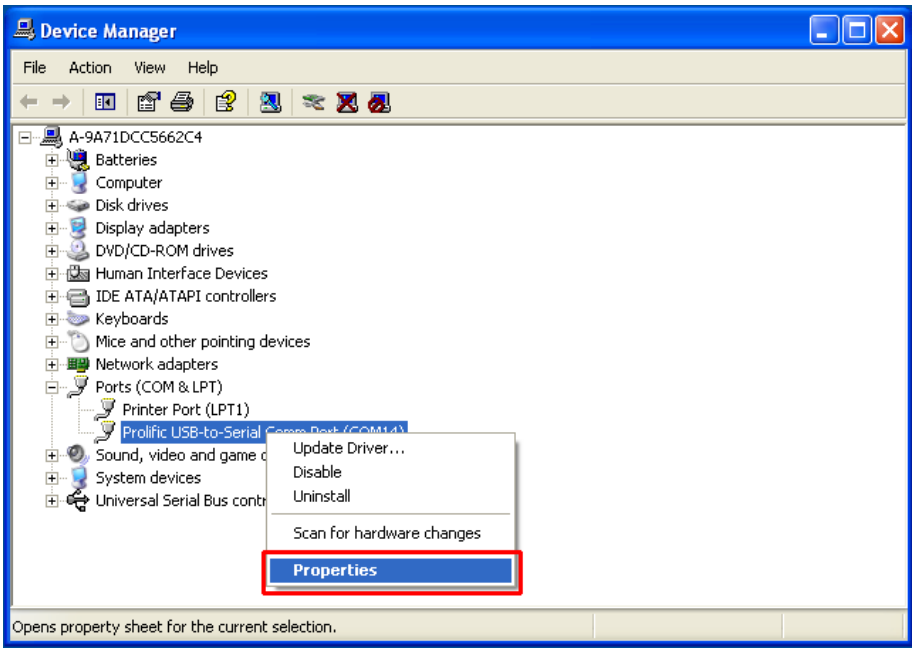


Figure 37 – Device Manager Context Menu

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

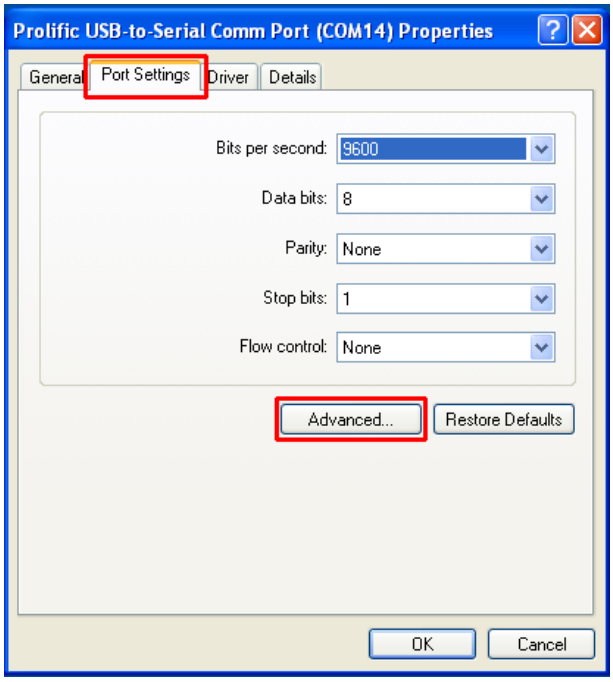


Figure 38 – COM Port Properties

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 in use.

