FACULTY OF ENGINEERING AND SURVEYING

DEVELOPMENT OF A RASPBERRY PI BASED, SDI-12 SENSOR ENVIRONMENTAL DATA LOGGER

A dissertation submitted by

Mr James Coppock

Dissertation submitted to the Faculty of Engineering and Surveying in partial fulfilment of the requirements for the degree of

Bachelor of Engineering

(Electrical and Electronics)

October, 2015

Abstract

SDI-12 is a powerful tool for sensor networking and environmental data acquisition (EDA). Sensory networks are employed by many commercial and non-commercial entities across a wide range of applications to achieve better outcomes for the environment, the investing parties or the wider community. Monitoring systems can reduce operation costs and improve quality of products or produce. Many applications for sensor networks are of ethical significance for example, applications related to sustainable living, education, scientific research and food production. Despite the potential benefits, whether people adopt a system is largely dependent on associated costs and complexity. Consequently an inexpensive, reliable and easy to use system is more likely to be adopted. The Raspberry Pi is a powerful and inexpensive computing platform for embedded projects which incorporates a 40 pin general purpose input output (GPIO) header for connecting to digital peripherals, which is used as the basis of this project.

The prototype SDI-12 logger software is written in C++ and uses an existing Arduino SDI12 C++ library that has been modified for use with the Raspberry Pi computer. The system is evaluated for its suitability as a simple easy to configure (plug-and-play) type logger.

The SDI-12 software developed, while functional, has only a subset of the features that a market ready device will need. Future work includes adding control outputs for automation, a graphical user interface and also leveraging the Raspberry Pi's network capabilities to allow remote access for setting and disabling alarms and also for uploading of data to an online database for remote access.

University of Southern Queensland

Faculty of Health, Engineering and Sciences

ENG4111/ENG4112 Research Project

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

James Coppock

Student Number: 0050067987

Coppock

Signature

29/10/15

Date

Acknowledgements

I would like to thank Jeroen van Loon for taking the time to explain the Priva building management systems and assist my understanding of this industry. I'd like to thank the director of ICT International Peter Cull who has helped me with the resources that have helped in construction of this dissertation. I gained a lot of my knowledge about the environmental monitoring industry from working for this business. I'd like to thank the team of engineers at ICT International for giving initial advice on software projects. Finally I must thank my supervisors Dr. Leslie Bowtell and Catherine Hills for their wisdom and guidance.

Table of Contents

List of Figures

List of Tables

Abbreviations

API: Application Programming Interface

BMS: Building Management System

CRC: Cyclic Redundancy Check

DMC: Distributed Measurement and Control

EDA: Environmental Data Acquisition

EM: Environmental Monitoring

EMS: Energy Management System

GPIO: General Purpose Input Output

HMI: Human Machine Interface

I/O: Input/output

NCAP: Network Capable Application Processor

OOP: Object Oriented Programming

OS: Operating System

SDI-12: Serial Digital Interface at 1200 baud

STIM: Smart Transducer Interface Module

TIM: Transducer Interface Modules

UART: Universal Asynchronous Receiver Transmitter

Organisations Cited

Priva: A company who develop and manufacture building management solutions.

OpenEnergyMonitor project: A group of developers building energy monitoring tools.

YDOC: A manufacturer of low-power data acquisition systems suitable to monitor off-grid or hard to reach locations.

Decagon Devices: A manufacturer of sensors and data loggers.

Raspberry Pi Foundation: Foundation to advance the education of adults and children in the field of computer science.

Chapter 1: Introduction

1.1 The Problem at Hand

Individuals, small and large businesses enterprise, and governments have and continue to take from the environment without concern for, desire to address, resources to address or understanding of the long term impacts (MacKay 2008). The general public are not always prepared to invest resources (time and money) if there is no obvious return. Sensing networks have the potential to empower people to understand the environment. Simple control systems can be used to optimise the variables that have a direct impact on the environment. There are many sensing network implementations each with advantages and disadvantages for a specific application class. SDI-12 is a cost effective digital interface for EDA.

A broad range of industry applications are reliant on sensory networking, including; science/research, building management, quality control, waste management, various industrial settings, energy supply, public infrastructure, horticulture, agriculture, aviation, military, mining and boating industries. A Raspberry Pi based SDI-12 logger could be used within both well-established industry applications and by individuals or groups with non-profit applications that would benefit society as a whole. The networking capabilities of the Raspberry Pi increase the potential for use as a data logger.

In modern society, it is important for businesses who harvest raw materials and change the environment in some way, manufacturers of products or providers of services and individuals who use the products or services (who create demand) to understand their impact on the environment.

Sensory networks may be used to provide feedback to automate processes and to empower people with knowledge. The constraints in setting up a successful sensor network to monitor and or control the environment or a process include,

1) lack of user resources (e.g. time, budget and expertise)
2) user has a lack of understanding or desire to change the way they use resources
3) product limitations specifically; product capabilities, flexibility, and complexity.

The development of a monitoring product should address points 1 and 3. Point two is a consequential effect of lack of education.

A constraint in time resources includes the time required for a potential user to source a system that fits the requirement, time in learning the system capabilities, time to setup, install and test the system and time to process, analyse and use the data. Factors influencing the sale cost of a commercial logging package include the scale of manufacture, the market size and competition, the life-cycle management costs, the design features offered, product reliability etc. A cheap and powerful tool such as the Raspberry Pi can greatly reduce the associated life-cycle management costs. Lack of user expertise is not easily addressed in the development of a product which includes difficulties in data analysis, installation requirements and knowledge of sensors technologies (i.e. what technologies are available for any specific application and how it is integrated to an existing system).

Point three is the inherent complexity in providing flexibility i.e. mesh networking, or interfacing to the various wireless sensors with various different protocols, SDI-12 sensors and 4-20mA while keeping the system user friendly. An intuitive graphical user interface will ease the learning curve for computer based products. Point three requires sufficient research and development and good engineering design.

There are unlimited applications for sensor networks. The applications of ethical importance are sustainable living and food production. There is a need for individuals to make changes in the way we use energy. United Nations Environment Programme (cited in Kumar 2013, p 1329) reports that 'buildings consume more than 40% of the available global energy and are responsible for one third of the global greenhouse gas emissions. The main source of green-house gas emissions is energy consumption'. A life cycle analysis of buildings reveals that 80% of the energy used over a building life time is through operation and 20% through manufacture. Monitoring technology can empower people to understand and optimise energy consumption and generation as well as provide a better environment for occupants (OpenEnergyMonitor 2015), (Kumar, Kim and Hancke 2013). Monitoring allow full control of when, how and where energy is used. MacKay (2008) highlights an approach for quantifying our use of energy in full and then calculating the renewables required to provide that energy. His book explores the solutions that are available and encourages people to make individual changes. The main solution is to increase the efficiency of our energy use and to use only sustainable energy sources to power the desired services.

One example of a prime candidate for EDA are commercial greenhouses which are increasing in popularity across the world. Greenhouses allow for greater control over the growing environment of plants and can thereby improve food production in marginal environments. The fact greenhouses are producing more of our food is a direct consequence of a rapidly expanding food demand and change in food eating habits. Blush Greenhouse in Guyra covers an area of 20 hectares and greenhouses around the world are reported to be larger than this. Growers typically monitor soil moisture, electrical conductivity, air humidity, temperature and control the climate through an automatic climate control. System maintainers need flexibility in terms of power and communication between logger and sensors. Sensors need to be moved seasonally and for other reasons.

Personal experience in providing support for environmental monitoring products has found that acceptance of a technology is partially dependent on design that is intuitive and simple to setup, install and configure. It is also important to address practical application issues, such as; minimising wiring and other hardware (sensor network hardware) and maximising the network coverage in built-up and open environments.

1.2 Project Aim

The aim of this project is to develop a low cost and reliable data logger based on the Raspberry Pi computing system that will lead to beneficial outcomes for the environment, businesses, individuals and or communities. The software interface will provide a way of performing basic configuration of SDI-12 sensors, initiating measurements, obtaining results and storing data to a ‘.csv’ file. The software will be available for free and should be modular and simple to understand so that it can be used and modified by others to provide a more fitting solution to any unique and specific SDI-12 logging requirement. The logging system should be simple enough to setup from scratch by a customer with basic knowledge of computers, access to a Raspberry Pi and skills to assemble electrical components that form the SDI-12 sensor interface.

1.3 Objectives

The project will be conducted in the following stages.

Stage 1. Review of environmental monitoring and broad context.

The dissertation investigates the environmental monitoring industry and technologies specifically focusing on the constraints to implementing flexible sensory networks such as; sensing requirements, installation requirements, interfacing hardware, communication channel mediums, communication protocols, environmental influences on communications, and emerging standards.

Stage 2. Conceptual design.

Review of SDI-12 protocol specification, the Raspberry Pi capabilities and programming languages.

Stage 3. Development and Test.

- Design and build a hardware interface to implement the SDI-12 protocol. Test.
- Design and code software modules for the SDI-12 protocol interface and data logger.
- Specify a configuration file format and data storage plan.
- Develop a basic HMI for the data logging system.

1.4: Assessment of Consequential Effects / Implications/ Ethics

As the designer of a product, consideration must be given to the direct and indirect impact it has on the environment, the user and the general public. Scheiber (2001) reports the purpose of manufacturing is to provide:

- The most products
- At the lowest possible cost
- In the shortest time
- At the highest possible quality

The impact an electronics products has on the environment includes the global warming potential resulting from extraction of materials, manufacture, use of and disposal of the unit.

The chief environmental concerns related to the electronic manufacturing industry include the use of energy (electricity), water and nasty chemicals that need to be disposed of particularly in the process of manufacturing microchips and printed circuit boards. A typical facility producing semiconductor wafers reportedly uses 240 000 kilowatt hours of electricity and over 7.5 million litres of water a day. The disposal of the electronics products is inherently difficult because they are not easily disassembled and contain toxic materials (lead for example) resulting in large amounts of waste that cannot be recycled. Sustainability of the industry is a concern due to the risk in polluting the environment.

The software is intended to be distributed for free without any hardware. The Raspberry Pi is purchased by a customer as a circuit board with no enclosure. Third-party enclosures developed specifically for the Raspberry Pi are available but an extensive search has not revealed a water proof model. Water proof enclosures not specifically for the Pi are available in all shapes and sizes that comply with standards such as IP64 and higher but the customer would need to mount the Raspberry Pi and cut holes for cables.

As the hardware is third-party and with no intension to sell with software, it is easy to disregard the consequential effects related to its manufacture. The positive implications of using a cheap, off the shelf third party computer platform is significant in terms of initial development costs and overall reduces the life cycle management cost related to;

- Product development
- Manufacturing
- Test
- Service
- Field returns
- The company's "image of quality"

The negative implication resulting from offering a software solution without the hardware is that the consumer is required to source, assemble and modify hardware. Sourcing hardware is can be difficult and without knowledge in handling circuit boards the computer can easily by fried by ESD. An indirect implication to a solution based on a cheap Raspberry Pi without an enclosure is that a consumer would possibly place less importance on protection of the equipment as it could easily be replaced. Less importance on protection has the potential to result in a high failure rate for example due to corrosion from inadequate protection in outdoor settings or humid environments. In designing of a water proof enclosure, pressure differences between the inside and outside of the enclosure due to rapid changes in temperature need to be considered. Pressure difference can draw water inside the unit. A short-lived computer that is continually replaced will have an impact on the environment.

Quality research and development will ensure a product lasts for a long time and is competitive. Part of research and development is testing of individual modules under a range of conditions to identify problems. The statistics on the failure rate of the Raspberry Pi is not likely to be available or accurate as the customer may not claim the warranty due to its low cost. An electronics product should be tested for a range of external influencing variables including temperature extremes, temperature cycling, vibrations, water ingress and more.

1.5 Existing Solutions

The consumer is demanding devices, and systems with better capabilities and high levels of functionality. Existing monitor solutions, hardware and applications are considered.

1.5.1 Low Cost Raspberry Pi Based Monitoring Solutions

There are many made-for-consumer-kits coming into the market based on either the Raspberry Pi computer or Arduino microprocessor or both. Both the Arduino and Raspberry Pi are small, single board computing platforms that simplify the process of producing cheap devices that sense and control the real world. The Arduino is an open-source computing platform consisting of a micro-controller board and a development environment for writing software. This Arduino platform is cheap at around $30, and makes the process of working with microcontrollers and prototyping easy. The Arduino cannot implement complex programs because of limitations in memory and computational power. The Raspberry Pi is a computer which is based on the Broadcom BCM2835, it includes a 900 MHz quad-core ARM Cortex-A7 CPU and a GPU. The Pi delivers a lot of performance per watt and costs around $40. With a Linux operating system installed on the Raspberry Pi developers are able to program on board hardware through the application programming interfaces (API). There is some effort required in learning the Linux operating system required to develop and test software. Both platforms have general purpose inputs and outputs for sensing and controlling. The wiringPi GPIO utility from Gordon Henderson assists in the development of software for the Raspberry Pi allowing the state of pins to be checked through simple terminal commands (Henderson 2015). This program simplifies experimenting with hardware.

The OpenEnergyMonitor project have developed open-source tools for energy monitoring and analysis. It is a project aimed at empowering people to monitor energy use (OpenEnergyMonitor 2015). The OpenEnergyMonitor tools are based on both the Arduino and Raspberry Pi and designed to be a cost effective solution for monitoring energy use. The main monitoring system consists of sensors connected to an Arduino. Data is transmitted to a Raspberry Pi wirelessly (using the RFM69CW module) and this information is then stored locally or remotely for viewing and analysis. It is designed to be easily configured using only supported systems and supported sensors although anyone can freely use and modify the software as it is open-source. The system is not currently capable of outputting control signals.

Although it is possible to build the system using an Arduino, a Raspberry Pi and the available online resources this would not be typical. The consumer kits which are ready to go preinstalled with software and include a protection case and other electronic hardware are reasonably priced. The logger base (EmonPi) is $350 and the wireless sensing node is $150 excluding sensors. It would be better for a consumer to have freedom of choice as to what sensors could be interfaced, and allow some level of control. It does not allow SDI-12 sensors but offers flexibility in monitoring of indoor environments.

1.5.2 Low Cost SDI-12 Loggers

A search of the internet reveals a small number of low cost loggers capable of SDI-12. One of the better solutions is the ML-315 data logger manufactured by YDOC with the basic system costing about $550. Decagon Devices and Campbell Scientific also produce loggers with SDI-12 capabilities. Decagon Devices' loggers are around $500 but are limited to a maximum of 5 sensors per logger and limited to decagon sensors only. Campbell Scientific offer more expensive solutions.

1.5.3 Building Management Systems

Priva Group and Schneider Electric are two companies that provide intelligent building management system (BMS) solutions. Priva develop and manufacture intelligent BMS that assist in maintaining healthy living and working environments and sustainable energy use. Priva (2015) indicates the function of their BMS is to:

- Optimise energy flows, and provide up-to-date energy use information
- Allow you to monitor and manage the indoor climate, energy, lighting, fire, security, and other comfort controls.
- Automate processes.

On 23 April 2015, Mr J van Loon, provided me with information about the Priva BMS systems. He indicates that Priva are able to provide solutions to automate processes in buildings. He says the Priva BMS solutions have analogue 0-5 volt inputs for connecting analogue sensors. The Priva BMS is capable of logging the input signals. Priva provide software for configuring control systems and data viewing remotely. The Priva BMS is a full feature solution but not aimed at domestic home owners as the solutions are expensive and more robust then needed in domestic installations. He said that there is potential for using an intermediate control system between the Priva hardware interface and sensors to make use of smart sensors.

1.6 Potential Project Outcomes

The benefit of a Raspberry Pi based logger over some of the commercial solutions include its small physical form (it's about the size of a credit card), the Linux operating system and the software available for it including development environments, the general purpose inputs/outputs (GPIO), its network capabilities and its low cost which can potentially be leveraged by the consumer or a business providing monitoring solutions or solutions to address unique requirements such as automation. The top of the line Raspberry Pi computer is available internationally for $35 US + shipping + local taxes and is sold through distributors all over the world (Raspberry Pi 2015). A consumer kit could be developed to save the consumer some work and sold for a small profit. The basic system may not need to be sold and a business could be based on creating solutions to specific applications and unique requirements such as automation. A simple base design with software modularity will allow easy modification of the software to suit any particular application.

There are expanding markets and or opening markets for effective low cost EDA/simple automation systems. These markets include building management systems (or integration into building management systems as discussed in section Existing Solutions), energy monitoring, agriculture, simple automation and environmental research industries. Low cost systems open the potential market to the general public who may be involved in projects related to sustainable living, production of food, automation of processes, or monitoring and analysing of the environment for comfort and optimisation and many more. Certain small and large business enterprise will also benefit greatly from low cost solutions. Use of low cost third party hardware will reduce the manufacturing and testing costs. The lifecycle management of the Raspberry Pi based system is also simplified and less costly (see section: Assessment of Consequential Effects / Implications / Ethics).

SDI-12 is good choice in applications where many smart sensors need to be connected to a single input (Decagon Devices 2015). It is a cost effective way to read data from many sensors. It simplifies the programming and hardware requirements of the logger. SDI-12 sensors can measure multiple parameters and return the values in engineered units. SDI-12 sensor are often produced to return multiple parameter measurements. The complete range of SDI-12 sensors produced by Decagon Devices return between 3 and 6 parameters per sensor. An example sensor is the Decagon GS3 which measures soil dielectric permittivity (unit: e), temperature (unit: ℃) and electrical conductivity (unit: dS/m). Smart sensors that take multiple parameter measurements are particularly cost effective if all multiple parameters are needed.

Bus based systems allow some flexibility required in the applications areas mentioned but there are some shortcomings in practical applications that could be addressed to increase the flexibility and thus acceptance and popularity of the bus based monitoring and SDI-12. A hybrid wireless SDI-12 host-to-SDI-12 bus is a potential product to address the unique requirements in field monitoring applications such as large greenhouses where sensors need to be moved, keeping cables out of the way, fitting systems within the environment and sending data over long distances. This could be implemented in a variety of environments to utilise the strengths of both wired and wireless systems. An extensive search does not find an existing SD1-12 host and slave wireless hybrid adaptions. Bus based system are suited to indoor and industrial environments. A well designed digital electrical interface should be very immune to noise that exists within building.

The Raspberry Pi's networking functionality makes it extremely useful for monitoring over the internet. People are connected to the internet in more ways through a wider variety of devices and with the SMART phone have convenient 24/7 access. Alarms can alert an individual via email or SMS. The Raspberry Pi can be accessed remotely to remove alarms. The Raspberry Pi has additional on board devices to enable video processing which can be used to provide visual confirmation about systems.

1.7 Project Methodology

This project is underpinned by an experimental methodology. The three key stages of the project include conceptual design, development and testing of software and hardware. The software will be developed iteratively with a series of experiments that develop an expanding body of knowledge. The software is implemented in C++ and consists of; specifying a configuration file format and modification of C++ Arduino SDI-12 library. The configuration of the logger channels, and initiating measurements is done through a command line based human machine interface (HMI).

1.7.1 Research

The research part of the methodology is as follows:

- Research the hardware capabilities of the Raspberry Pi GPIO pins including the driving and loading limits.
- Assess both the UART and GPIO pin approach to SDI-12 asynchronous serial communication. Look for existing accounts where the GPIO of a Raspberry Pi is used to communicate serial data. Inform an approach based on the difficulty and other advantages and disadvantages identified.
- Review of SDI-12 protocol specifications and capabilities.
    - Assess the robustness of the SDI-12 protocol in terms of not error detection.
    - Calculate the bus impedances and find the driving and loading limits.
    - Identify the commands that will be issued
    - Assess the format of a sensor response to each command that will be issued.
- Review literature on C++ and also for developing applications for the Raspberry Pi.

1.7.2 Prototype Implementation

Hardware is required to translate from Raspberry Pi TTL logic to the SDI-12 data line voltage levels (0-5.5 V). Prototyping will also involve testing and experimenting with manipulating the GPIO pins. The prototype part of the methodology is as follows:

- Experiment with the Raspberry Pi GPIO pins and confirm the informed approach to SDI-12 asynchronous communication is possible.
    - Experiment with timing for changing GPIO pins from inputs to outputs.

    - Investigate the operating system scheduling interruption frequency and period that the interruption lasts. Write a test program to toggle 1's and 0's and analyse the output using an oscilloscope set to trigger for pulses that are greater than a set period. If the interruption are severe the UART approach may be more suitable.
    - Investigate and experiment with the Raspberry Pi interrupts.
- Investigate hardware options for performing bi-directional communication and level shifting from 3.3 volts to 5 volts and from 5 volts to 3.3 volts.
- Design hardware and implement using a breadboard.
- Test hardware. Develop test software which writes toggles a digital HIGH and LOW to a GPIO pins and analyse the signal using an oscilloscope. Connect any hardware and test at the output. Load the hardware and retest.

1.7.3 Software Development

With no previous C/C++ programming experience knowledge will be gained through analysing existing code, experimentation and testing. Face-to-face mentoring is suggested to be a good way to learn how to program (Agile manifesto 2015). It would be more effective learning to code through experienced software developers at work or practical based training programs. The program will be written in C++ using the Geany development environment for Linux which is installed on the Raspberry Pi. Skills in programming can be developed by writing small test programs. Individual parts of the C++ code will be tested with a range of input conditions to ensure each section will work as intended.

- Design and code software modules for the SDI-12 protocol interface and data logger.
    - Assess the logical sequences for SDI-12 exchanges, and identify an approach to:
        - Setting states of the GPIO pins for transmitting and listening if this approach is used. Both GPIO and UART approach would be different.
        - Sending commands. (A command is a string of ASCII characters. Each ASCII character is sent in a frame with 7 data bits, 1 parity bit, a start and stop bit)
        - Receiving ASCII characters
- Specify a configuration file format and data storage plan.

    - SDI-12 devices do not return extensive information about themselves in response to the identification command so a user will have to manually configure the SDI-12 channel by giving it a name, unit, and specifying which result from that sensor corresponds to it. This information will be stored for each logger channel in the configuration file.
    - Research how to read and write from a file.
    - Identify a sequence of configuration file entries that will allow the logger to perform a measurement from each configured sensor address and return the parameter(s) to its corresponding channel.
- Develop a main program and command line based HMI for configuring the data logger.
    - Use the SDI-12 functions to send the measurement commands to configured sensors at measurement intervals and append data to the data file.
    - Allow basic configuration of sensors.