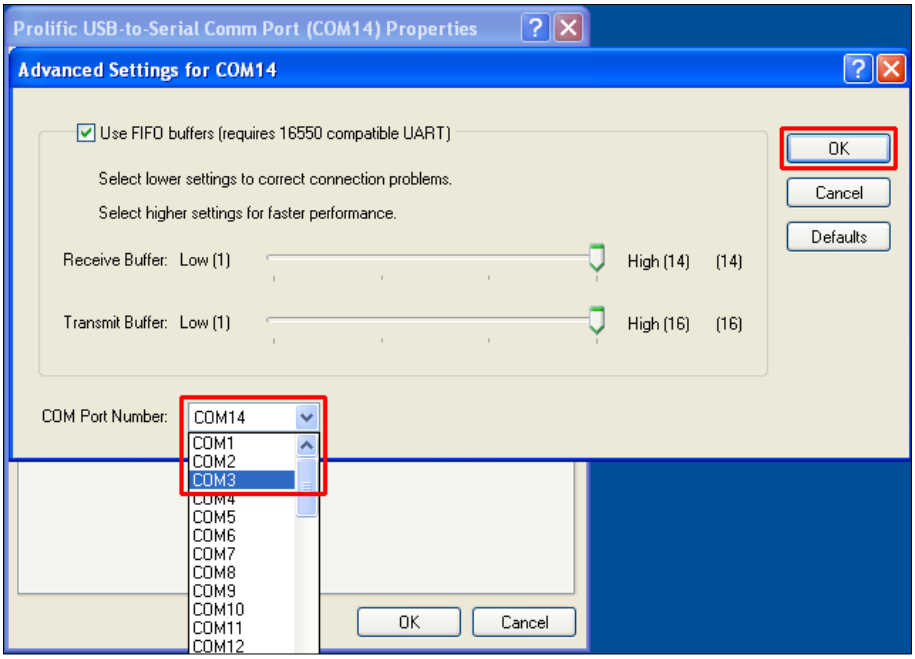


Figure 39 – COM Port Advanced Settings (Prolific)

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

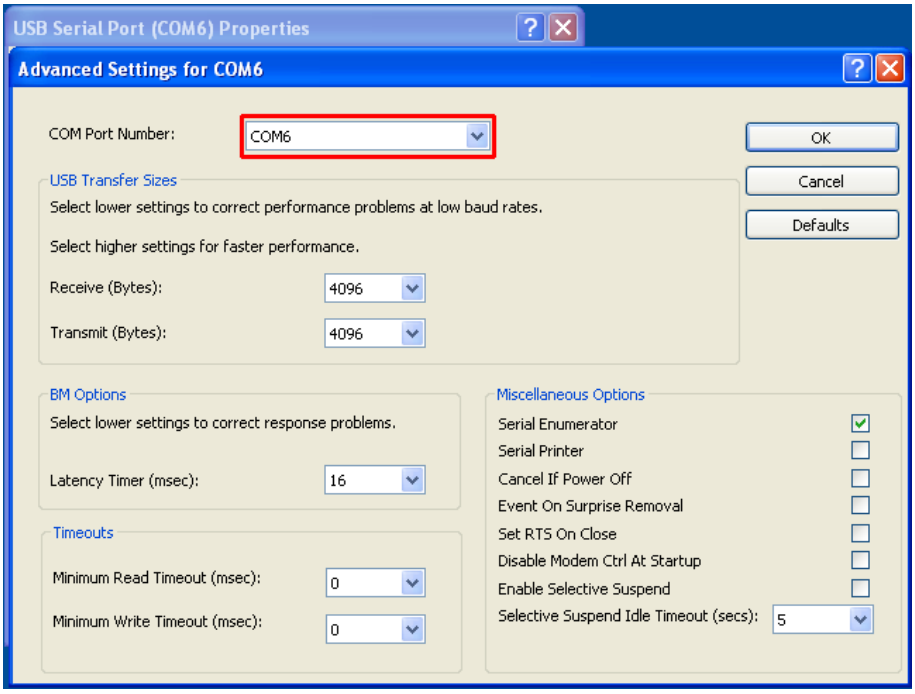


Figure 40 – COM Port Advanced Settings (FTDI)

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.

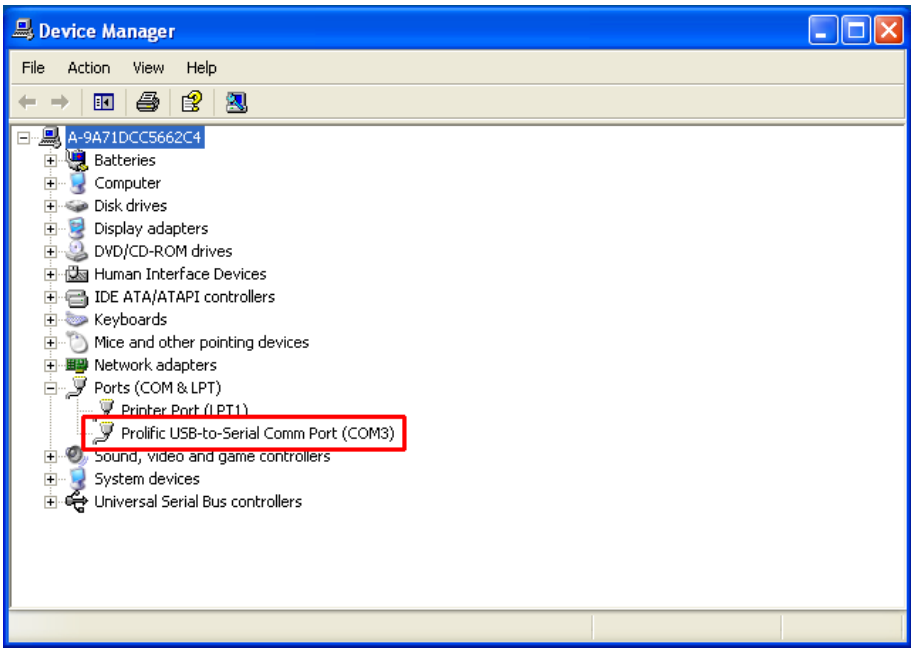


Figure 41 – Device Manager (Reconfigured Port COM3)

Appendices

Appendix A: MBI Protocol Description

Note that although the Type 2 Liverpool Ringing Simulator uses the basic MBI serial protocol described below, it does not upload or store the delay timers in the Simulator Interface Module, and those parts of the protocol are not implemented. Delays are instead applied by the Simulator Software Package on the Simulator PC. This allows the simulator to support more than 12 sensors and allows more flexibility, for example in the simulation of raising and lowering.

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 sensors, 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 PC 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 4 – 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. • This command is not used or supported by the Type 2 Liverpool Ringing Simulator firmware.
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 when operating in MBI mode. • This command does not appear to be used by the tested versions of Abel or Virtual Belfry. • This command is not used or supported by the Type 2 Liverpool Ringing Simulator firmware.
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 when operating in MBI mode. • This command is not used or supported by the Type 2 Liverpool Ringing Simulator firmware.

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 5 – Inter-Blow & Inter-Row Intervals

Bells[32]	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
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

[32] Including the cover bell, if any.

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 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 6 – 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 7 – 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

BDC-to-Strike Intervals

The following chart shows the average time differences obtained during development between the triggering of an infra-red optical Sensor Module and a 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.

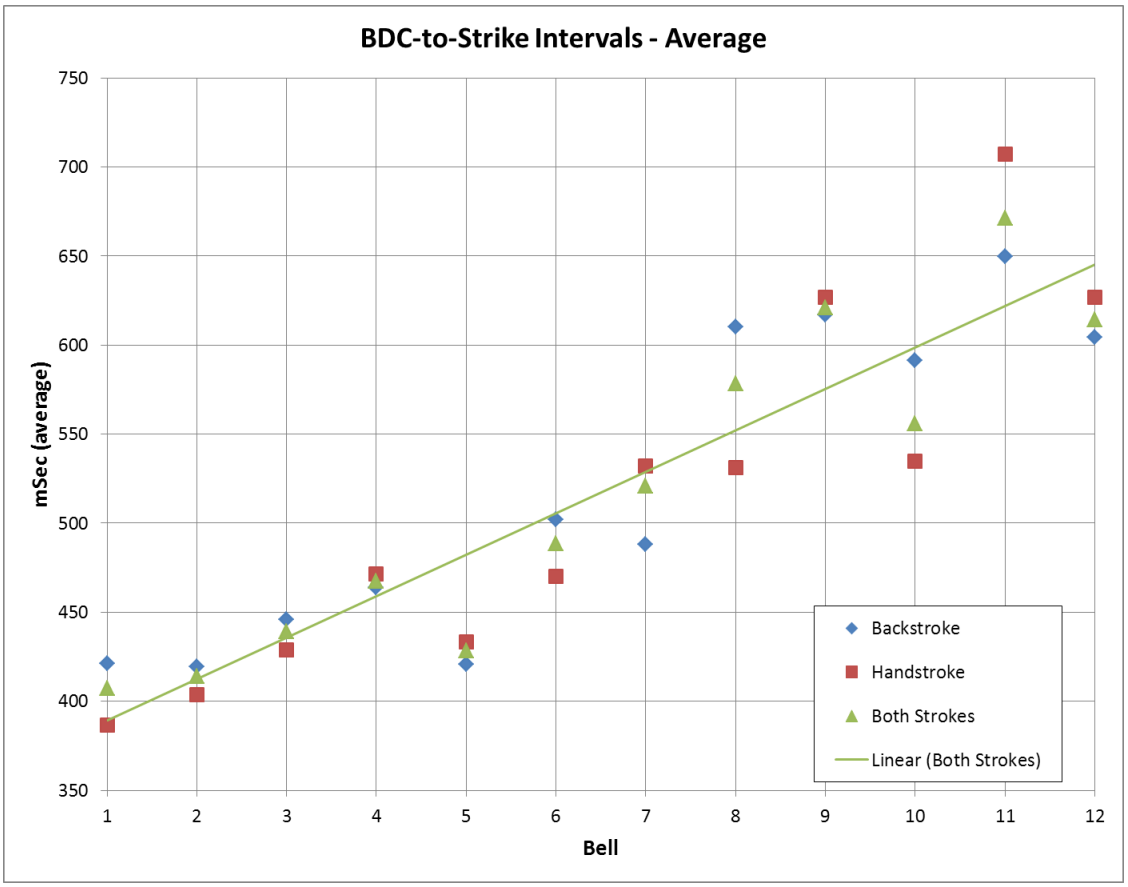


Figure 42 – Liverpool Cathedral Odd-Struckness Chart

Appendix D: CLI Command Reference

The following table lists the additional CLI commands supported by the Type 2 Simulator Interface firmware, and their functions, as of version 3.5.