1.7.4 Final Testing and Evaluating the System

- Test the logger with 3 SDI-12 sensors
    - Manipulate the configuration file parameters to see that data is stored correctly.
    - Set the device to log for an extended period of time and analyse the results.

Chapter 2: Background Information

The development process must give consideration to the various technologies, techniques and principles. The first section of this chapter gives background information on environmental monitoring systems. The second section of this chapter looks at SDI-12. The third section gives background information on the Raspberry Pi and an assessment of the GPIO driving capabilities.

2.1 Review of Environmental Monitoring Systems

> Major factors that usually rule the development of Environmental Monitoring (EM) systems are; energy efficiency, cost of the overall system, response time of the sensor module, good accuracy of the system, adequate signal-to-noise ratio, radio frequency interference/electro-magnetic interference (RFI/EMI) rejection during varying atmospheric conditions and in inhomogeneous environments, a user friendly interface with the computer, and complexity of computation (Kumar et al. 2013).

2.1.1 Overview of Environmental Sensors

SDI-12 was designed for environmental data acquisition (EDA) (SD1-12 support group 2015). Classification of sensors is conventionally by physical quantity being measured, the conversion principle, technology used, or by application (Sinclair 2001). Environmental monitoring sensors can also be classed according to broad physical characteristics of the environment it is measuring. A sensor can be discriminated as measuring either biotic or abiotic environmental quantities. Sensors measuring abiotic quantities can be further categorised according to the physical properties it is measuring for example soils, atmosphere, water and manmade environments or systems. I.e current through a wire conductor would be a manmade system. This is not a perfect organisation for example a temperature sensor can exist in all four categories. An abiotic environment may be related closely to a biotic environment so environmental sensor distributors may choose to separate sensors by application also.

Examples quantities measured by atmosphere monitoring sensor including light intensity, radiation, pollution, temperature, noise, humidity, wind-chill, wind direction, wind speed, oxygen and many more. Physical quantities measured by soil sensors include moisture, electrical conductivity, temperature, acidity, heat flux and many more.

2.1.2 Smart Sensors

An analogue sensor is a transducer that detects a physical, biological, or chemical parameter and responds with an electrical signal. Electrical signals from a sensor may need to be conditioned in the sensor. Signal conditioning is a process involving filtering, amplification, compensation and normalisation. Further processing of the conditioned signal may take place in the data system such as adaptive noise cancelling, spectrum analysis or any other algorithm needed for a specific purpose (IEEE Standards Board 1997).

Sensors can integrate microprocessors, ADC’s and other electronics, which are getting smaller, cheaper and more powerful into the sensor to form a ‘smart’ sensor. A microprocessor in a sensor may calibrate the sensor, control the sensor measurement, control a process and measure the process variable(s) and convert raw sensor reading into engineering units (SD1-12 support group 2015). A smart sensor is able to communicate measurements directly to a data system. The smart sensor makes the job of the application engineer easier by shifting the task of designing the signal conditioning, and complex process control and measurement functions to the sensor manufacturer. A smart sensor is able to supply the analogue sensors with any specific power and voltage requirements using internal regulators that would otherwise be difficult to do with from the data node. Assuming the digital interface between smart sensor and the data system is well designed there should not be data corruption due to noise pick up. Furthermore, smart sensors can be networked.

2.1.3 Sensor Networks

Sensor network are distributed autonomous sensors to monitor physical quantity which cooperatively pass their data through the network to a main location. The types of nodes in a sensor network are; sensor nodes, data nodes and aggregator node. Sensor nodes are a ‘smart’ sensor composed of one or more sensors and a communication device and do not store data. SDI-12 sensors are an example of a sensor node Data nodes are sensor nodes that store data. Aggregator node are composed of a communication device, a recording device and no sensors. The three types of nodes in a sensor network are shown in figure 1.

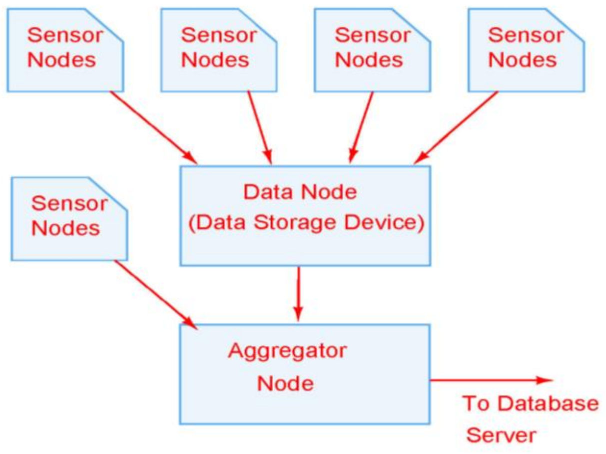

Figure 1 Types of nodes in a sensor network (Bell 2013, p. 27)

Data is communicated between nodes either through a wired or wireless medium. Hybrid systems combine both wireless and wired technologies.

Wired Networks

Many implementations of wired networks are currently available. Digital sensor buses were developed mainly due to a need in the process and control industry to connect sensors and transducers directly to digital networks for factory automation and closed loop-control (Write and Dillon n.d.). Write and Dillon (n.d.) states the 'large growth in slow speed sensors (for the measurement of temperature, pressure and position for example) contributed to the development of digital bus architectures such as Fieldbus, Profibus, LonWorks, and DeviceNet'. He said that, 'these systems have drawbacks and problems like bandwidth limitations, proprietary hardware and the major design work needed to interface with existing sensors'. Other wired communication protocol implementations include TCP/IP, BASnet, Modbus and Ethernet. Wired systems in general have drawbacks relating to the need to purchase and route cables which add costs in installation, maintenance and upgrade.

Wireless Networks

Wireless communication is more versatile than wired communication. Wireless communication protocol standards such as 802.11 (WiFi), 802.15.1 (Bluetooth), 802.15 4(ZigBee & XBee) are some of the options. Radio performance is affected by obstacles in and around the transmission path. ZigBee devices can transmit data over long distances by passing data through a mesh network of intermediate devices to reach more distant ones.

Hybrid Network

Hybrid networks combine both wired and wireless medium. An example maybe an interface for SDI-12 logger-to-SDI-12 sensor/bus. The slave interface at the sensor site would power the sensors from an internal battery or external source and provide the SDI-12 protocol messaging and timing. The master interface can be powered from the host (logger). This would allow use of SDI-12 solutions for more widely distributed monitoring applications. An SDI-12 bus can be routed to a well position wireless SDI-12 slave module providing a lot of flexibility.

2.1.4 Distributed Measurement and Control – Towards Networked Smart Sensors

Applications for control systems can take several forms. The most notable control environments are segregated (or individual), centralised, or distributed control (Lee & Schneeman 2000, p. 623). The *segregated* (or *individual)* control functions as a standalone system in a non-networked environment. An *individual* control node may control one or more processes which will be application specific. The centralised and distributed control forms function in Distributed Measurement and Control (DCM) environments. The *centralised* control form is where a master controller directly controls slave nodes on a network. The process connection is thorough the slave nodes. The *distributed* control form is where sensor information is sent to a networked nodes through a controller gateway(s). A *distributed* networked node is able to process information on the network and make a control decision. A distributed control networked node has distributed intelligence.

Further work outside the scope of this project involves the development of the Raspberry Pi logger to work as a node at the process control level in a DMC environment. The Suitability of the Raspberry Pi as a process control node of DMC environment will depend on measurement and control requirements, overall system frameworks and technologies used.

The Smart Sensor Interface Standard (IEEE1451)

> The distributed measurement and control industry is migrating away from proprietary hardware and software in favour of open source systems and standardised approaches (LEE & Schneeman 2000, p. 621).

The *'IEEE standard for smart sensor transducer interface for sensors and actuators'* (IEEE 1451), is a standard to define common interfaces for connecting transducers to microprocessor based systems, instruments and field networks.

> IEEE Standards Board (1997, p. iii) states, *the main objectives of the IEEE 1451.2 standard are to:*
>
> - *Enable plug and play at the transducer level by providing a common communication interface for transducers.*
> - *Enable and simplify the criteria of networked smart transducers.*
> - *Facilitate support of multiple networks.*

IEEE Standards Board (1997, p. iii) states: 'The existing fragmented sensor market is seeking ways to build low-cost, networked smart sensors.' There are many implementations of the control network each with its own strengths and weaknesses for a specific application class. IEEE Standards Board (1997, p. iii) reports that, 'interfacing sensors to all the control networks and supporting a wide variety of protocols represents a significant and costly effort to transducer manufacturers'. Existing control networks include digital bus protocols such as Fieldbus, Profibus, LonWorks, Modbus and DeviceNet, TCP/IP, LonWorks, BASnet, Ethernet and many more. Wireless protocols used include WiFi, Bluetooth, ZigBee & XBee. The IEEE 1451 standard objective is to develop an interface standard that will 'isolate the choice of transducer from the choice of networks' (IEEE Standards Board 1997, piii).

Transducer manufacturers may produce a sensor which can interface to one or more of the control networks or may even adopt a proprietary communications interface that limit the data available to higher-level networks. The Smart Sensor Interface Standard (IEEE 1451) defines a common interface for connecting transducers to control networks. It is implemented at the lowest level of an industrial automation environment, the *process connection* level, and the next level up, the *distributed intelligence* level. The *application level* is the third and final level which is not defined in the IEEE 1451 standard. A functional block diagram of IEEE 1451 is given in figure 2 and shows 5 functional sub-sections of the IEEE 1451 standard.

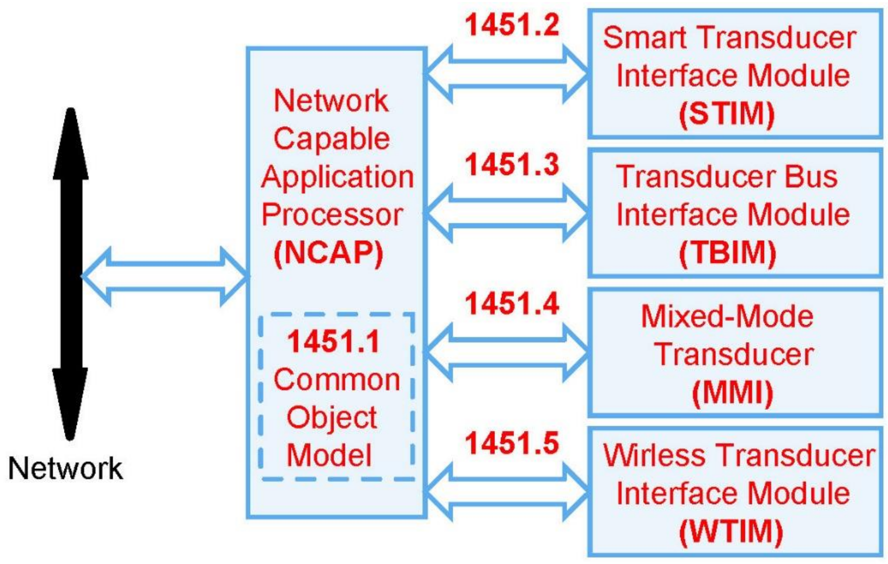
1451.2
Smart Transducer Interface Module (STIM)
Network Capable Application Processor (NCAP)
1451.3
Transducer Bus Interface Module (TBIM)
1451.4
Mixed-Mode Transducer (MMI)
1451.1 Common Object Model
1451.5
Wirless Transducer Interface Module (WTIM)
Network

Figure 2 Functional block diagram of IEEE 1451 (Wright & Dillon n.d. p2)

The first section of the IEEE 1451 standard specifies an interface for connecting the transducer interface modules (TIM's i.e. STIM, TBIM, MMI and WTIM) in figure 2 to a network that performs network communications to transducers for performing data conversion. The NCAP is a network node (see section: *Distributed Measurement and Control*). The transducer(s) which are connected to the TIM's may be analogue or digital. Analogue and digital sensor data is converted using analogue-to-digital converters (ADC) and digital input/output (DI/O) ports. Data can also be transmitted wirelessly between transducer and TIM as specified in the IEEE 1451.5 subsection. Figure 3 shows the context of a Smart Transducer Interface Module (STIM) as specified in IEEE 1451.2. A STIM module contains Transducer Electronic Data Sheets (TEDS), logic to implement the transducer interface, transducer measurement interface and any signal conditioning. The STIM can accommodate 255 channels. Wright & Dillon (n.d., p. 5) said 'the success of IEEE 1451 depends on the development of the TIM's to meet individual needs and special applications'. The IEEE 1451 standard specifies a TEDS to be stored within the TIMs. TEDS are blocks of information stored in non-volatile memory within a TIM that describe a transducer. The IEEE 1451 standard does not specify the signal conditioner and data conversion functions, TEDS reader function or transducer measurement interface. It is possible to develop a STIM specifically for SDI-12 sensors which return a single parameter measurement to a STIM channel, however, further investigation is needed to determine whether SDI-12 sensors that return multiple parameter measurements can be connected to a STIM channel.

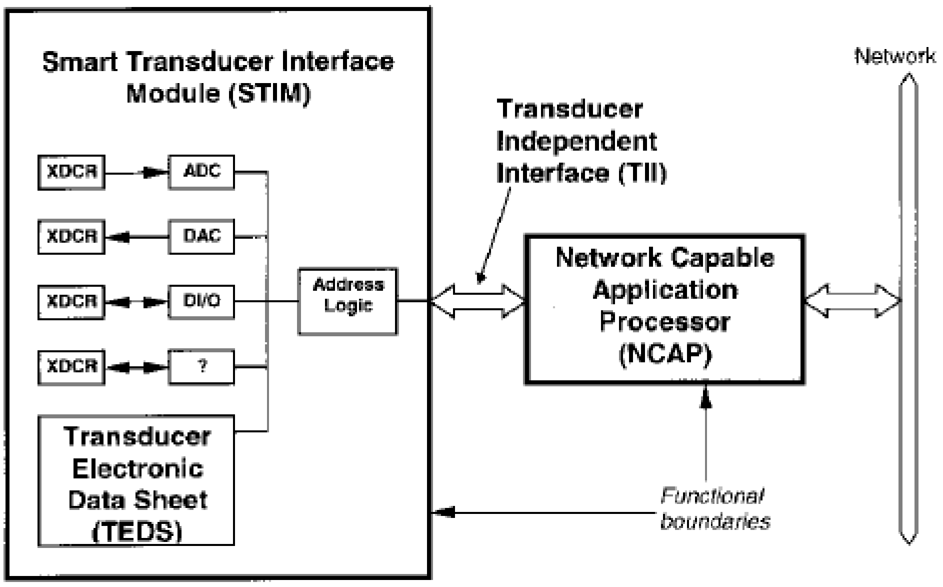
Smart Transducer Interface Module (STIM)
XDCR
ADC
XDCR
DAC
XDCR
DI/O
XDCR
?
Address Logic
Transducer Electronic Data Sheet (TEDS)
Transducer Independent Interface (TII)
Network Capable Application Processor (NCAP)
Network
Functional boundaries

Figure 3 Overview of the STIM (IEEE Standards Board 1997)

2.2 Overview of SDI-12

This section describes an interface between data loggers (data nodes) and microprocessor based sensors (sensor nodes).

2.2.1 Introduction

SDI-12 stands for Serial Digital Interface at 1200 baud. SDI-12 is a three wire serial digital interface providing a means for transferring measurements taken by an SDI-12 sensor to a data node for environmental data acquisition. The SDI-12 standard defines an electrical interface, protocols and timing (SD1-12 support group 2013). The protocol describes the normal data retrieval operation on the SDI-12 bus using a command set specified in the standard. SDI-12 sensors are an intelligent microprocessor based sensor which use SDI-12 to talk with data nodes. Only the data node can initiate communications. While all SDI-12 sensor must conform to the SDI-12 specification, it is not a plug and play type system. Detailed knowledge of the sensor such as the amount of data the sensor is returning and the order in which the data is returned is required to configure it.

2.2.2 Background

The SDI-12 protocol grew out of a need for a low power, standard serial interface for serial-data acquisitioning (SD1-12 support group 2015). The first version of SDI-12 specification was created in 1988 with input from a group of companies that were operating in the environmental monitoring industry but mostly written by Campbell Scientific. SDI-12 has been refined over the years but remains backward compatible with earlier versions. The SDI-12 support group maintains the SDI-12 standard (SD1-12 support group 2015).

2.2.3 Electrical Interface

The SDI-12 protocol defines a multi-drop, multi-parameter interface. Multi-drop means that one or more SDI-12 sensors are connected on a single cable as shown in figure 4. Sharing a communication wire is often called a ‘bus’. Multi-parameter means each of the SDI-12 sensors is capable of measuring one or more parameter and returning multiple values to the data node.

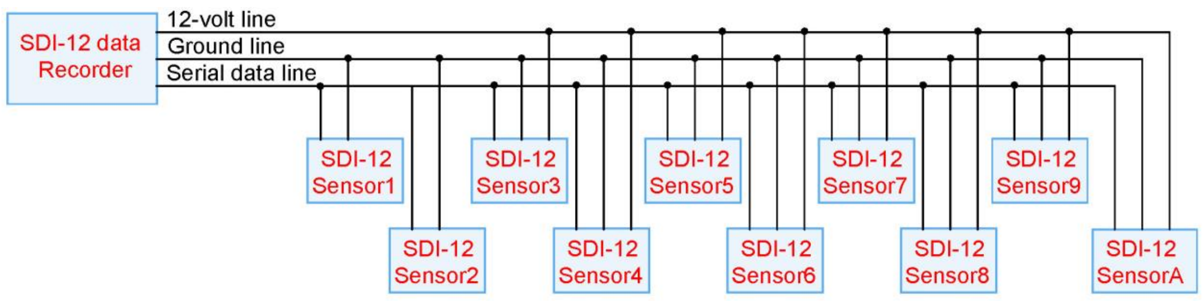

Figure 4 Physical context of the SDI-12 interface

Each sensor on the multisensory bus has a unique address with a maximum of 62 addresses on the bus. SD1-12 support group (2013) specifies that an SDI-12 bus with 10 sensors would limit the maximum cable length per sensor to 60 meters but indicates that using fewer sensors means a longer length of cable per sensor is possible. This means that 10 SDI-12 sensors could be placed anywhere within a 60 meter radius of the data node if the cable goes directly between each sensor and the data node. Decagon Devices produce SDI-12 sensors with a 'low impedance' variant of the equivalent circuit recommended in the SDI-12 specification (Decagon Devices n.d.). Decagon Devices (n.d., p. 5) states that 'this allows for up to 62 sensors to be connected onto the bus at one time instead of the 10 that is stated in the standard'. The recommended equivalent circuit and the Decagon Devices 'low impedance' variant are discussed in the serial data line section below. The complete range of Decagon Devices' SDI-12 compatible sensors also specify a different logic HIGH requirement which also allows more SDI-12 sensors to be connected to the bus. The Decagon Devices SDI-12 sensor digital input variant is also discussed in the serial data line section below.

Line Definitions

The SDI-12 communication is done over a single data line. The 3 physical connection are,

1) 12-volt line
2) a ground line
3) a serial data line

Electrical Specifications

12-Volt Line

a) The voltage on the 12-volt line shall be between 9.6 V and 16 V with respect to the ground line under maximum sensor load of 0.5 A.

b) The data node is expected to be able to supply a maximum sensor load power 0.5 amperes. The SDI-12 protocol initiates communication by waking all sensors on the SDI-12 bus therefore the 12 volt line must power all devices at once for a small period of time. While all devices do not take a measurement at this time or at once it is possible that the SDI-12 control circuit in the sensor (typically consisting of a micro-processor or FPGA) may need up to 50 mA, however, for low power sensors using low power 8 bit microprocessors it is likely to be less than 50 mA. The complete range of Decagon Devices' SDI-12 compatible sensors (approximately 10 in total) require between 10 and 25 mA of current during a measurement and up to 0.3 mA quiescent (Decagon Devices 2015).

c) An SDI-12 sensor may use separate power supply as necessary.

Ground Line

a) The ground line must be connected to the data node circuit ground and an earth ground at the data node. The sensor circuit ground is connected to the ground line. The sensor circuit ground is usually isolated from ground (frame or earth).

b) The ground line should be large enough so that a maximum voltage drop along line between sensor and data node is 0.5V for combined sensor current drain.

Serial Data Line

Logic Levels

Digital signals representing data and control signals can be either a one or zero and are represented physically by two voltage levels. The term logic HIGH refers to a HIGH voltage level for either positive or negative logic. The SDI-12 data line uses negative logic. Events are associated with changing logic, a LOW-HIGH transition shall be referred to as a positive edge and a HIGH-LOW transition a negative edge. The serial data line use a single bi-directional data line with three states. The SDI-12 data transmission logic and voltage levels are shown in table 1. Bi-directional means there is two-way communication on a shared data line, which is possible using precisely timed signal conditioning. Decagon Devices is a sensor manufacturer that produce SDI-12 sensor which have a lower binary HIGH than those specified in table 2. Decagon produced SDI-12 sensors that read a low asserted between 2.8V and 3.9V. On 24 June 2015, Mr M Galloway, assured me that this does allows more sensors to be connected to the bus.

Table 1: SDI-12 voltage thresholds (SDI-12 support group 2013, p. 3)

Condition	Binary State	Voltage range
marking	1	-0.5 to 1.0 volts
spacing	0	3.5 to 5.5 volts
conditioning	undefined	1.0 to 3.5 volts

Drive and Loading

The recommended SDI-12 equivalent circuit is shown in figure 5. This allows a maximum of 10 sensors to be connected on the bus. For a bus with 10 sensors connected the AC impedance seen by the transmitter is approximately 4.1kΩ∠-58°. The AC impedance is calculated using a sinusoidal approximate to the digital square wave at frequency 1200Hz which is an approximate to the SDI-12 baud data rate of 1200 baud. When the transmitter is driving towards a HIGH state (MAX 5.5V transmitter output) it must be capable of sourcing a maximum of 1.33 mA. The voltage on the data line with 10 sensors connected would be approximately 3.6 volts if driven at 1200 Herts. This is at its lower HIGH voltage specification (see table 1) while the transmitter output is at its upper specification (5.5V). Extra loss not accounted for includes the impedance between bus and sensor, conductor line resistance and stray capacitance on the data line. The 10 sensor limit specified in the SDI-12 standard leaves no tolerance and requires a transmitter outputting 5.5 V. The DC impedance seen by the transmitter with 10 sensor is 19.3kΩ thus the transmitter driving a HIGH state would only need source 285µA but this is unrealistic.