Table 8 – CLI Command Reference

Command	Function
?	Display current settings This command sends a detailed set of configuration and operational values to the serial port for review.
B/b	Set the de-bounce timer (1ms -> 20ms, default 2ms). This command prompts for the value of the de-bounce time, in milliseconds. See also command “S”.
G/g	Set the guard timer (1cs -> 50ms, default 10cs). This command prompts for the value of the guard time, in centiseconds (1cs = 10ms). See also command “S”.
E/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 Module. See also command “S”.
R/r	Remaps Sensor Module channel numbers (1 to 16) to Simulator output signals, following the usual conventions for bell numbers (0-90ETABCD). Can also be used to map channels to the MBI external switch codes (WXYZ), supported by Abel only.
S/s	Save settings to EEPROM. This command saves the value set by the “B”, “G”, “E” and “R” commands to non-volatile memory. If Debug Mode is enabled, the Debug Mask (“M”) is also saved.
P/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 immediately 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.
T/t	Enable test mode. Generates several useful test patterns on the serial interface. A hardware reset is required to exit test mode.
H/h	Display CLI help text.
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. 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 Simulator Software Package.
d	Turn debug mode OFF
The following commands are available in Debug Mode only.	

M/m	Set or unset the debug mask bits. (Debug Mode only) 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 Module. See also command "S".
Z/z	Set defaults. (Debug Mode only) This command is available only when Debug Mode is turned on, and sets default values for all the configurable settings. See also command "S".
0 … 9	Print debug markers (Debug Mode 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.
L/l	Display CLI debug mode help text (Debug Mode only)

Appendix E: Diagnostic LED Codes

The following tables list the meanings of the codes shown by the flashes of the diagnostic LED on the Simulator Interface Module. This list is correct as of firmware v3.5.

- The LED flashes slowly when test mode is running.

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

Table 9 – LED Signal Codes

Long	Short	Meaning	CLI Command
N	*M*	Announces the firmware version on power-on.	-
1	0	The LED gives one long flash every time the Sensor Module connected to Channel 1 is triggered. The LED lights when the signal from the Sensor Module is received, and extinguishes when the corresponding guard timer expires (the length of this flash is therefore equal to the setting of the guard timer). The *Debug Show LED* debug flag enables this behaviour for all channels for which debugging has been selected.	-
On Steady		On after power-on: Port speed is not 2400bps, or a sensor channel is disabled (Debug mode only).	-
0	1	CLI key press acknowledgement.	-
1	1	MBI Protocol command 0xFD received.	-
1	2	MBI Protocol command 0xFE received.	-
1	3	MBI Protocol delay timers (and terminator byte) received.	-
2	3	De-bounce timer set.	B
2	4	Guard timer set.	G
2	5	Settings saved to EEPROM.	S
2	6	Enabled channels set.	E
2	7	Serial port speed set in EEPROM.	P
2	8	Debug mode on.	D
2	9	Debug mode off.	d
2	10	Debug mask set.	M
2	11	Channels remapped.	R
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		
JLCPCB	PCBs	https://jlcpcb.com/
Seeed	PCBs	https://www.seeedstudio.com/
OSH Park	PCBs	https://oshpark.com/
David Bagley	MBI, Sensors	http://www.ringing.demon.co.uk
Belfree	Wireless Ringing System	https://belfree.co.uk/
Microchip	Microcontrollers (formerly Atmel)	https://www.microchip.com
Maxim	Serial Line Drivers	https://www.maximintegrated.com/
Information		
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	https://www.cl.cam.ac.uk/~fhk1/Bells/equations.pdf
Tools		
Arduino	Electronic Prototyping	https://arduino.cc/
PuTTY	Free Terminal Emulator	https://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	https://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	https://www.audacityteam.org

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 and detection of the Sensor Module was reliable, with no missed signals (which would have been indicated as very high values) or spurious triggers (very low values).