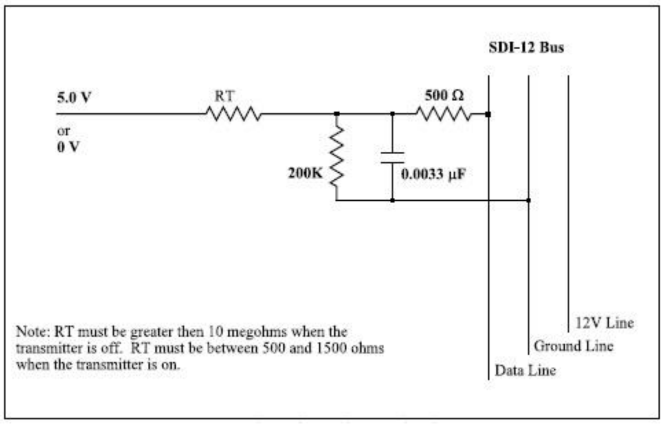
SDI-12 Bus
5.0 V
or
0 V
RT
500 Ω
200K
0.0033 µF
Note: RT must be greater then 10 megohms when the transmitter is off. RT must be between 500 and 1500 ohms when the transmitter is on.
12V Line
Ground Line
Data Line

Figure 5 Recommended equivalent SDI-12 circuit (SD1-12 support group 2013, p. 4)

The Decagon Devices 'low impedance' variant on the recommended equivalent circuit is shown in figure 6. This circuit has a lower DC impedance in comparison to the DC impedance of the recommended circuit but a higher AC impedance at 1200 Hz. For a bus with 10 sensors connected the AC impedance seen by the transmitter is approximately 9.9kΩ∠-9°. When the transmitter is driving towards a HIGH state (5.5V) it must be capable of sourcing 540 µA, which is much less than required if sensor use the recommended equivalent circuit. The voltage on the data line with 10 low impedance variant SDI-12 sensors connected would be approximately 4.72 volts if driven at 1200 Hz. The capacitive reactance of the Decagon low impedance variant equivalent circuit is reduced and so current and voltage are almost in phase. This may increase the slew rate. The DC impedance seen by the transmitter is 10kΩ.

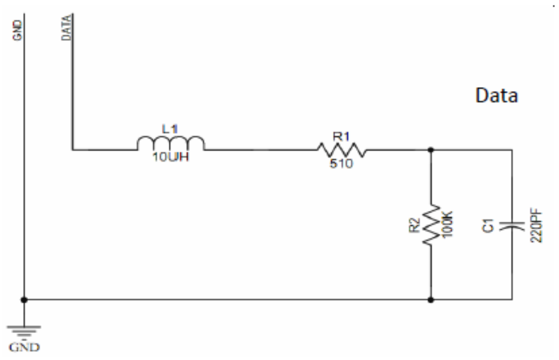

Figure 6 Decagon Devices' low impedance equivalent SDI-12 circuit (Decagon Devices n.d., p. 2)

Voltage Transitions

The slew rate on the SDI-12 serial data line must not be greater than 1.5 volts per microsecond.

2.2.4 Protocol

The SDI-12 communication is done over a single data line (the ‘serial data line’) via precisely-timed signal conditioning, resulting in an exchange of ASCII characters as defined by the standard. The SDI-12 protocol allows for a maximum of 62 sensors to be connected to the bus. Sensor measurements are triggered by protocol command. Communication is addressed specifically to each sensor. Each sensor requires a unique ASCII character address. The valid addresses are ASCII characters 0-9, a-z and A-Z giving a total of 62 unique addresses. To add more than one SDI-12 sensor to a system the address of each sensor should be changed while no other sensors are connected to the bus.

Baud Rate and Byte Frame Format

The SDI-12 protocol sends characters at 1200 baud. Each byte frame has 1 start bit, 7 data bits (LSB first), 1 parity bit (even parity) and 1 stop bit. For even parity, the number of bits whose value is 1 are counted. If that total is odd, the parity pit is set to 1, making the total count of 1's in the set an even number. If the count of 1's in a given set of bits is already even, the parity value remains 0. The SDI-12 protocol uses negative logic. An example of a transmission of character ‘a’ is shown in figure 7. The 7 bit binary code for ASCII character ‘a’ is 110 0001.

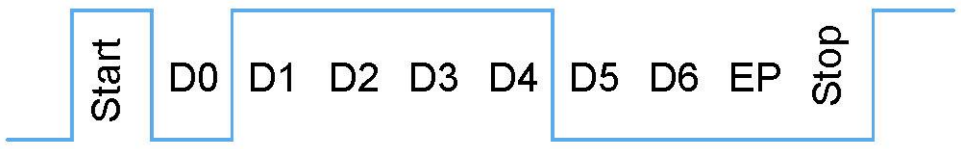

Figure 7 Example SDI-12 transmission of character 'a'

SDI-12 Timing

All SDI-12 commands and response must adhere to the timing diagram given in figure 8.

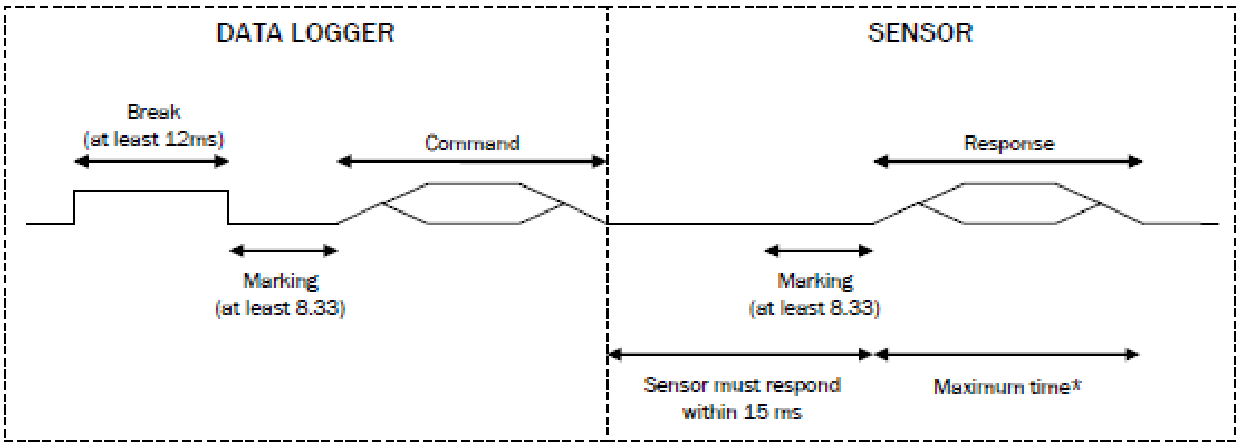

Figure 8 SDI-12 timing (SDI-12 support group 2013, p. 24)

The maximum time for a sensor response to all but the concurrent measurement is 380 ms. The SDI-12 interchanges follow the general pattern presented here:

1) The data node wakes all sensors by placing a *break* on the SDI-12 bus. 'Break' is the name of a command for a continuous spacing for at least 12 milliseconds. There is no upper limit on the break period. A sensor must wake within 100 ms after detecting the break.
2) The data node sets the data line to marking (logic LOW) for at least 8.33 ms. A sensor can go back to sleep after 100ms of marking so the upper limit on marking can be say 90ms.
3) The data node announces an SDI-12 command (see table 3) to a specific sensor and immediately waits for the reply.
4) If the addressed sensor is awake and has detected the command it will set the data line to marking for 8.33 ms immediately followed by the transmission of the command response. The response must begin within 15 ms of receiving the command stop bit of the last byte . If there are other sensors on the bus and detect an invalid address they must return to standby.
5) The data node captures the response. If the response is not received the data node will retry after a minimum of 16 ms has passed since the last stop bit of the command but no longer than 87 ms. At least two more retries will be attempted with at least one being 100 ms from the transition from break to marking.

SDI-12 Commands and Responses

A subset of the SDI-12 commands and sensor responses are given in table 2. The commands listed in table 2 will be issued by the prototype Raspberry Pi logger which will allow automatic configuration of SDI-12 sensors and logging of configured sensors. The first character of all command issued by a data logger and the first character in the sensor response is the sensor address. The last character of every command is always an exclamation mark (!). The final characters of all sensor responses are carriage return and line feed (<CR><LF>). Sensor that comply with the latest standard have a variant command that includes a three character CRC with the response. These variants may be implemented in the future.

Table 2: SDI-12 Command set and response format (SDI-12 support group 2013, p. 8)

Command Name	**Command String**	**Response**
Address Query	?!	a<CR><LF>
Send Identification	aI!	allccccccccmmmmmmvvvxxxxxxxxxxxxx<CR><LF>
Send Data	aD0-9!	a<value(s)<CR><LF>
Start Measurement	aM!	atttn<CR><LF>

Table 3: Key for table 3 response characters

Key	**Meaning**
a	Address
l	SDI-12 version number
c	8-character vendor identification
m	6-characters indicating sensor model number
n	Number of data values being returned
t	Time in seconds until data will be ready (when service request should be issued)
v	3-character sensor version number
<value(s)> pd.d	P – polarity sign d- numeric digits before the decimal place . – decimal point (optional) d – numeric digits after the decimal point
<CR><LF>	Terminates the response

Benefits in Using SDI-12 in Environmental Data Acquisition

Using SDI-12 provides considerable benefits, which include plug-and-play modularity, availability of a growing number of SDI-12 sensors, one data logger port for connecting different sensors and the benefits of smart sensors as outlined in the section 'Smart Sensors'. In essence using SDI-12 will simplify the process of installing and configuring sensors. Using SDI-12 sensor nodes simplifies the hardware requirements of the data node as the measurement channels and signal conditioning is part of the sensor. SDI-12 sensors may control a complex process and measure the process variable(s) which are used to determine the desired quantity. An example of this might be measuring the sap flow in a tree using the heat ratio method. Heat is input via a probe and the upstream and downstream temperature is recorded over time from which sap flow can be calculated. Assuming the SDI-12 digital interface is well designed there should not be data corruption due to noise pick up. The disadvantage to SDI-12 is the more sensors you have on a bus the more difficult it will be to isolate a faulty sensor and restore the sensor network.

2.3 The Raspberry Pi

The Raspberry Pi is a small and inexpensive personal computer developed and manufactured by the Raspberry Pi Foundation which first released the computer to the public in 2012 (Raspberry Pi Foundation 2015). Two Raspberry Pi models have been released, "Model A" and "Model B". The current Model A revision is the Raspberry Pi model A+ which is recommended for embedded projects. Model A is shown in figure 9. The current Model B revision is the Raspberry Pi 2 Model B and it costs $44 AU (element14 2015). Model B is shown in figure 10. The Raspberry Pi Foundation provide Debian Linux ARM distributions for download. The Raspberry Pi has a large range of inputs and outputs available to interact with the environment. This provides the perfect set of factors allowing people to build cheap devices and learn about technology.

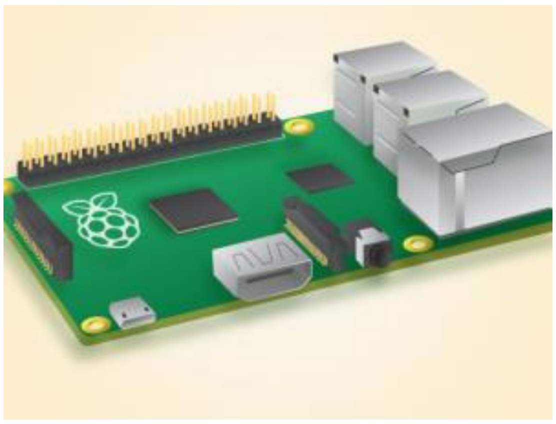

Figure 10 Raspberry Pi 2 Model B (Raspberry Pi Foundation 2015)

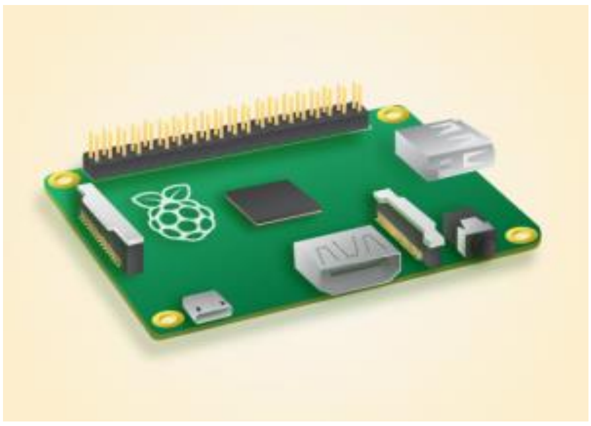

Figure 9 Raspberry Pi Model A+ (Raspberry Pi Foundation 2015)

2.3.1 Raspberry Pi's Operating System

The Raspbian operating system (a Linux distribution) is not a real time system and any thread can be interrupted by the OS. While it is possible to code a bare metal version of logging firmware for the Raspberry Pi that would be strictly real-time it would require a lot of development. The Raspberry Pi kernel has useful features to manage a program (or thread) that can be leveraged to interface with the outside world. The Linux kernel allows an interrupt to be detected on any GPIO input as a rising or falling edge transition and the main program can continue to run while waiting for an interrupt.

2.3.2 Raspberry Pi Hardware Specifications

The Raspberry Pi has most on board devices found on a typical personal computer, including; HDMI port, USB ports, micro SD Card port, and an Ethernet port. It is a small but capable computer. The Model A+ is less suitable for use as a logger as it does not include an Ethernet port and is thus not considered as an alternative. The Specification for the Raspberry Pi 2 Model B are highlighted here.

Broadcom BCM2836 - System on Chip (SoC) (CPU, GPU, DSP, SDRAM, 1 USB port)

The Raspberry Pi is based on the Broadcom BCM2836 SoC, which was designed for multimedia processing applications (Raspberry Pi 2015). The Raspberry Pi 2's CPU is a 900 MHz quad-core ARM Cortec-A7. It has 1 GB SDRAM RAM which it shares with the GPU.

Dimensions

The Raspberry Pi is 85.6 mm x 56.5 mm (about the size of a credit card).

Ethernet Port

The Raspberry Pi's network readiness maybe leveraged to make use of existing internet hardware to get information to a BMS interface. The Raspberry Pi can be connected into a Router or Network switch. A USB Wi-Fi transmitter device can be connected as an alternative way of connecting it to the internet.

Power Supply Requirements

The Raspberry Pi is powered through the micro USB-type B port. It requires a supply of 5 volt. The current drawn by the Raspberry Pi depends on what is connected to it therefore it is recommended that a PSU is current limited. It is recommended that the Raspberry Pi 2 is supplied using a power supply unit with a capacity of 1.8 A (Raspberry Pi 2015). Without any peripherals (bare-board) the active current consumption is up to 500 mA. The maximum total USB peripheral current draw is 600 mA. The maximum current that can be supplied through the 5V GPIO power pin safely is 1.3 Amps (1.8 A – 0.5 A) without any other peripherals connect i.e. monitor, keyboard, mouse etc. A powered USB hub can be connected to the Pi USB hub when required.

General Purpose Input Output (GPIO) Pins

The Raspberry Pi 2 has a 40 pin header allowing connection of the GPIO's of the BCM2836 SoC to digital devices. These pins can be programmed as either inputs or output. The mapping of this header is shown in figure 11. This is a representation of the header as viewed looking above. BCM are the Broadcom pins of which there are 28 in total. The BCM pins are also referred to as GPIO and have the same numbering. The GPIO pins can draw a maximum combined current of 50 mA safely. An individual pin can draw a maximum of 16 mA safely (Raspberry Pi 2015). These pins are not current limited. The GPIO voltage are 3.3 V and there is no over voltage protection on the board. To interface anything will require an external board with buffers, level conversion and analogue circuits. The pins cannot drive a capacitive circuit.

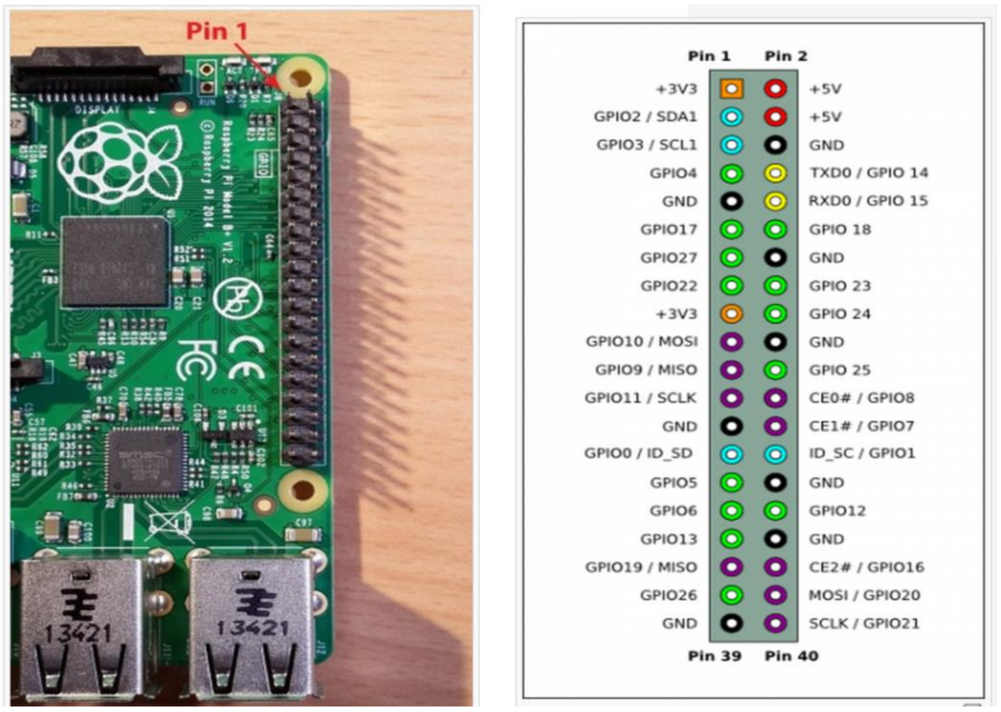

Figure 11 Raspberry Pi pinout (Raspberry Pi 2015)

The GPIO pin voltage thresholds are given in table 4. The GPIO when set as an input can be configured as a Schmitt trigger, with input hysteresis. With hysteresis there are different threshold voltages for rising and falling transition but these are not actually specified by in the Broadcom BCM2835-ARM-Peripherials datasheet (Broadcom 2012).

Table 4: GPIO pin voltage threshold

Output LOW voltage (V_{OL})	< 0.7 V
Output HIGH voltage (V_{OH})	> 2.6 V
Input LOW voltage (V_{IL})	< 0.8 V - > 0 V
Input HIGH voltage (V_{IH})	> 2.0 V - < 3.3 V

RTC

The Raspberry Pi can keep time while powered on only. Networked Raspberry Pi's will update their RTC automatically on start-up (Raspberry Pi 2015). Non-networked units will need to have their RTC updated manually. Updating the RTC via the internet saves some cost to the consumer but is unsuitable for many stand-alone applications. An RTC can be added if the device is running as a standalone system.

Chapter 3: Hardware Implementation

This chapter outlines the design process for the SDI-12 electrical interface. The first section of this chapter looks at two different approaches to the implementing SDI-12 and informs an approach based on the advantages and disadvantages of both. The second section of this chapter investigates hardware options for level shifting the SDI-12 exchanges. The third section presents the schematic and the fourth and final section of the chapter outlines a series of functional test and results.

3.1 Assessment of both the GPIO and UART Approaches to Implementing SDI-12

SDI-12 communications relies on precise signal conditioning resulting in an exchange of ASCII characters in frames with 7 data bits, 1 parity bit, a start bit and a stop bit at 1200 Baud. Two possible approaches to interface the Raspberry Pi to SDI-12 sensors are, 1) the GPIO pins 2) the UART. In both approaches hardware is required to level shift between the Raspberry Pi TTL logic and the SDI-12 voltage levels.

3.1.1 GPIO

Henderson (2015) reports that the Raspberry Pi 2 is capable of toggling a GPIO pin 0 and 1 at a frequency of 9.6 MHz using his wiringPi C/C++ library functions (with no overclocking). By setting a delay period after each transition a GPIO pin is capable of transmitting and receiving serial data at 1200 baud. The major disadvantage of this approach is the possibility of un-reliable data exchanges due to the Raspbian operating system sharing system resources with other threads which may result in intermittent read and write control.

In the event that the OS de-schedules the process while transmitting a command it is highly likely that the data received by the sensor will be invalid and in this case the sensor will not respond. The command will subsequently be re-issued by the logger to the sensor. There are just twenty SDI-12 commands in total which are 3, 4 or 5 ASCII characters in length (including the sensor address). The first character of a command is always the sensor address and the final character is always an exclamation mark (!). Any interruption to an outgoing command will result in a different combination of the 3, 4 or 5 characters. When considering a command that is 3 ASCII characters (excluding the address) such as the measurement commands, there will be three 10 bit frames or a total of 30 bits resulting in 2^{32} possible bit combinations. With only 20 valid bit combinations the likelihood of receiving an incorrect command is small. Additionally, the parity bit and stop bit must also be valid.

The inherent risk in receiving the incorrect sensor response is greater as the response length is variable (up to 35 characters (350 bits)) and only the first 10 bits and the last 20 bits of the response are known. All frames of a sensor response must have a valid start and stop bit. Certain commands including the send measurement command can also request a 3 character CRC code which will provide assurance the data is valid. Only sensors compliant with the latest SDI-12 specification version 1.3 (2013) implement the CRC feature.

The advantage of the GPIO pin approach over the UART approach is that multiple SDI-12 buses are possible, i.e. as many as there are GPIO pins available on the Raspberry Pi.

3.1.2 UART

The Raspberry Pi has one externally accessible Universal Asynchronous Receiver/Transmitter (UART) on pins BCM14 (TXD) and BCM15 (RXD) that is capable of performing SDI-12 exchanges with additional hardware. The UART sends and receives serial data and performs timing and parity checking. The UART transmits and receives one bit at a time at a specific rate using hardware such as bit shift registers that get timing from the system clock. The UART approach is inherently safer than the GPIO approach.

A consideration for implementing the UART approach is that the idle state of the UART is a logic HIGH whereas the idle state of the SDI-12 bus is a logic LOW (Broadcom 2012). This will not be a significant problem as the hardware required to level shift the incoming and outgoing signals can also invert the signals.

3.1.3 Informing the Approach

After extensive investigation into the UART approach, it is dropped in favour of the GPIO approach because some design concepts with the UART approach remain unanswered and successful outcome could not be guaranteed. Confirmation with an approach would be possible by experimenting with the UART and hardware but is considered to be a risk due to time constraints and unknown work load. The unanswered problems to address for the UART approach are:

1) How to send a wake sensors command (12 ms break (logic HIGH) and 8.33 ms marking (logic LOW)). The break is longer than a 10 bit frame (8.33 ms). It would not be possible to send a break using the UART TX unless it is possible to disable the stop bit generation at the end of the frame (logic low).
2) How to level shift between the Raspberry Pi UART TX pin to the SDI-12 voltage levels. Any hardware used to level shift and drive the SDI-12 bus must be capable of being put into a high impedance state when the SDI-12 logger is listening for a response from a sensor on a bi-directional line. As the UART is somewhat automatic it may be a challenge to know exactly when the UART starts and finishes sending a command to the sensor. If the time the transmission finishes is known, the driver can be put into a high impedance state.

3.1.4 Testing the GPIO Suitability

Before proceeding with the GPIO approach experimentation is undertaken to detect the nature of the interruptions. The test aims to approximate the frequency of the interruptions and the length of time the interruptions last. The piHiPri() function from the wiringPi library is tested also. piHiPri() is a function that sets the priority of a process anywhere from low (1) to high (99).

A test program written in C toggles a GPIO pin at three frequencies (5 MHz, 2.5 kHz and 600 Hz) and the pin state is analysed with an oscilloscope which is set to trigger on a positive pulse with width greater than α. The number of re-triggers are recorded in table 5 for a test duration of 2 minutes with increasing values of α starting at 50 µs and where the process is given both low and high priority using piHiPri(). Two test conditions are implemented which vary the demand on the CPU from minimal to high. The minimal demand condition is equivalent to the demand to which the logger is expected to run in the field and it is when only the background (or daemon) programs are running i.e. those of which are part of the Raspbian OS. The high demand condition is when multiple user application are running simultaneously with the test program. The user applications is a web browser that is continually being refreshed and another GPIO application running continually. The interruptions are observed to be predictable under these test conditions and the test duration of two minutes long enough to give accurate results that are repeatable. As the oscilloscope trigger will only happen on positive or negative pulse width that is greater than α, the actual number of interruptions for each test should be interpreted as double the value given in table 5.

Table 5 Frequency of delays to 3 GPIO toggle frequencies for a set of minimum pulse width times.

α (a positive pulse width greater than α will re-trigger the oscilloscope)	**Test conditions – demand placed on the CPU**	**Test 1 - toggle at 9 MHz (no delay)**		**Test 2 - toggle at 2.5 kHz (200µs delay)**		**Test 3 - toggle at 600 Hz (830µs delay)**	
		Low Priority (1)	**High Priority (99)**	**Low Priority (1)**	**High Priority (99)**	**Low Priority (1)**	**High Priority (99)**
> 50 µs	**High Demand**	> 1000	> 500	-	-	-	-
> 100 µs	**High Demand**	> 1000	> 100	-	-	-	-
> 200 µs	**High Demand**	> 1000	3	-	-	-	-
> 500 µs	**High Demand**	> 50	0	> 500	> 100	-	-
> 500 µs	**Minimal Demand**	1	0	> 100	0	-	-
> 1 ms (1.25 bts at 1200 baud)	**High Demand**	> 40	2	> 500	> 35	> 1000	**120**
> 1 ms (1.25 bits at 1200 baud)	**Minimal Demand**	0	0	27	0	80	**0**
> 1.26 ms (1.5 bits at 1200 baud)	**High Demand**	40	0	> 500	0	> 500	**60**
> 1.26 ms (1.5 bits at 1200 baud)	**Minimal Demand**	0	0	20	0	70	**0**
> 1.66 ms (2 bits at 1200 baud)	**High Demand**	30	0	> 500	0	> 500	**20**
> 1.66 ms (2 bits at 1200 baud)	**Minimal Demand**	1	0	50	0	60	**0**
> 2.49 ms (3 bits at 1200 baud)	**High Demand**	24	0	> 350	0	> 500	**0**
> 2.49 ms (= 3 bits at 1200 baud)	**Minimal Demand**	2	0	23	0	30	**0**

The results in table 5 show that that by giving the process a high priority using the piHiPri() function from the wiringPi library the number of interruption can be significantly reduced. The results in test 2 and 3 also show that a slower toggle frequency experiences more frequent interruptions than the case of maximum toggle frequency with similar priority. The results show significant interruptions for test 3 with high priority that last 1.66 ms (equal to double the toggle delay) when there is high demand. This is an indication that the GPIO approach may fail if there is a high demand on the CPU, however if the SDI-12 exchange failed the command would be re-issued. Figure 12 shows a screenshot from the oscilloscope for the test where the GPIO is toggled at 600 Hz. The oscilloscope has captured a positive pulse width of 2.08 ms which was a result of the operating system de-scheduling the process momentarily.

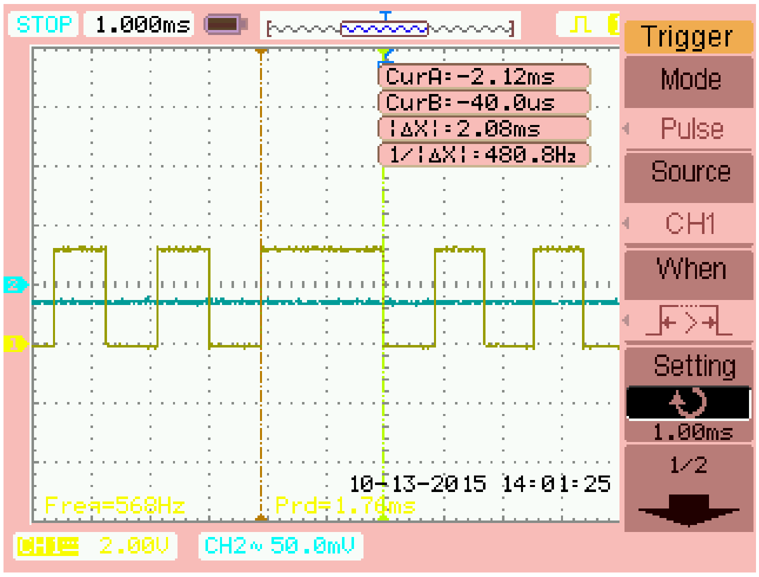

Figure 12 Screen shot of the oscilloscope for a test where GPIO toggled at 600 Hz

3.2 Component Selection and Implementation Considerations

The SDI-12 exchanges occur on bi-directional data line and so any hardware used to level shift and drive the SDI-12 bus (outgoing transmission) must be capable of being put into a high impedance state for when the SDI-12 logger is listening for a response from a sensor. A HIGH on the SDI-12 bus is between 3.5 and 5.5 V and to achieve the maximum number of sensors on the SDI-12 bus the data line should be driven close to 5V (5.5V if possible). The 74XX1T45 and the 74XX240 series chips are assessed for suitability.

3.2.1 74XX1T45 Series Chip

The 74XX1T45 series chips are a single-bit dual supply bus transceiver with configurable voltage translation and 3-state outputs (it is not available in a dual inline package for breadboard prototype). Figure 13 shows a functional block diagram for the SN74LVC1T45. Texas Instruments (2014) states the 'logic levels of the direction-control (DIR) input activate either the B-port outputs or the A-port outputs. The device transmits data from the A bus to the B bus when the B-port outputs are activated and from the B bus to the A bus when the A-port outputs are activated.' The high voltage level at each port is dependent on the supply voltage V_{CCA} or V_{CCB}. V_{CCA} and V_{CCB} accept any voltage from 1.65 V to 5.5 V.

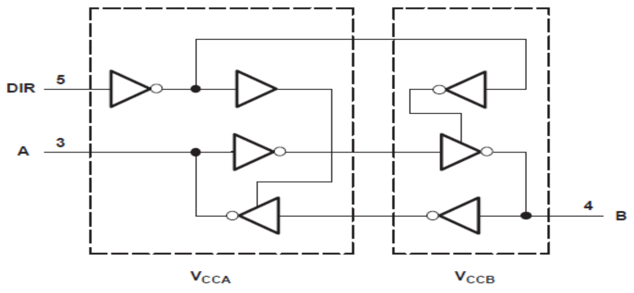

Figure 13 Functional block diagram for the SN74LVC1T45 (Texas Instrument 2014)

Table 6 shows minimum and maximum expected voltage levels for two supply voltages that would allow exchanges between the Raspberry Pi and SDI-12 sensors. In this case port A is connected to the Raspberry Pi and port B is connected to the SDI-12 bus. The voltage levels in table 6 are for when port A supply voltage (V_{CCA}) is between 3 and 3.6 V and port B (V_{CCB}) is between 4.5 and 5.5 V.

Table 6: Minimum and maximum voltage levels for SN74LVC1T45 where, V_{CCA} = 3-.3.6V and V_{CCB} = 4.5 - 5.5 V

	Port A connected to RPi data pin. (RPi logic HIGH voltage is: 3.3 V)		**Port B connected to SDI-12 bus. (Max Logic HIGH SDI-12 voltage is: 5.5V)**	
	Min	**Max**	**Min**	**Max**
V_{IH}	2 V	5.5 V	3.15 V	5.5 V
V_{IL}	0 V	0.8 V	0 V	1.65 V
V_{OH} (I_{OH} = -24 mA)	2.4 V	3.3 V	4.2 V	5.5 V
V_{OL} (I_{OH} = 24 mA)	0 V	0.55 V	0 V	0.55 V

3.2.2 74XX240 Series Chip

The 74XX240 chips are an octal buffer and line driver with 3-state outputs available in a PDIP-20 package. Figure 14 shows a function block diagram. Texas Instruments (2003) states; 'the 74HCT240 devices are organized as two 4-bit buffers/drivers with separate output-enable ($\overline{OE}$) inputs. When $\overline{OE}$ is low, the SN74HCT240 passes inverted data from the A inputs to the Y outputs'. When $\overline{OE}$ is high, the outputs are in the high-impedance state.

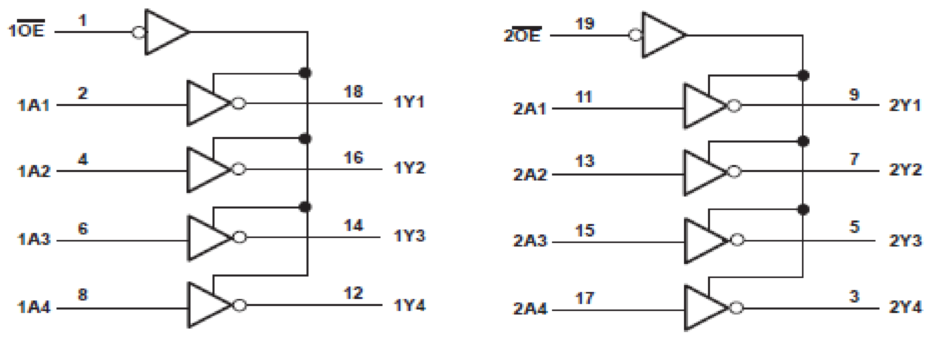

Figure 14 Functional block diagram for a SN74HCT240 (Texas Instruments 2015)

Figure 15 shows a simplified circuit diagram to implement the 74XX240 series chip. Additional components are required if this chip is used.

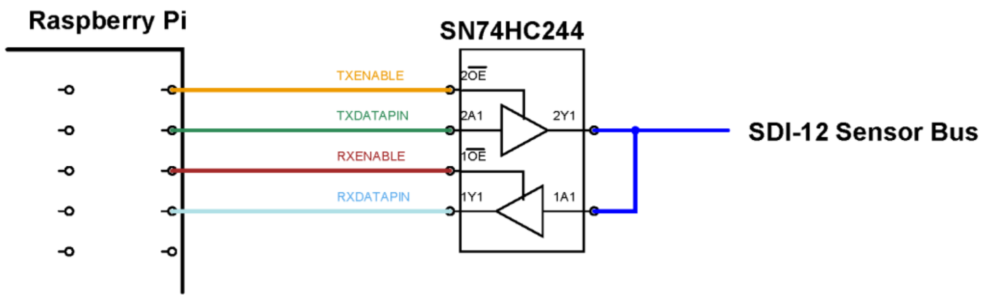

Figure 15 Simplified circuit schematic

The state of four GPIO pins will allow exchanges on the SDI12 bus. Table 7 shows minimum and maximum expected voltage levels for the separate 4 bit buffer/drivers. The SN74HCT240 input are TTL compatible and the output is CMOS compatible. The red entries are incompatible with either the Raspberry Pi or SDI-12 voltage levels. Other 74XX240 series chips are available with both input and outputs that are either CMOS or TTL compatible.

Table 7: Minimum and maximum voltage levels for SN74HCT240 – referenced to figure 15

	1Y1		**2A1**		**1A1**		**2Y1**	
	Min	**Max**	**Min**	**Max**	**Min**	**Max**	**Min**	**Max**
V_{IH}	-	-	2 V	3.3 V	2 V	3.3 V	-	-
V_{IL}	-	-	0 V	0.8 V	0 V	0.8 V	-	-
V_{OH} (I_{OH} =-6 mA)	3.84 V	5 V	-	-	-	-	3.84 V	5 V
V_{OL} (I_{OL} =6 mA)	0 V	0.33 V	-	-	-	-	0 V	0.33 V

3.3 Schematic of Prototype Implementation

As the Texas Instruments SN74HCT240 is available in a PDIP package it chosen for the prototype implementation over the alternatives including the SN74LVC1T45. The schematic given in figure 16 was drawn using XCircuit. The voltage divider consisting of a 66 kΩ and 130 kΩ resistor may not be a suitable solution for lowering the voltage level for the input 1A1 of the SN74HCT240 (TTL compatible input) and will need to be tested. The voltage divider may not be suitable due to the low current through the 68 kΩ resistor making the input to 1A1 more susceptible to noise. The minimum current through the 68 kΩ resistor for a logic HIGH state is:

$$I_{68k(min)} \sim \frac{V_{bus(min)}}{R_1 + R_2} = \frac{3.5}{198\text{k}} = 17\mu\text{A} \qquad (3.1)$$

$I_{68k(min)}$ given in equation 3.1 is small but acceptable. The input voltage level at 1A1 for the maximum logic HIGH state voltage on the SDI-12 bus is given in 3.2. $V_{1A1(max)}$ is just less than the 3.3 V maximum threshold for the SN74HCT240.

$$V_{1A1(\max)} = \frac{V_{bus(max)}R_1}{R_1 + R_2} = \frac{5 \times 130\text{k}}{130\text{k} + 68\text{k}} = 3.28\text{ V} \qquad (3.2)$$

The lowest incoming SDI-12 voltage level that will be recognised as a logic HIGH is:

$$V_{bus(min)} = \frac{V_{1A1(\min)}(R_1 + R_2)}{R_1} = \frac{2 \times 198\text{k}}{130\text{k}} = 3.0\text{ V} \qquad (3.3)$$

Where $V_{1A1(min)}$ is the minimum input voltage for the SN74HCT240 (see: table 7). Any voltage level on the SDI-12 bus above 3 volts (equation 3.3) will be recognised as a logic HIGH. Future designs may not use the SN74HC240 series chip but if they do a comparator with hysteresis will be used to lower the voltage instead of a voltage divider. An implementation using a comparator without hysteresis is shown in figure 17.

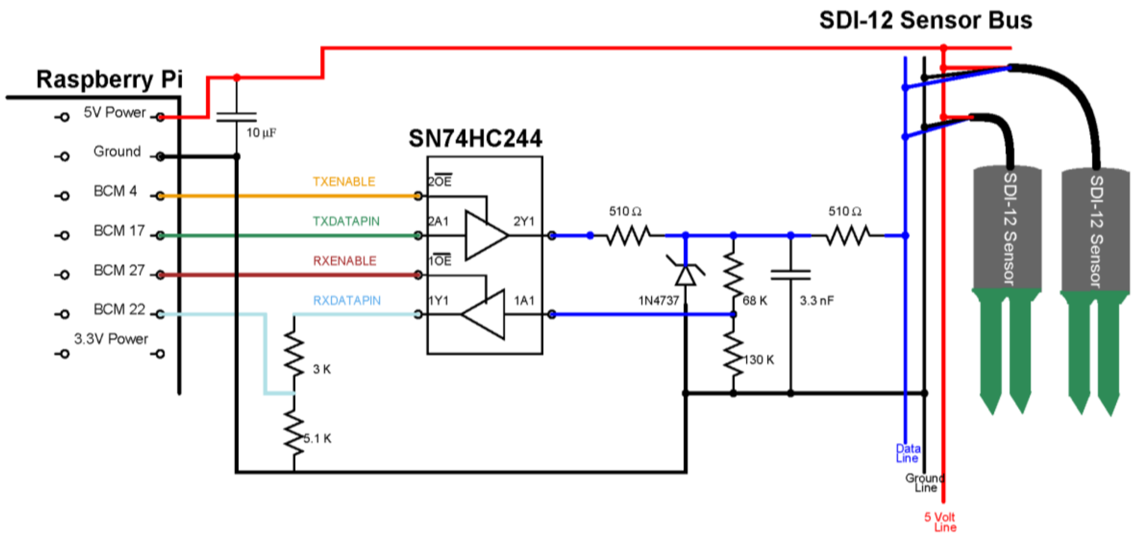

Figure 16 Schematic of prototype implementation

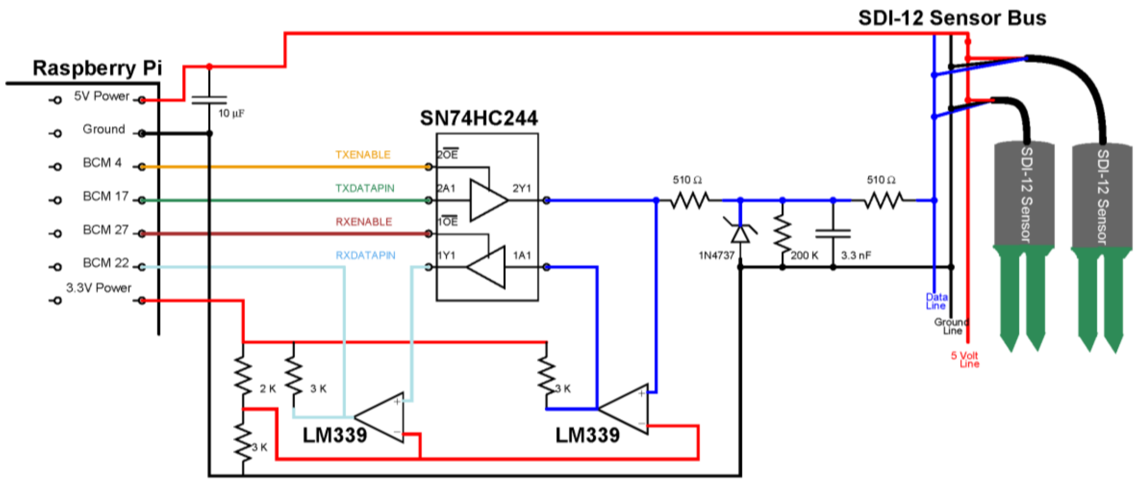

Figure 17 Schematic with comparator

3.4 Implementing and Testing Hardware

The hardware is assembled on a breadboard and tested using an oscilloscope. Figure 18 gives a picture of the prototype implementation.