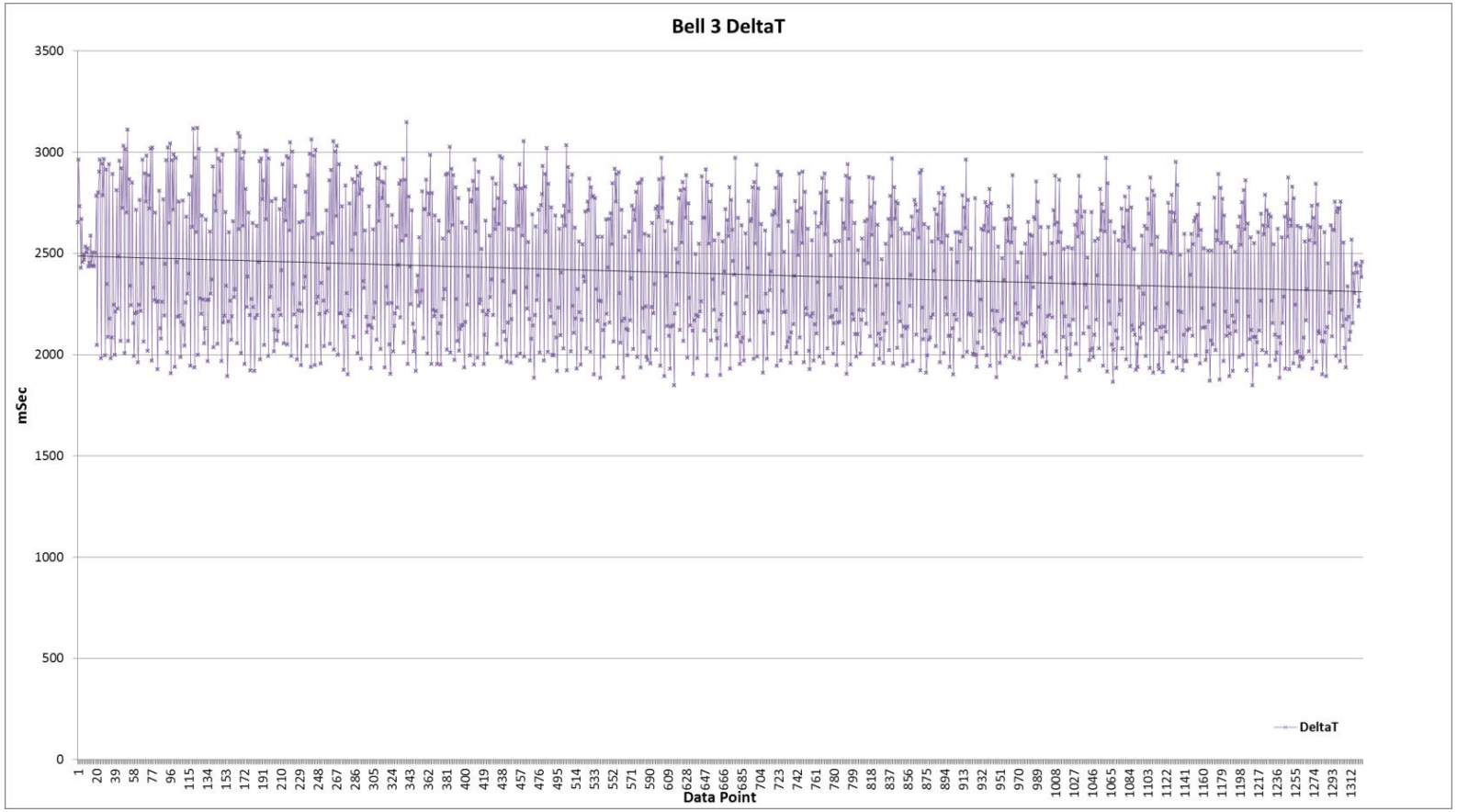


Figure 43 – Quarter Peal Sensor Head Test Timings

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[33] are released under the Creative Commons Attribution-ShareAlike 4.0 International License (CC BY-SA),[34] 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.*

Software

All original software released as part of the Liverpool Ringing Simulator project is released under the GNU General Public Licence (GPL), Version 3[35], 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.*

[33] http://www.simulators.org.uk
[34] http://creativecommons.org/licenses/by-sa/4.0/
[35] http://www.gnu.org/licenses/gpl-3.0.en.html

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 and ringers of the following towers for their willingness to be the crash test dummies of simulator design and testing.

- Liverpool Cathedral
- St George's, Isle of Man
- St Mary, Chirk, Wrexham
- St John, Higham, Kent
- St Margaret, Crick, Northamptonshire
- St Mary & St Peter, Lois Weedon, Northamptonshire