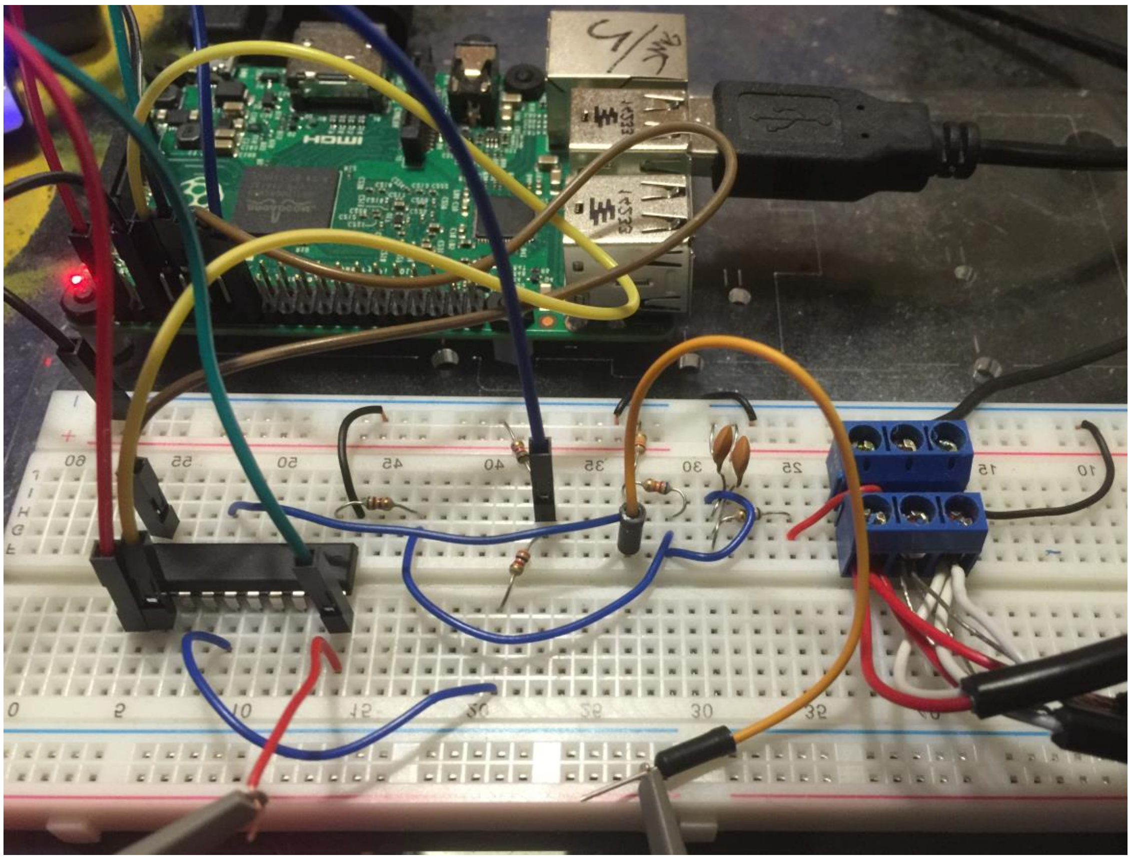

Figure 18 Picture of prototype implementation

A test program written in C toggles BCM 17 (TXDATAPIN - see figure 16) of the assembled prototype shown in figure 18 at 600 Hz (no sensors connected). BCM 22 (RXDATAPIN) is set as an input and BCM 27 (RXENABLE) and BCM 4 (TXENABLE) are setup in output mode with logic LOW. When output enable pin of the SN74HCT240 is presented a logic LOW the output is the inverse of the input, when presented a logic HIGH the output is in a high impedance state. Figure 19 gives a screen shot from the oscilloscope where the yellow waveform is the TXDATAPIN voltage (2A1) and the blue waveform is the RXDATAPIN voltage. Figure 19 shows that logic HIGH of the RXDATAPIN is about 3 volts and so will be read as a logic HIGH by the Raspberry Pi.

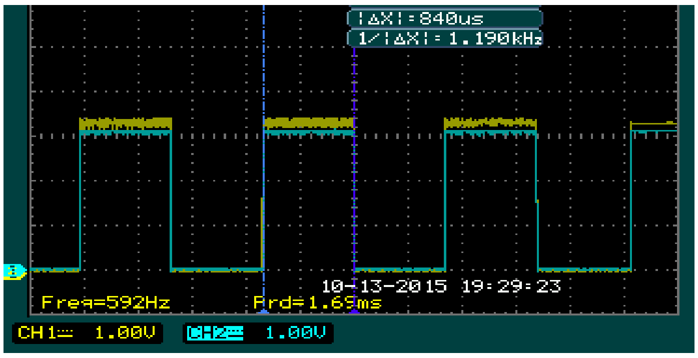

Figure 19 Waveform of TXDATAPIN voltage (yellow) and RXDATAPIN (blue) (see figure 16)

Figure 20 shows two waveforms, the yellow waveform is the TXDATAPIN voltage and the blue waveform is voltage at 2Y1 (see figure 16). Figure 20 shows that logic HIGH of the SDI-12 bus is driven at approximately 5 volts.

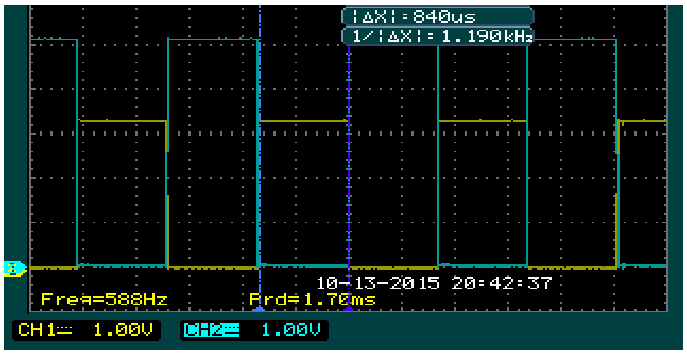

Figure 20 Waveform of TXDATAPIN voltage (yellow) and voltage at 2Y1 (blue) (see figure 16)

Figure 21 shows two waveforms, the yellow waveform is the TXDATAPIN voltage and the blue waveform is voltage at 1A1 (see figure 16). Figure 21 shows that the logic HIGH from the SDI-12 bus (5V) is received as 3.2V and compatible with the SN74HCT240 input voltage thresholds.

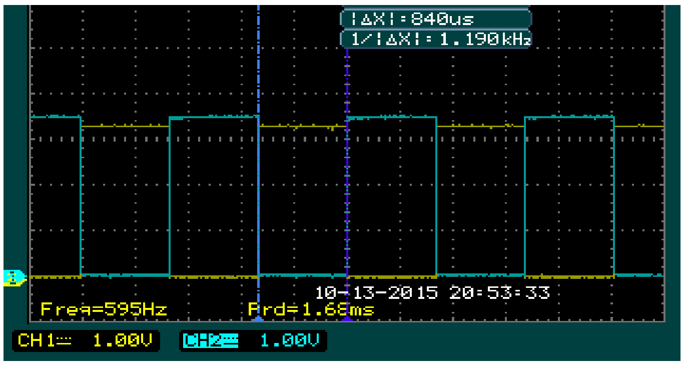

Figure 21 Waveform of TXDATAPIN voltage (yellow) and voltage at 1A1 (blue) (see figure 16)

Chapter 4: Software Development

The software is written in C++ using the Geany development environment for Linux which is installed on the Raspberry Pi. An Arduino SDI12 C++ library written by Kevin Smith has been modified for use with a Raspberry Pi. Before the main SDI-12 logger program is written all the functions of the modified Arduino SDI-12 library are tested extensively to ensure the functions works as intended. The first section of this chapter describes the Arduino SDI-12 library implementation. The second section of this chapter outlines the configuration file format. The third section gives a description of the SDI-12 logger program including the measurement handling and device configuration functions.

4.1 The Arduino SDI-12 Library Implementation

The SDI-12 library was originally authored by Kevin M. Smith for an Arduino based logger. The Arduino SDI-12 library has been modified for use in the Raspberry Pi based logger. The modified C++ code is listed in Appendix C and acknowledges the author as either James Coppock or Kevin Smith. The first section describes modifications to the SDI12 library. The next five sections gives a description of the most important member functions of the SDI12 C++ library.

4.1.1 Description of Modifications to SDI12.cpp

New SDI12 library member functions have been written and some original ones modified. New member functions are listed below with a brief description.

CRCheck() – public function to check the second last character in the buffer is a carriage return <CR> without consuming (without regressing the index to the buffer tail).

LFCheck() – public function to check the last character in the buffer is a line feed <LF> without consuming (without regressing the index to the buffer tail).

overflowStatus() - returns the overflow status.

parityErrorStatus() - returns parity error status. The original code did not check for parity error on the incoming transmission. If parity is incorrect for a received frame the parityError status is set and the interrupt is disabled.

Modifications were done to four member functions. A brief description of the function and modification is given below.

flush() – original function clears the buffers contents by setting the index for both buffer head and tail back to zero. Modification resets the status of the buffer overflow status and parity error status to 'false'.

setState() - the original function defines the state of one data pin and enables or disables the interrupts. Modifications to this function define the state of four pins with additional changes made to the enabling and disabling of interrupts. A global state variable is also added to each state which is set to 'false' in all but the LISTENING state. This variable is checked by the handleInterrupt() function when an interrupt is triggered.

receiveChar() – the original function reads the bit stream from the SDI-12 bus and enters the ASCII character into a buffer without checking for parity error or stop bit state. The original code skipped the stop bit however this allows an additional means of checking a frame to the parity however this increase the number of bits and if the last bit (stop bit) only is invalid all data is invalid. A parity check and stop bit check is added to the original code. If either a parity error or incorrect stop bit is read by the receiveChar function the parity error status is set to true and the state is set to DISABLED (interrupt is disabled). The delay timing is optimised by approximating the overhead in various stages of the code so that the line states is read at the beginning of a bit, allowing more room for error if extra delay is imposed of the process.

writeChar() – original function writes a ASCII character to the bus. A new method for calculating parity bit was implemented.

4.1.2 Description of the setState(state) Function in the SDI12 library for Defining Five Communication States

setState(state) is a private function within the SDI12 class that sets the mode and logic state of four Raspberry Pi pins used in SDI-12 exchanges and also enables or disables interrupts on the RX data pin. Five 'state' parameters taken in a setState(state) function are INTERRUPTENABLE, HOLDING, TRANSMITTING, LISTENING and DISABLE. The original library defined four valid states. The original library used one single digital I/O pin of an Arduino. The Raspberry Pi pins are 3.3 volt and can supply 16 mA from a single pin safely (pins are not current limited). The SDI12 exchanges are implemented using four Raspberry Pi pins connected to the SN74HCT240 inverting tristate buffer line driver as shown in the simplified schematic in figure 22.

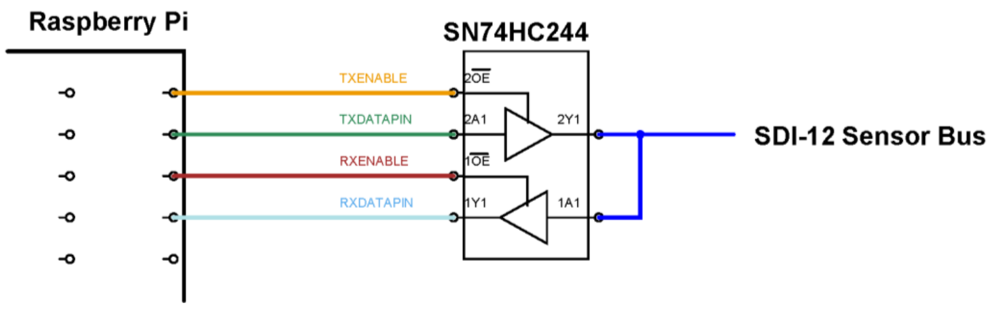

Figure 22 Simplified schematic

The five 'state' parameters in the setState(state) function are described below.

The **INTERRUPTENABLE** state enables the falling edge interrupt on the RX data pin. This is a new state not part of the original library. The original library enabled and disabled the interrupts when changing between the DISABLED, HOLDING, TRANSMITTING, and LISTENING states however this was not possible using the wiringPi library for enabling and disabling interrupts. The wiringPi interrupt function was found to have an unacceptable overhead when going from disabled interrupt in TRANSMITTING state to an enabled interrupt in the LISTENING state. Instead of enabling and disabling interrupt in the various states the interrupt is enabled once when command is sent to begin exchanges and then disabled after all exchanges have finished. In all but LISTENING state the RX data pin is isolated from the SDI-12 bus by driving the $1\overline{OE}$ pin of the SN74HCT240 HIGH so that 1Y1 is in a high impedance state. When the RX data pin is isolated from the SDI-12 bus the interrupt will not be triggered.

The **HOLDING** state is set before sending a command on the SDI-12 bus or after a failed communication attempt due to noise. A HOLDING state holds the line to logic LOW so all sensors are in a quiescent state (ready to receive). The SDI12 standard specifies that SDI12 compatible sensor can go into a quiescent state after 100 ms of line LOW (therefore command should be sent before 100 ms). When in the HOLDING state the RX data pin is isolated from the SDI-12 bus. $2\overline{OE}$ is driven LOW so that the output 2Y1 is the inverted logic at 2A1. The TX data pin is driven HIGH and the SDI12 data line is driven LOW.

In the **TRANSMITTING** state, the RX data pin is isolated from the SDI-12 bus so that the transmission will not trigger an interrupt. $2\overline{OE}$ is driven LOW so that the output 2Y1 is the inverted logic at 2A1. The TX data pin transmits the SDI-12 command 1 character at a time in 10 bit frames with a start bit, 7 data bits (inverted logic LSB first), 1 parity bit and a stop bit.

In the **LISTENING** state a response from a sensor is anticipated. When a sensor responds while in the LISTENING state the start bit will trigger an interrupt which is handled by the interrupt service routine. $2\overline{OE}$ is driven HIGH so that the output 2Y1 is in a high impedance state. $1\overline{OE}$ pin of the SN74HCT240 is driven LOW so that 1Y1 is the inverted input.

The **DISABLED** state isolates the Raspberry Pi from the SDI-12 bus by driving $2\overline{OE}$ and $1\overline{OE}$ HIGH so that 1Y1 and 2Y1 are in a high impedance state. The falling edge interrupt on the RX data pin is disabled.

The mode and logic level of the four Raspberry Pi pins in the 5 states is summarised in table 8.

Table 8 Mode and logic level of Raspberry Pi pins

State	RXDATAPIN Mode: Input	RXENABLE Mode: Output	TXDATPIN Mode: Output	TXENABLE Mode: Output
INTERRUPTENABLE	Input (pullup resistor) Enable falling interrupt	HIGH	HIGH	LOW
HOLDING	Input (pullup resistor) Falling interrupt enabled	HIGH	HIGH	LOW
TRANSMITTING	Input (pullup resistor) Falling interrupt enabled	HIGH	VARYING	LOW
LISTENING	Input (pullup resistor) Falling interrupt enabled	LOW	Don’t Care	HIGH
DISABLED	Input (pullup resistor) Disable falling interrupt	HIGH	Don’t Care	HIGH

In the Raspberry Pi implementation of the SDI12 library the sequence of states for each exchange of information is;

INTERRUPTENABLE → TRANSMITTING → LISTENING → DISABLED

The command is sent in the TRANSMITTING state and a variable length response is received in the LISTENING state. Each response is checked before sending subsequent commands as it is expected that some commands will be resent multiple times.

4.1.3 Waking Up and Talking To Sensors

The sequence of public function calls inherited from the SDI12 library that complete an exchange with an SDI-12 sensor (private SDI12 function calls excluded) are as follows;

begin() → sendCommand(cmd) → end()

Implementation of the Raspberry Pi SDI12 library is as follows; one command is sent to the sensor and after a predefined time period the interrupt is disabled (end() function) and the error status of the response is checked (see section: 4.1.6 Checking for a valid response). If the error status is true the buffer is flushed and command resent otherwise the buffer is read before flushing and proceeding with next command. The variable length time period is dependent on the specific command sent. Where a command sequence must be sent, a delay between commands is determined through an initial command which queries the time delay before subsequent commands in the sequence are sent. The inheritance diagram given in figure 23 shows function calls between the 'sendAndReceive()' function (see section: 4.3.2 Generic functions) and the SDI-12 object that result in an exchange with an SDI-12 sensor. The function names on the arrows are inherited from the object it is pointing to. The sequence of commands shown in figure 23 will result in one command being sent to a single sensor address. The setState function was described in the previous section (see section: 4.1.2 Description of setState function).

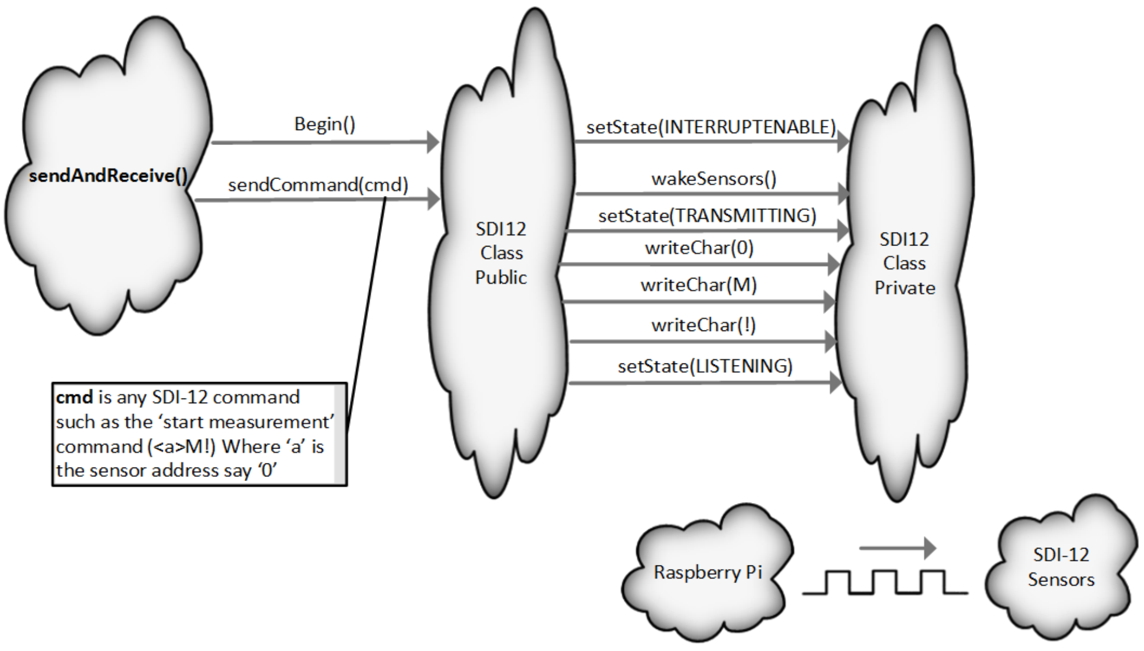

Figure 23 Inheritance diagram for waking up, and sending a command to a SDI-12 sensor

A description of four functions shown in figure 23 that result in the transmission of a command is given.

begin() is a public function within the SDI12 class that begins the functionality of the SDI-12 object. It sets the state of pins to the INTERRUPTENABLE state which enable the falling edge interrupt on the RX data pin.

sendCommand(string cmd) is a public function within the SDI12 class that sends out the character of the String cmd one by one to the data line.

wakeSensors() is a private function that is called by sendCommand(). The state is set to the TRANSMITTING state. wakeSensors() will wake the sensors on the SDI-12 bus by placing spacing (HIGH voltage level) for a minimum of 12 milliseconds (no upper limit specified in standard). This is followed by a marking (logic LOW) for at least 8.33 ms with an upper limit of about 90 ms. Allowing extra time on the minimum, the break is held for 14 ms and the marking for 10 ms. Figure 24 gives a flow chart for both the public sendCommand function and the private wakeSensors function.

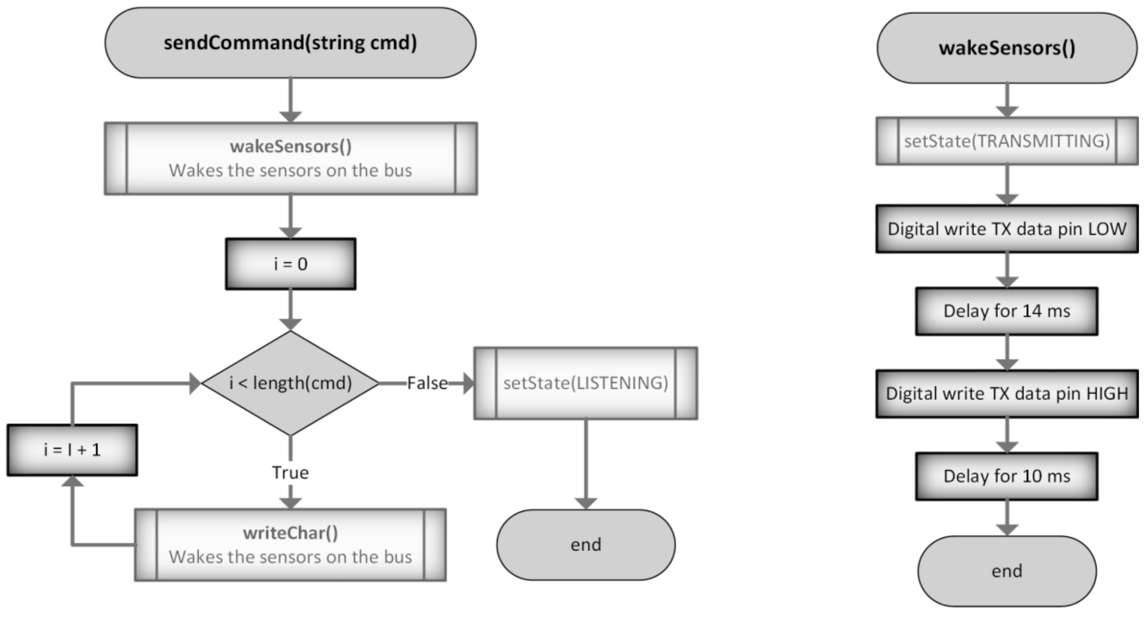

Figure 24 Flowchart for sendCommand() and wakeSensors()

writeChar(uint8_t out) is a private function that is called by the sendCommand function to write characters one at a time to the data line. There are four steps to the transmission.

1 –Calculate the parity bit. The original code used a function from the parity.h library to calculate the parity. I have used an alternate algorithm that calculates the number of 1's in the 7 data bits. The parity bit with the data bits should have an even number of ones for even parity. The algorithm returns the even parity bit as either 0 (even number of 1's) or 1 (odd number of 1's). The parity algorithm merges the first 4 bits with the last 4 bits of the byte containing a 7 bit ASCII code using an XOR operation. As shown below the parity of two bits is computed with an XOR operation.

(0 XOR 0) -> 0
(0 XOR 1) -> 1
(1 XOR 0) -> 1
 1 XOR 1) -> 0

Now with four bits there are 16 possible values for the even parity bit. The hexadecimal number 6996 (0110 1001 1001 0110 binary) represents the correct even parity for the 16 possible 4 bit combinations. Shifting 0x6996 to the right by the number represented by the four bits leaves the relevant parity bit in bit position 0. A zero in bit position 0 means there is an even number of ones and a 1 means an odd number of ones in the ASCII code.

2 – Send the start bit and delay 820 µs. The start bit is a 1 on the SDI12 data line. Writing a LOW to the TX data pin will cause the SN74HCT240 to output a HIGH. A LOW is written to the TX data pin for 820 µs.

3 –Send the payload (7 data bits of the ASCII character plus the parity bit) least significant bit first and inverse logic. This is accomplished using a bitwise AND operations on a moving mask (00000001) --> (00000010) --> (00000100)... and so on. An if statement determines whether a 1 or 0 should be sent.

```
if(out & mask){
  digitalWrite(_txDataPin, HIGH);
}
else{
  digitalWrite(_txDataPin, LOW);  }
```

4 – Send the stop bit ('0') by writing the TXDATAPIN HIGH for 820 µs.

Figure 25 gives a flow chart for the writeChar function.

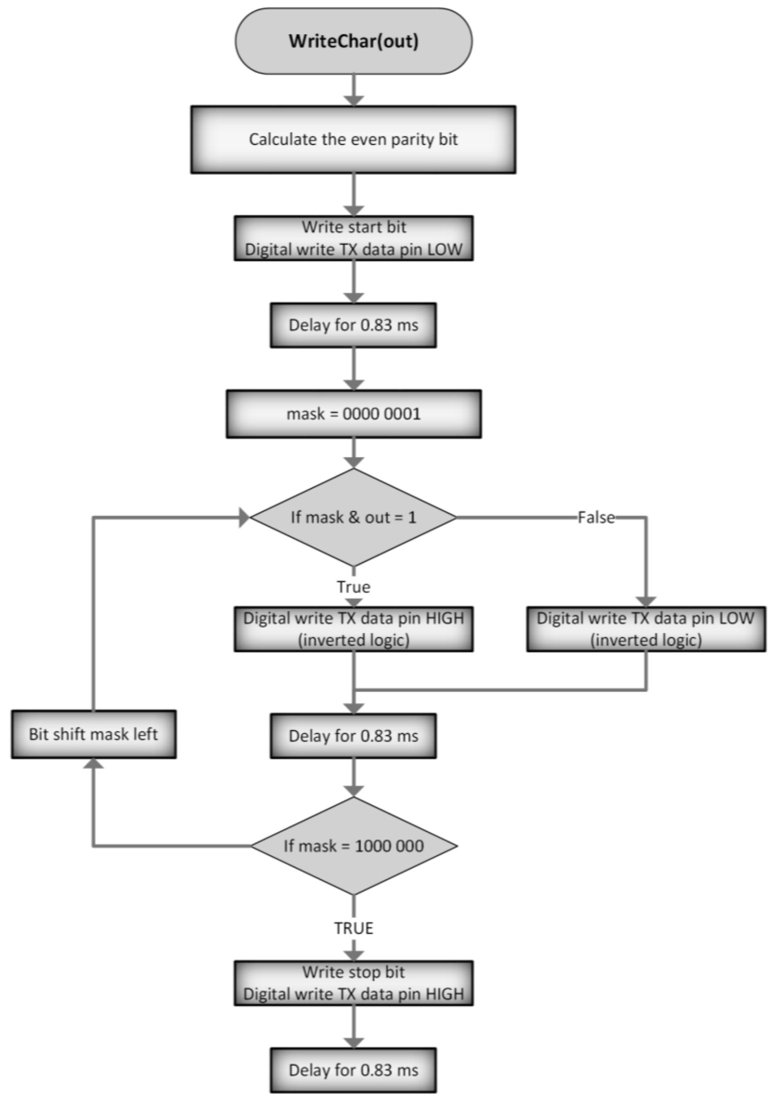
WriteChar(out)
Calculate the even parity bit
Write start bit
Digital write TX data pin LOW
Delay for 0.83 ms
mask = 0000 0001
If mask & out = 1
False
True
Digital write TX data pin HIGH (inverted logic)
Digital write TX data pin LOW (inverted logic)
Bit shift mask left
Delay for 0.83 ms
If mask = 1000 000
TRUE
Write stop bit
Digital write TX data pin HIGH
Delay for 0.83 ms

Figure 25 Flowchart representation of the writeChar function

4.1.4 Interrupt Service Routine to Read Data into the Buffer

This section gives an overview of the Raspberry Pi interrupts and functions from the SDI-12 library that reads asynchronous data in 10 bit frames from the data line, checks the parity and enters valid data into a buffer.

Overview of Raspberry Pi Interrupts

The Raspberry Pi kernel provides a rising or falling edge interrupt on the GPIO. The interrupt service runs as a thread separate to the main program. Two wiringPi C/C++ library interrupt functions are assessed. The difference in the two interrupt functions is, one stalls and waits for a rising, falling or both rising and falling edge to trigger the interrupt and the other allows the main program to continue until the edge transition triggers an interrupt service routine (ISR) which is any function.

For both functions the pin interrupt must be enabled or disabled. The pin interrupt is enabled either through the GPIO Utility or alternatively with the wiringPiISR() function described below. An application named the 'GPIO Utility' created by Gordon Henderson (Henderson 2015) is used in testing the wiringPi C/C++ functions from the terminal command prompt. The wiringPi C/C++ library of functions provide a simple approach to controlling of the GPIO pins such as reading the digital voltage on a pin setup as an input or changing the mode of a GPIO pin to input or output. With the GPIO Utility installed the command entered into the Linux terminal command prompt to enable or disable the interrupt on a GPIO pin (export pin as an interrupt) is:

```
gpio edge <pin> rising/falling/both/none
```

The edge detection can be set for a rising edge, falling edge, or both. A program can send a command to the terminal command prompt through a 'system call'. A system calls is a function in the C library stdlib.h. A system call example which will disable the interrupt through the terminal command prompt is;

```
system ("gpio edge 17 none");
```

The two wiringPi C/C++ library interrupt functions are described below.

1) **waitForInterrupt(int pin, int timeOut)** - when called, the program will stall and wait for the interrupt on the GPIO pin taken as one of the parameters. The timeOut parameter gives a time in milliseconds before the program resumes if no interrupt occurs. An integer value -1 for the timeOut parameter will cause the program to wait for ever. The pin must be initialised for interrupts before calling the waitForInterrupt function. The pin can be initialised using the system call method. If the pin interrupt is not initialised (exported) the program will exit when waitForInterrupt() function is called with error. If the pin interrupt is initialised but disabled (edge = 'none') the program will stall when the waitForInterrupt function is called and will only continue after a specified timeout period (parameter timeOut).

2) **wiringPiISR(int pin, int edgeType, void(*function)(void))** - registers a function to receive interrupts. wiringPiISR() exports the pin as an input with either a rising, falling or both by specifying edgeType parameter as either INT_EDGE_FALLING, INT_EDGE_RISING, INT_EDGE_BOTH or INT_EDGE_SETUP. There is no disabling of the interrupt routine with this function. To disable the ISR function on the exported pin a system call is used i.e. system ("gpio edge pin none").

   When an interrupt is triggered, it is cleared in the register before calling the function and so when a subsequent interrupt fires it will be captured. The wiringPiISR() function cannot be called on any one pin more than once; doing so will result in the ISR function being called multiple times when an interrupt is triggered. When wiringPiISR is called, the pin will be set up as an input and the pullup and pulldown resistor state will remain. There is a bug in the wiringPiISR function where the first interrupt after calling wiringPiISR (exporting a pin) will cause the *function to be called twice.

wiringPiISR is the better choice and allows more flexibility in programming but has to be enabled and disabled using the system call approach.

Enabling and disabling the interrupt

Testing of the system call method of enabling and disabling an interrupt find the system call takes about 10 ms. The overhead is unacceptably long so the interrupt is enabled once at the beginning of a send command and disabled after receiving the response as opposed to disabling when transmitting and enabling when listening. This is possible because the pin of the SN74HCT240 which is connected to the RXDATAPIN can be put into a high impedance state isolating it from the bus when not in LISTENING state.

receiveChar()

The inheritance diagram given in figure 26 shows the sequence of function calls after a start bit triggers an interrupt. The interrupt service routine (handleInterrupt) passes responsibility for the interrupt to the receiveChar function which reads data into the buffer. The function names on the arrows belongs to the object it is pointing to. In this section a description of the receiveChar function is given.

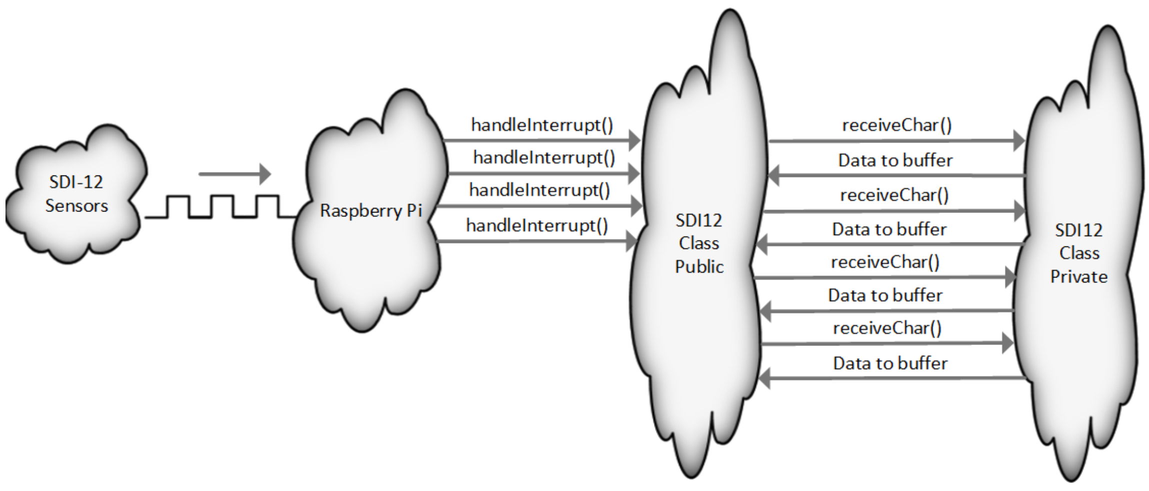

Figure 26 Inheritance diagram for reading data into the buffer

handleInterrupt() is the function registered by the wiringPiISR function as the interrupt service routine for RXDATAPIN. handleInterrupt() is a static member function of the SDI-12 class library. The function is initialised in the main program using the wiringPi function wiringPiISR(). handleInterrupt() will pass of responsibility for interrupt to the receiveChar function. handleInterrupt is declared as the interrupt service routine vector with the following function call:

wiringPiISR (RXDATAPIN, INT_EDGE_FALLING, SDI12::handleInterrupt);

receiveChar() is a private function that is called by the handleInterrupt function. receiveChar() reads asynchronous data in 10 bit frames into the buffer. This takes place over a series of steps outlined below.

1. Check the start bit is a logic LOW. There may have been a false trigger of the interrupt
2. Declare a variable for the incoming char as an unsigned integer of 8 bit (uint8_t)
3. Delay the a small delay period (800 µs)
4. Read the 7 data bits plus the parity bit
5. Check the stop – if stop bit LOW set _parityError = true and disable interrupt
6. Check for parity error – if parity error set _parityError = true and disable interrupt
7. Enter 7 bit ASCII character into buffer

The original receiveChar function did not include a parity error check or stop bit check. A wiringPi C/C++ library function is used to read whether or not the line state is HIGH or LOW.

The function is; int digitalRead(int pin). Figure 27 gives a flow chart for the receiveChar function.

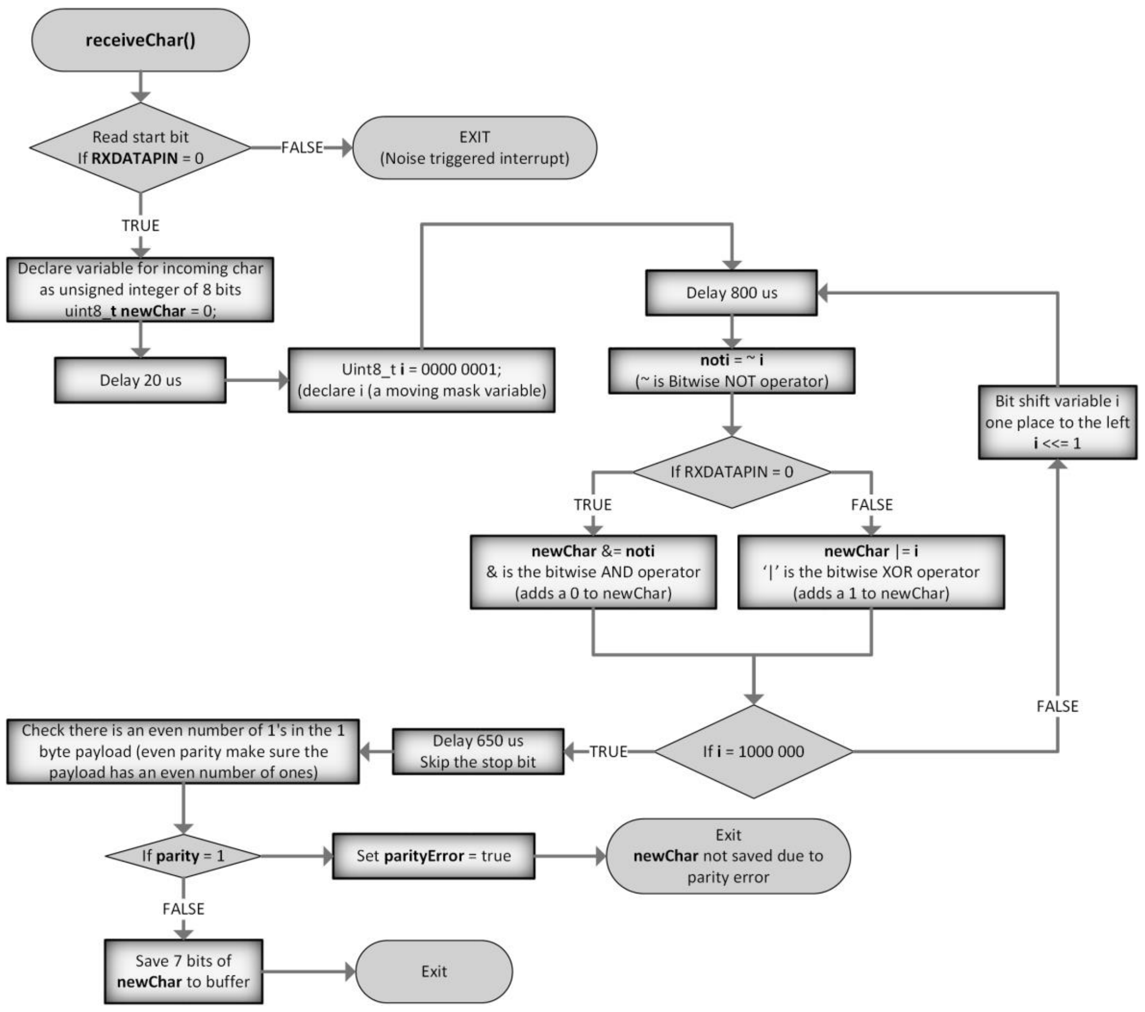
receiveChar()
Read start bit If RXDATAPIN = 0
FALSE
EXIT (Noise triggered interrupt)
TRUE
Declare variable for incoming char as unsigned integer of 8 bits uint8_t newChar = 0;
Delay 20 us
Uint8_t i = 0000 0001; (declare i (a moving mask variable)
Delay 800 us
noti = ~ i (~ is Bitwise NOT operator)
Bit shift variable i one place to the left i <<= 1
If RXDATAPIN = 0
TRUE
FALSE
newChar &= noti & is the bitwise AND operator (adds a 0 to newChar)
newChar |= i '|' is the bitwise XOR operator (adds a 1 to newChar)
FALSE
Check there is an even number of 1's in the 1 byte payload (even parity make sure the payload has an even number of ones)
Delay 650 us Skip the stop bit
TRUE
If i = 1000 000
If parity = 1
Set parityError = true
Exit newChar not saved due to parity error
FALSE
Save 7 bits of newChar to buffer
Exit

Figure 27 Flowchart representation of the receiveChar function

4.1.5 Checking for a Valid Response and Reading from the Buffer

Figure 28 below shows public functions inherited from the SDI12 class that are used to check the validity of a sensor response read into the and read contents out of the buffer.

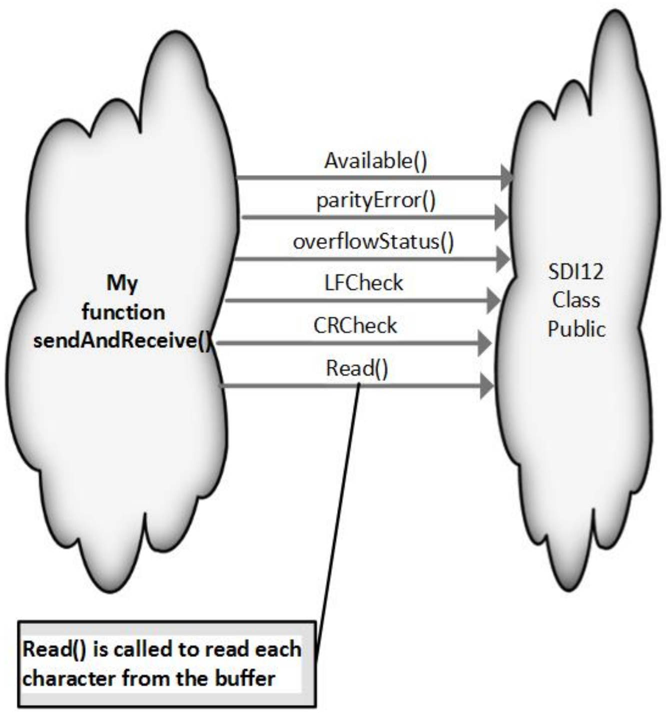
Available()
parityError()
overflowStatus()
LFCheck
CRCheck
Read()
My function sendAndReceive()
SDI12 Class Public
Read() is called to read each character from the buffer

Figure 28 SDI12 functions for checking a valid response and reading from the buffer

Available() reveals the number of characters in the buffer and can be used to make a decision about the data. For some commands the exact number of characters in the response is known while others are variable length responses. Variable length response will have a minimum response length. If the number of characters is less than that expected the buffer is flushed and command resent otherwise further checks made or data read from buffer one character at a time using **read().**

parityError() and **overflowStatus()** reveals the status of the _parityError and _bufferOverflow. If the error status is true the buffer is flushed and command resent otherwise further checks made or data read from buffer one character at a time using **read().**

LFCheck() and **CRCheck()** check the last two character in the buffer are a carriage return and line feed. If either the line feed or carriage return is not available the buffer is flushed and command resent otherwise further checks made or data read from buffer one character at a time using **read().**

4.2 Configuration File

4.2.1 Conceptual Design

In order for the logger to perform its main function (to retrieve one or more parameter values from each sensor connected to the SDI-12 bus and store values to a data file) without having to re-enter the logging session parameters at the start of a session, key parameters are stored to a configuration file “loggerconfiguration.txt” which can either be read or modified manually or through the configuration interface. Each parameter returned by a multi-parameter sensor is assigned a channel number. Channel numbers are sequential integers starting with channel one. The configuration file is modified when a new sensor is added or removed from the SDI-12 bus, the mapping of channel numbers to configured sensor parameters is changed, and the measurement interval is changed. The configuration file is read from before initiating a measurement sequence to determine configured sensor addresses and which channel number the parameter values belongs to. The configuration file also holds the channel names which are appended to the data file at the start of a logging session.

To modify a specific line of the configuration file with software the contents of the file before the line to be modified will be re-written to a new file then the modified line(s) appended to the bottom of the new file, after which any remaining lines from the old data file that are needed should be appended to the bottom of the new file. The information in the configuration file will be organised into separate entries with markers for each section. When line(s) of text within a section is modified the whole section will be re-written.

The first key consideration in the configuration file format development is, how a parameter from a specific sensor address is mapped with a logging channel so that it can be used. Ultimately, any parameter from any specific sensor address can be assigned with any channel number from one to the total number of configured channels. Further consideration to the format is given to how the sensors are to be added and configured. As SDI-12 devices do not return extensive information about themselves in response to the identification command a user will have to manually configure the SDI-12 channel by giving it a name, unit, and specifying which result from that sensor corresponds to it. A user may add a sensor to the logger by carefully modifying the configuration file manually. Alternatively a sensor may be added using the HMI. Two approaches to adding a sensor through the HMI are considered and outlined below. In both approaches it is assumed the address of the sensor is known and unique. If the address is not known it could be found using the “address query” command before configuring it (only one sensor to be connect to the bus when issuing the “address query” command). If the address is not unique it can be changed using the “change address” command.

HMI approach 1 to adding an SDI-12 sensor. In this approach the SDI-12 sensor does not need to be connected to the logger. A HMI would be developed that asks the user to enter the address of the SDI-12 sensor to be configured and the name and unit of each parameter in the order that the parameters are returned. The information needed would be available from the sensor datasheet.

HMI approach 2 to adding an SDI-12 sensor. In this approach the SDI-12 sensor not yet configured is connected to the SDI-12 port. A database of information about SDI-12 sensors is kept locally on the Raspberry Pi which can easily be updated by a user for any new sensor. The database holds details about a sensor such as the name and unit for each parameter in the order that the parameters are returned by the sensor. The software will send a “send identification” command to the SDI-12 sensor not yet configured. The “send identification” command returns the sensor model number, vendor code, and the SDI-12 specification version number that it is compatible with. The software parses the database of SDI-12 sensors file for the sensor model and vendor ID and automatically configures the sensor by assigning each parameter a channel number.

Approach 2 is preferred as it is the easiest way to quickly add a new sensor from the user’s perspective. Approach 2 is fully specified in the next section (section 4.2.2). When a sensor is added the channel numbers are assigned automatically where the first parameter returned by the sensor being added will take the next sequential channel number (next sequential channel number = the total number of configured channels + 1), if a second parameter is returned by the sensor it takes the next sequential number and so on. Therefore the channel numbers are assigned to a particular sensor parameter by the order that the sensor is added and by the order the result is returned. All parameters returned by a sensor address must be assigned a logger channel. Future implementation will allow reordering the channels and removing any channels from the configured list. To change the order of the channels the configuration file would need to be deleted and sensors added in the correct order.

4.2.2 Specification for the Database of SDI-12 Sensors

The SDI-12 sensor database holds key information on the sensor as specified below and is considered simple enough for a user to update with new sensors. A send identification command (<a>I!) can be sent to the sensor address which will return:

• up to an 8 character company name	i.e. DECAGON
• up to a 6 character sensor model number	i.e. GS3

The database will be parsed for the sensor model key with a value field which holds the number of parameters returned by the sensor. A unique key can be formed for each parameter with value that specifies the name and unit of the parameter.

```
; *********************** SDI-12 sensor database ***************************
; Key values that can be extracted from the database includes;
;      1) Number of parameter returned by the sensor
;      2) The name and unit of the parameters returned in correct order as
;         returned by the sensor
;
; *************** Description of the database structure *********************
;
; For each SDI-12 sensor added to the database there will be one line with
; structure defined by SensorInfEntry 1 and at least one line with structure
; defined by SensorInfEntry 2 below.
;
; SensorInfEntry 1 - consists of a single line for each new sensor
; SensorInfEntry 1:        <x1> = <y1>
; 'x1' is the sensor model, it is up to 6 ASCII character, and is case
; sensitive e.g. 'GS3'
; 'y1' is the number of parameters returned by the sensor.
; KEY 'x1' is checked by addChannels() function.
; VALUE 'y1' is read by addChannels() function.
;
; SensorInfEntry 2 - consists of a single line entry for each parameter
; returned by the sensor.
; SensorInfEntry 2:        <x2>_N<z2> = <y2> (<w2>)
; 'x2' is the sensor 'model'.
; 'z2' is the order of parameter returned from sensor where 1 = first
; parameter returned.
; 'y2' is the name of the parameter returned.
; 'w2' is the units of the parameter returned.
; VALUES is read by addChannels() function.
;
; The database currently has a GS3 and 5TM sensors entered as an example of
; the correct entry structure. The GS3 is a Decagon Devices sensor for
; measuring soil moisture, temperature, and electrical conductivity.
; The 5TM is Decagon Devices sensor for measuring soil moisture & temperature.
; ***************************** Database Entries ***************************

GS3 = 3
GS3_N1 = Dielectric (e)
GS3_N2 = Temperature (deg.C)
GS3_N3 = Electrical Conductivity (uS/cm)

5TM = 2
5TM_N1 = Dielectric (e)
5TM_N2 = Soil Temperature (deg.C)
```

4.2.3 Specification for the Configuration File

The information in the configuration file is organised into sections with entries of a specific category identified with markers at the start and end of the section so that the all entries within the section maybe modified when changes are made to the channel configuration. There are 8 configuration file sections with entries that allow adding SDI-12 sensors and logging. The configuration file as presented below is configured with 3 SDI-12 sensors. The three SDI-12 sensors that have been configured to the logger are 2x GS3 and 1x 5TM. Certain entries exist for purpose of defining unique keys in other sections of the configuration file.

```
; ************************* Configuration File *******************************

; This is the logger config file that is written to when a new sensor is added
; or removed and read from before initiating a measurement. There are 8 unique
; entries in the config file that specify the logger configuration.
; An entry of the form:          E<x>=:
; is a ConfigFileEntry marker that marks the beginning of a new entry type.
; An entry of the form:          E<x>=::
; is a ConfigFileEntry marker that marks the end of a new entry type.
; When a sensor is added or removed from the config file using the logger
; configuration menu information will be added, modified or removed within the
; section marker (entry category). 'x' is an integer from 1 to 8 that
; specifies the entry type. Note: the ';' character represents the start of a
; comment. It is placed at the beginning of a line when no key = value
; structure is in place i.e. a line with only a comment. The general structure
; is
;                   key = value      ; comment
; Each entry is described in detail below.


; ConfigFileEntry 1 - is a single line entry.
; ConfigFileEntry 1:             ChanConfigChange = <y1>
; 'y1' is equal to yes or no. yes means that a change has occurred in the
; logger configuration since the last logging session. This will mean
; the channel names will be appended to the bottomof the .csv file and the
; value changed from 'yes' to 'no'.
; KEY 'ChanConfigChang' is checked in measurementHandler()
; VALUE 'y1' is checked in measurementHandler() if 'yes' new channel names are
; written at the bottom of the data file and the value 'y1' is changed to 'no'
; VALUE 'y1' is assigned 'yes' in addChannels() after changing channel config.
E1=:                                              ; Start section 1
ChanConfigChange=no
E1=::                                             ;   End section 1


; ConfigFileEntry 2 - is a single line entry.
; ConfigFileEntry 2:             ConfiguredAddresses = <y2>
; 'y2' is an integer representing the number of sensors with unique addresses
; that have been configured.
; VALUE 'y2' is checked in takeMeasurement()
E2=:                                              ; Start section 2
ConfiguredAddresses=3
E2=::                                             ;   End section 2
```

```
; ConfigFileEntry 3 - is a single line entry.
; ConfigFileEntry 3:              MeasurementInt = <y3>
; 'y3' is an integer value equal to the measurement interval (in minutes).
; Valid 'y3' values are 2, 5, 10 or 20.
; VALUE 'y3' is checked in measurementDelay()
E3=:                                           ; Start section 3
MeasurementInt=2
E3=::                                          ;   End section 3


; ConfigFileEntry 4 - is a single line entry.
; ConfigFileEntry 4:              NoOfConfigChannels = <y4>
; 'y4' is an integer value that is equal to the number of configured channels.
; VALUE 'y4' is checked in addChannels() function
; VALUE 'y4' is checked in measurementHandler() function
E4=:                                           ; Start section 4
NoOfConfigChannels=8
E4=::                                          ;   End section 4


; ConfigFileEntry 5 - exists for each new sensor that is added to the logger.
; ConfigFileEntry 5:              a<x5> = <y5>
; 'x5' is an integer starting at 1 (assigned to the first sensor added
; to the configuration list) up to the value in key "ConfiguredAddresses".
; 'y5' is the address of the first sensor added. The next entry will be for
; the second sensor added and so on.
; VALUES 'y5' are checked in takeMeasurement()
E5=:                                           ; Start section 5
a1 = 6
a2 = A
a3 = 8
E5=::                                          ;   End section 5


; ConfigFileEntry 6 - exists for each new sensor that is added to the logger.
; ConfigFileEntry 6:              add<x6> = <y6>
; 'x6' is the sensor address "0 - 9", "a to 'z' or 'A to 'Z'.
; 'y6' is the number of parameters returned from that address.
; KEY 'x6' is checked by checkAddress() function.
E6=:                                           ; Start section 6
add6 = 3
addA = 2
add8 = 3
E6=::                                          ;   End section 6


; ConfigFileEntry 7 - exists for each parameter returned for each address.
; ConfigFileEntry:                add<x7>P<z7> = <y7>
; 'x7' is the sensor address "0 - 9", "a to 'z' or 'A to 'Z'.
; 'z7' is the parameter return order (first result returned = 1, second = 2,
and so on)
; 'y7' is the assigned channel for result (first result returned = next
available channel)
E7=:                                           ; Start section 7
add6P1 = 1
add6P2 = 2
add6P3 = 3
addAP1 = 4
addAP2 = 5
add8P1 = 6
add8P2 = 7
add8P3 = 8
E7=::                                          ;   End section 7
```

```
; ConfigFileEntry 8 - exists for each channel number. The unique key holds a
; value with sensor address, channel name and unit. This entry specifies a
; channel heading in the datafile. Note the address is stored in front of the
; name and unit in to identify so that datafile results belong to which can be
; linked to a specific sensor.
; ConfigFileEntry:                CH<x8>n = <z8>_<y8> (<w8>)
; 'x8' is the channel number
; 'z8' is the address of the sensor.
; 'y8' is the name of the channel parameter
; 'w8' is the unit of the channel parameter
; VALUE is read by dataFileHeading().
E8=:                                              ; Start section 8
CH1n = 6_Dielectric (e)
CH2n = 6_Temperature (Deg.C)
CH3n = 6_Electrical Conductivity (uS/cm)
CH4n = A_Dielectric (e)
CH5n = A_Soil Temperature (Deg.C)
CH6n = 8_Dielectric (e)
CH7n = 8_Temperature (Deg.C)
CH8n = 8_Electrical Conductivity (uS/cm)
E8=::                                             ;  End section 8
```

4.3 SDI-12 Logger Program

The Raspberry Pi logger software consists of the main C++ source code file 'SDI12Logger.cpp' and three non-standard C++ libraries. The three required C++ libraries are;

- SDI12.cpp (see section 4.1, authored by Kevin Smith available at https://github.com/StroudCenter/Arduino-SDI-12),
- Parser.cpp (authored by Sarmanu available at http://www.dreamincode.net/forums/topic/183191-create-a-simple-configuration-file-parser/),
- wiringPi.h (authored by Gordon Henderson available at https://projects.drogon.net/raspberry-pi/wiringpi/download-and-install/).

After building knowledge and understanding of the SDI-12 library the program is developed from the bottom up. This section gives a description of the SDI12 Logger program. The SDI12Logger.cpp program listing is given in appendix D.

4.3.1 Broad Overview of Functions Called through the HMI

The SDI12 Logger program has functions that perform measurement handling and a partially complete set of functions that perform SDI-12 device configuration tasks. Figure 29 gives a flowchart that shows a sequence of functions called and key tasks performed. The function names are given in green text on the connectors preceding the description of tasks. The blue text on the connectors between blocks shows that the function call is in response to a input.

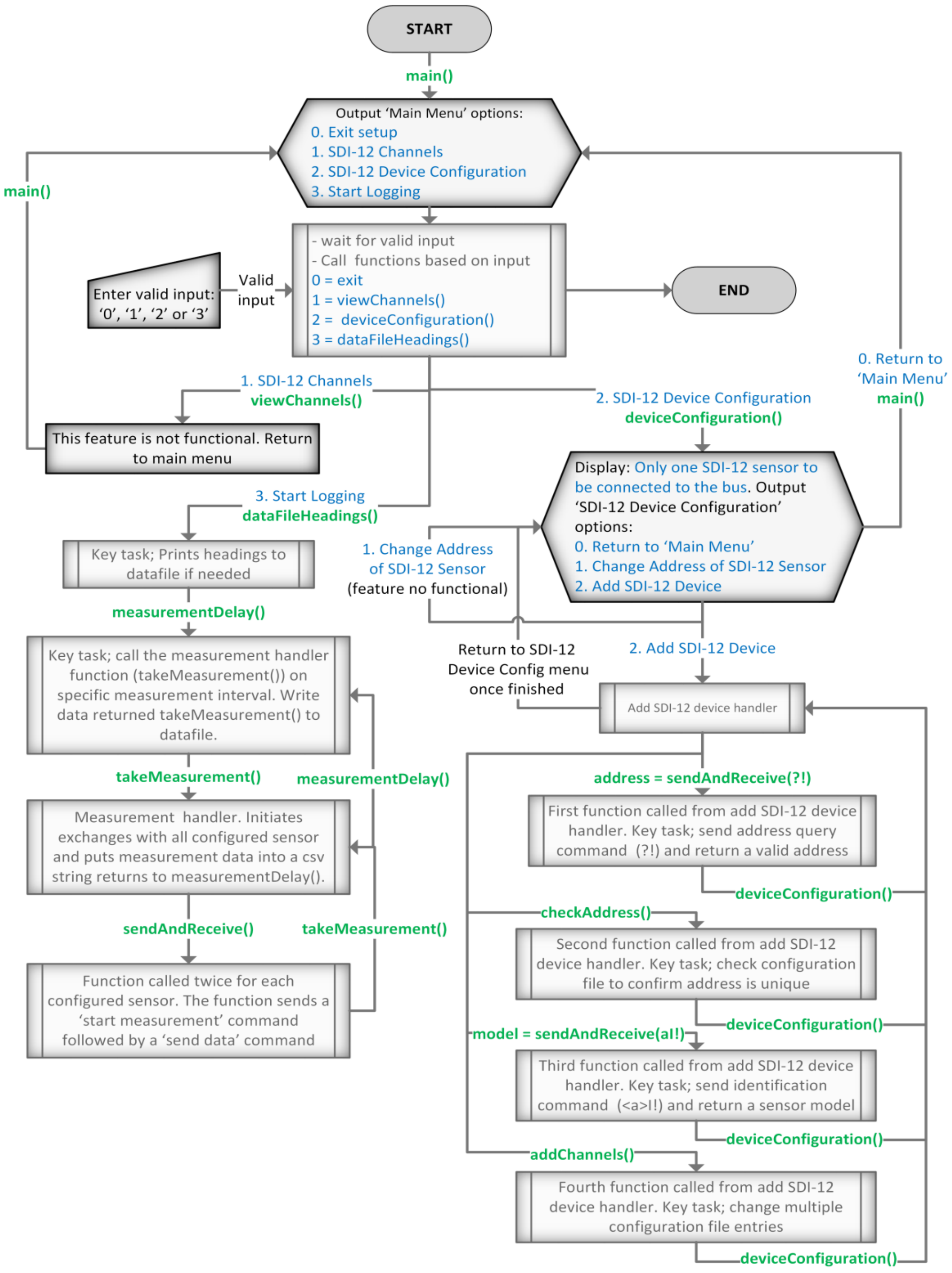
START
main()
Output 'Main Menu' options:
0. Exit setup
1. SDI-12 Channels
2. SDI-12 Device Configuration
3. Start Logging
main()
- wait for valid input
- Call functions based on input
0 = exit
1 = viewChannels()
2 = deviceConfiguration()
3 = dataFileHeadings()
Enter valid input: '0', '1', '2' or '3'
Valid input
END
0. Return to 'Main Menu'
main()
1. SDI-12 Channels
viewChannels()
2. SDI-12 Device Configuration
deviceConfiguration()
This feature is not functional. Return to main menu
Display: Only one SDI-12 sensor to be connected to the bus. Output 'SDI-12 Device Configuration' options:
0. Return to 'Main Menu'
1. Change Address of SDI-12 Sensor
2. Add SDI-12 Device
3. Start Logging
dataFileHeadings()
Key task; Prints headings to datafile if needed
1. Change Address of SDI-12 Sensor (feature no functional)
measurementDelay()
Key task; call the measurement handler function (takeMeasurement()) on specific measurement interval. Write data returned takeMeasurement() to datafile.
Return to SDI-12 Device Config menu once finished
2. Add SDI-12 Device
Add SDI-12 device handler
takeMeasurement()
measurementDelay()
address = sendAndReceive(?!)
Measurement handler. Initiates exchanges with all configured sensor and puts measurement data into a csv string returns to measurementDelay().
First function called from add SDI-12 device handler. Key task; send address query command (?!) and return a valid address
deviceConfiguration()
sendAndReceive()
takeMeasurement()
checkAddress()
Second function called from add SDI-12 device handler. Key task; check configuration file to confirm address is unique
Function called twice for each configured sensor. The function sends a 'start measurement' command followed by a 'send data' command
deviceConfiguration()
model = sendAndReceive(aI!)
Third function called from add SDI-12 device handler. Key task; send identification command (<a>I!) and return a sensor model
deviceConfiguration()
addChannels()
Fourth function called from add SDI-12 device handler. Key task; change multiple configuration file entries
deviceConfiguration()

Figure 29 Overview of functions called and key tasks in SDI-12 logger program

4.3.2 Organisation of SDI-12 Logger Program

Functions within the SDI-12 Logger program are organised under four sections. The four section of code are;

1) Main
2) Measurement Handling Functions
3) SDI-12 Device Configuration Functions
4) Generic Functions

Generally the set of functions in each section complete a task or set of task as initiated by a user input. The ‘main’ section contains the main function which is the first function called after starting the SDI-12 logger program. The ‘measurement handling’ section contains the functions called when main menu option 3 (‘Start Logging’) is selected by the user. The ‘device configuration’ section contains functions called when main menu option 2 (‘SDI-12 Device Configuration’) is selected. The ‘generic’ section contains functions common to two or more sections.

4.3.3 Main

The ‘main’ section has one function named main() described below. A flowchart of this function is presented in figure 30.

main() - the first function called in a C/C++ program. The task of this function is to setup and initialise the pin states, output ‘main menu’ options to the terminal command prompt and call the generic function getInteger(). getInteger() is responsible for returning a valid user input (see section 4.3.5: Generic functions).

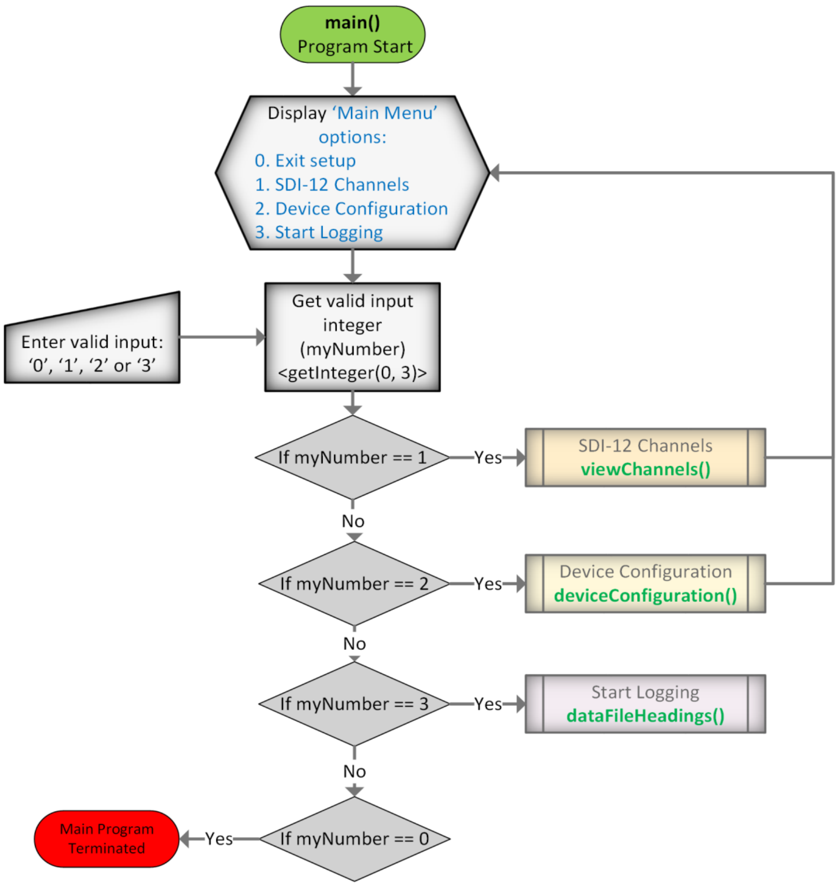

Figure 30 Flowchart of main()

4.3.4 Measurement Handling Functions

Functions in the measurement handling section with addition to the generic 'sendAndReceive' function perform the task of logging. To initiate the logging session from the 'Main' menu (assuming the logger has been pre-configured), user enters option 3 ('Start Logging'). The measurement handling functions include;

- dataHeadings()
- measurementDelay()
- takeMeasurement()

The 'sendAndReceive' function is called from the 'takeMeasurement' function twice for each configured sensor. The first time it is called it sends the 'start measurement' command and the second time it sends a 'send data' command. Figure 31 below gives a basic flowchart of showing functions called when option three is selected from the ‘Main’ menu. The flowchart shows the flow of external data between the functions, configuration file and data file.

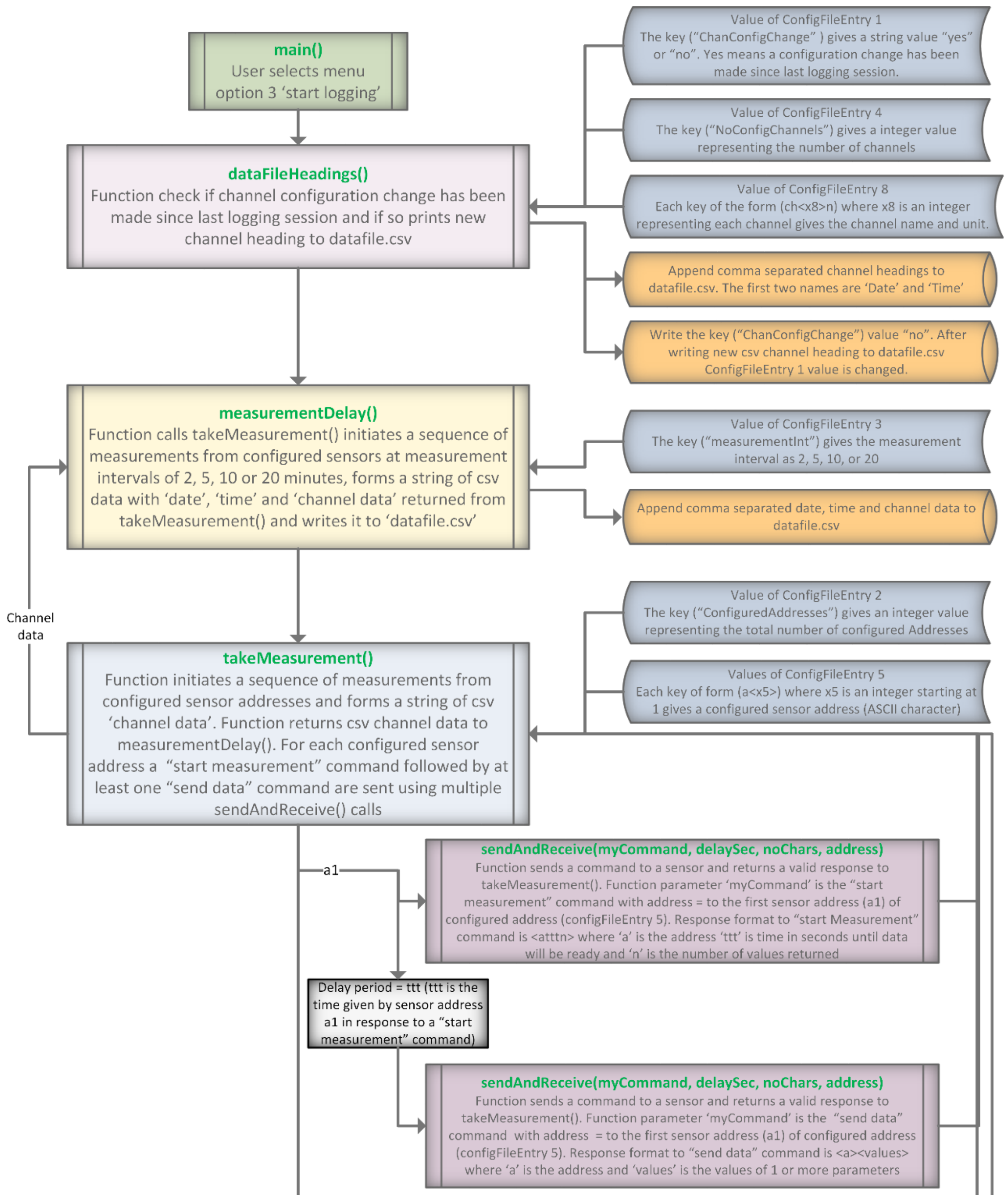
main()
User selects menu option 3 ‘start logging’
dataFileHeadings()
Function check if channel configuration change has been made since last logging session and if so prints new channel heading to datafile.csv
Value of ConfigFileEntry 1
The key (“ChanConfigChange”) gives a string value “yes” or “no”. Yes means a configuration change has been made since last logging session.
Value of ConfigFileEntry 4
The key (“NoConfigChannels”) gives a integer value representing the number of channels
Value of ConfigFileEntry 8
Each key of the form (ch<x8>n) where x8 is an integer representing each channel gives the channel name and unit.
Append comma separated channel headings to datafile.csv. The first two names are ‘Date’ and ‘Time’
Write the key (“ChanConfigChange”) value “no”. After writing new csv channel heading to datafile.csv ConfigFileEntry 1 value is changed.
measurementDelay()
Function calls takeMeasurement() initiates a sequence of measurements from configured sensors at measurement intervals of 2, 5, 10 or 20 minutes, forms a string of csv data with ‘date’, ‘time’ and ‘channel data’ returned from takeMeasurement() and writes it to ‘datafile.csv’
Value of ConfigFileEntry 3
The key (“measurementInt”) gives the measurement interval as 2, 5, 10, or 20
Append comma separated date, time and channel data to datafile.csv
Channel data
takeMeasurement()
Function initiates a sequence of measurements from configured sensor addresses and forms a string of csv ‘channel data’. Function returns csv channel data to measurementDelay(). For each configured sensor address a “start measurement” command followed by at least one “send data” command are sent using multiple sendAndReceive() calls
Value of ConfigFileEntry 2
The key (“ConfiguredAddresses”) gives an integer value representing the total number of configured Addresses
Values of ConfigFileEntry 5
Each key of form (a<x5>) where x5 is an integer starting at 1 gives a configured sensor address (ASCII character)
a1
sendAndReceive(myCommand, delaySec, noChars, address)
Function sends a command to a sensor and returns a valid response to takeMeasurement(). Function parameter ‘myCommand’ is the “start measurement” command with address = to the first sensor address (a1) of configured address (configFileEntry 5). Response format to “start Measurement” command is <atttn> where ‘a’ is the address ‘ttt’ is time in seconds until data will be ready and ‘n’ is the number of values returned
Delay period = ttt (ttt is the time given by sensor address a1 in response to a “start measurement” command)
sendAndReceive(myCommand, delaySec, noChars, address)
Function sends a command to a sensor and returns a valid response to takeMeasurement(). Function parameter ‘myCommand’ is the “send data” command with address = to the first sensor address (a1) of configured address (configFileEntry 5). Response format to “send data” command is <a><values> where ‘a’ is the address and ‘values’ is the values of 1 or more parameters

Figure 31 Flowchart for SDI-12 logging (option 3 from ‘Main’ menu) and flow of external data

This section gives a detailed description of the three measurement handling functions.

dataFileHeadings() - a function called from main() when menu option 3 is selected. Figure 32 gives a flowchart for the ‘dataFileHeadings’ function. The sequence of key tasks performed is:

- Check if any channels are configured, if not return to main menu.
- Checks the logger configuration file to see if any channels have been added since the last logging session. If a change has been made new channel headings are appended to the bottom of the existing data file and configuration file subsequently updates. A channel headings string begins with ‘Date,Time,’ and is followed by a string of comma separated channel names. Channel names have the following structure (<address of sensor>_<parameter name> (<parameter units>)).

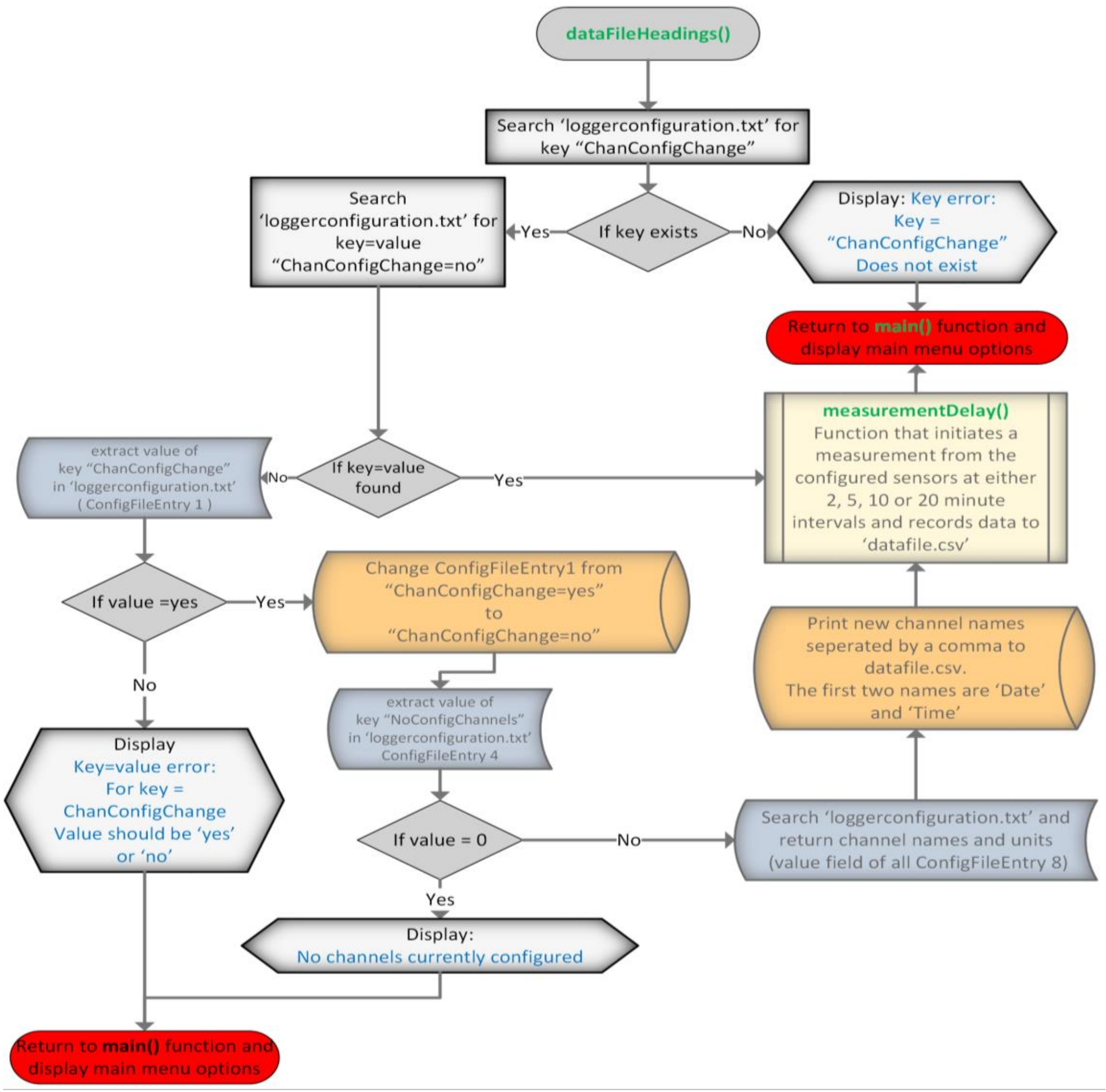
dataFileHeadings()
Search ‘loggerconfiguration.txt’ for key “ChanConfigChange”
If key exists
Yes
No
Display: Key error: Key = “ChanConfigChange” Does not exist
Return to main() function and display main menu options
Search ‘loggerconfiguration.txt’ for key=value “ChanConfigChange=no”
If key=value found
measurementDelay() Function that initiates a measurement from the configured sensors at either 2, 5, 10 or 20 minute intervals and records data to ‘datafile.csv’
extract value of key “ChanConfigChange” in ‘loggerconfiguration.txt’ (ConfigFileEntry 1)
If value =yes
Change ConfigFileEntry1 from “ChanConfigChange=yes” to “ChanConfigChange=no”
Print new channel names seperated by a comma to datafile.csv. The first two names are ‘Date’ and ‘Time’
extract value of key “NoConfigChannels” in ‘loggerconfiguration.txt’ ConfigFileEntry 4
Display Key=value error: For key = ChanConfigChange Value should be ‘yes’ or ‘no’
If value = 0
Search ‘loggerconfiguration.txt’ and return channel names and units (value field of all ConfigFileEntry 8)
Display: No channels currently configured
Return to main() function and display main menu options

Figure 32 Flowchart for dataFileHeadings()

measurementDelay() - a function called from dataFileHeadings(). Key tasks performed are:

- Retrieve measurement interval from configFileEntry 3. Valid measurement intervals are 2, 5, 10, and 20 minutes.
- Call takeMeasurement() on a specific minute and second of the hour. Reference time is hh:mm:ss = hh:00:00. E.g. if a measurement interval is every 5 minutes and logging is started at 5:06:10 the takeMeasurement function will be called at 5:10:00 followed by 5:15:00 and so on.
- Append a string of data to the bottom of the datafile. The first two comma separated values are <current date> and <current time>, followed by a string of comma separated channel values returned from takeMeasurement().

Figure 33 presented below gives a flowchart for the 'measurementDelay' function

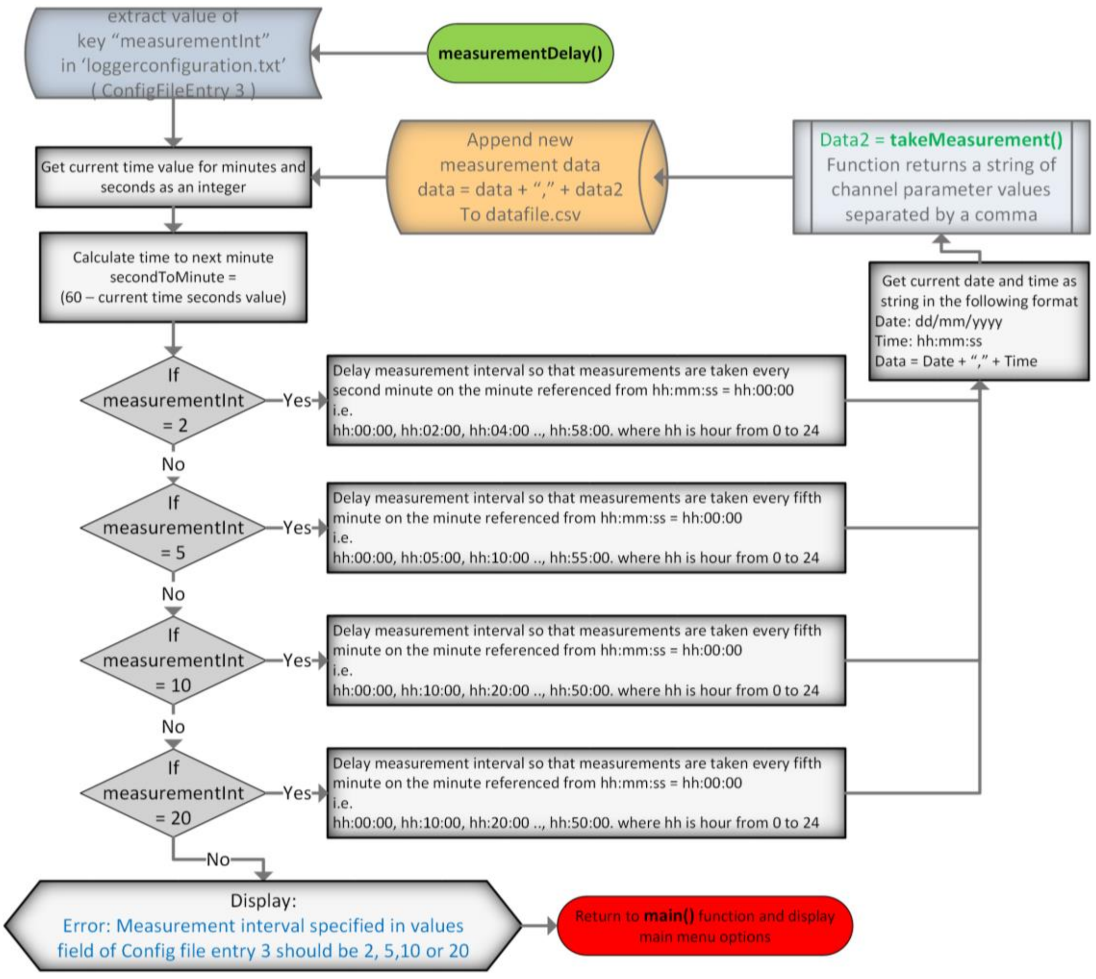
measurementDelay()
extract value of key "measurementInt" in 'loggerconfiguration.txt' (ConfigFileEntry 3)
Get current time value for minutes and seconds as an integer
Calculate time to next minute secondToMinute = (60 – current time seconds value)
Append new measurement data data = data + "," + data2 To datafile.csv
Data2 = takeMeasurement() Function returns a string of channel parameter values separated by a comma
Get current date and time as string in the following format Date: dd/mm/yyyy Time: hh:mm:ss Data = Date + "," + Time
If measurementInt = 2
Yes
No
Delay measurement interval so that measurements are taken every second minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:02:00, hh:04:00 .., hh:58:00. where hh is hour from 0 to 24
If measurementInt = 5
Delay measurement interval so that measurements are taken every fifth minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:05:00, hh:10:00 .., hh:55:00. where hh is hour from 0 to 24
If measurementInt = 10
Delay measurement interval so that measurements are taken every fifth minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:10:00, hh:20:00 .., hh:50:00. where hh is hour from 0 to 24
If measurementInt = 20
Delay measurement interval so that measurements are taken every fifth minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:10:00, hh:20:00 .., hh:50:00. where hh is hour from 0 to 24
Display: Error: Measurement interval specified in values field of Config file entry 3 should be 2, 5,10 or 20
Return to main() function and display main menu options

Figure 33 Flowchart for measurementDelay()

takeMeasurement() - a function called from measurementDelay(). This function is responsible for getting channel data from configured sensor. The sequence of key tasks performed are:

- Read the number of configured addresses from configFileEntry 2.
- LOOP 1 (outer loop) – repeats for each configured sensor address in order they are added to configuration file.
    - Find address of next sensor by creating a sequential key for configFileEntry 5 and extract value.
    - Find number of parameters returned by the sensor by creating a unique key in the configuration file using the address of the sensor (key = add<a> where, ‘a’ is the address).
    - Create a ‘start measurement’ command string (aM!) where ‘a’ is the address of the sensor.
    - Call sendAndReceive(myCommand, delaymSec, noChars, address), where function parameters ‘myCommand’ is the ‘start measurement’ command, delaymSec is the minimum delay time for response based on the maximum number of characters returned, noChars is the minimum number of characters returned (some responses are variable in length), address is the address of sensor. The sendAndReceive function sends the command and checks for a valid response. The sensor response is returned to takeMeasurement from sendAndReceive() as either valid or invalid. A valid response to command is atttn<CR><LF>, where ttt is the time in seconds until the sensor will have measurement data ready. ‘a’ ad ‘n’ are not used.
    - If invalid response “comm error” is printed to the corresponding channels data and program continues at LOOP 1 incrementing to next address.
    - If response was valid, the delay is converted to an integer and delay time set.
    - LOOP 2 (inner loop) – repeats if multiple ‘send data’ commands need to be sent to the current sensor address to get all data.
        - Create a ‘send data’ command (aDx!) where ‘a’ is the address of the sensor and ‘x’ is a sequential number starting at 1. The response to this command is a<values><CR><LF>. The values field is up to 35 characters and each parameter of a multi-parameter sensor is of variable length up to a maximum of 9 characters which includes a polarity sign. Multiple parameters values are returned in a values field and the number of parameter values returned is

unknown. That is an unknown number of parameter values are returned in each 'send data' command.

- The sensor response is returned to takeMeasurement from sendAndReceive() as valid or invalid.
- If invalid response, "No Value Returned" printed to the corresponding channels data and program continues at LOOP 1 incrementing to next address.
- If valid response, each parameter is extracted by searching for '+' or '-' polarity signs.
- If the polarity count is equal to the total number of parameter returned by the sensor, the channel values are recorded and program continues at LOOP 1 incrementing to next address. If the polarity count is not equal to the total number of parameter returned by the sensor, the channel values are recorded and program continues at LOOP 2 incrementing to 'x' in the 'send data' command.

A flowchart for the 'takeMeasurement' function is given in appendix E.

4.3.5 Device Configuration Functions

The most important configuration feature is an HMI for adding SDI-12 sensors to the logger configuration. Functions presented in the device configuration section with addition to the generic 'sendAndReceive' function will perform this feature. To add an SDI-12 sensor to the logger configuration from the 'Main' menu, user enters option 2 ('SDI-12 Device Configuration'), then from the 'Device Configuration' menu, user enters 2('Add SDI-12 Device'). Functions called in add SDI-12 device include;

- deviceConfiguration()
- checkAddress()
- addChannels()

This section gives a detailed description of the three device configuration functions.

deviceConfiguration() - a function called from main() when 'Main' menu option 2 is selected. The sequence of key tasks performed are:

- Output ‘device configuration menu’ options to the terminal command prompt.
- Call the generic getInteger() function. getInteger() is responsible for returning a valid user input to the terminal command prompt (see section 4.3.5: Generic functions).
- If option 2 is entered (‘Add SDI-12 Device’) the generic sendAndReceive function is called
- Create a ‘query address’ command string (?!).
- Call generic sendAndReceive(myCommand, delaymSec, noChars, address), where function parameters ‘myCommand’ is the ‘query address’ command, delaymSec is the minimum delay time for response with 3 characters (i.e. delaymSec = (0.833us x 10 x 3)+25 = 50 ms), noChars is the minimum number of characters returned (three characters for ?! command), address is unknown and is passed is a fake address. The sendAndReceive function sends the command and checks for a valid response. The sensor response is returned to deviceConfiguration from sendAndReceive() as either valid or invalid.
- If address is used return to ‘device configuration’ menu.
- If address is not used call checkAddress(address) to check the configuration file to see if the sensor address is already being used by another configured sensor.
- Create a ‘send identification’ command string (aI!) where, ‘a’ is the sensor address.
- Call generic sendAndReceive(myCommand, delaymSec, noChars, address), where function parameters ‘myCommand’ is the ‘send identification’ command, delaymSec is the minimum delay time, noChars is the minimum number of characters returned (35 for aI! command). The sendAndReceive function sends the command and checks for a valid response. The sensor response is returned to deviceConfiguration from sendAndReceive() as either valid or invalid. A valid response to command is a<CR><LF>, where ‘a’ is the address of the sensor.
- Display sensor model to command prompt and output 1. Yes or 2. No and wait for user response.
- If sensor model is not correct return to ‘device configuration’ menu.
- If correct model is displayed call addChannels(model address) which searches the searches the database for the sensor model and adds sensor to the configuration file.

A flowchart for this function is given in appendix E.

addChannels(model, address) – the fourth function called from deviceConfiguration() that performs the main task when option 2 (‘Add SDI-12 Device’) of the ‘device configuration’ menu is selected. The function adds a new SDI-12 sensor to the configuration file taking a 6 character sensor model and a single character sensor address as parameters. The function rewrites the logger configuration file adding or modifying lines within each section of the configuration file. This function searches the sensor database (see section: 4.2.2 Specification for the database of SDI-12 sensors) and extracts the parameter names and unit for each parameter in order that they are returned. Add channels combines the address of the sensor with the parameter names and unit ad adds it to the configuration file.

4.3.6 Generic Functions

This section gives a description of the most important generic function.

sendAndReceive(command, delay, noChars, address) – a function called to send an SDI-12 command (function parameter) to the SDI12 bus and delay for a period (function parameter) before checking a valid response and reading from the buffer. Public functions from the SDI-12 library described in section 4.1.5 are called to check a valid response and read buffer contents. If data is not valid the command is resent to the sensor otherwise the function returns a valid response. A flowchart for this function is given in appendix E.

Chapter 5: Analysis and Performance

The first section of this chapter demonstrates a simple SDI-12 exchange captured with an oscilloscope and through the terminal command prompt. The second section of this chapter presents some data obtained during the testing of the logger with three sensor attached.

5.1 Analysis of SDI-12 Exchanges

In this section the SDI12Logger program is tested. The data line voltage is analysed using an oscilloscope and the program terminal command output is presented.

5.1.1 Test Description

An address query (?!) command is sent to an SDI-12 sensor for analysis of the signal voltages and timing at key points. The 7 bit ASCII code for the two characters in the 'address query' command '?' and '!' are provided below for reference.

? = 63 (Decimal) = 011 1111 (Binary).

! = 33 (Decimal) = 010 0001 (Binary).

The LSB for the binary representation is given on the right side of the page. The even parity bit is not shown but would be 0 for both. An SDI-12 sensor with address set as 6 will respond to an address query command with 6<CR><LF>. The 7 bit ASCII code for the three characters in the response are as follows:

6 = 54 (Decimal) = 011 0110 (Binary).

CR = 13 (Decimal) = 000 1101 (Binary).

LF = 10 (Decimal) = 000 1010 (Binary).

The expected sequence of SDI-12 data line states during the transmission of the command is given below.

1 0000 0011 0 (?) → 1 0111 1011 0 (!) (5.1)

The first bit in the asynchronous command (first bit is the start bit of frame with '?' character) is shown on the left of the page. The start bit of a frame is shown in blue text, stop bits in red, even parity bits in green and data bits in black. Data bits are sent using inverse logic with least significant bit first. The expected sequence of data line states during the response is given below.

1 1001 0011 0 (6) → 1 0100 1110 0 (CR) → 1 1010 1111 0 (LF) (5.2)

5.1.2 SDI-12 Data Line Waveform Analysis

Figure 34 shows the bus data line signal as captured with an oscilloscope. The main time scale division is 10 ms and the voltage division is 1 volt. The command voltage is close to 4.5 V and the response about 3.7 volts. The minimum SDI-12 data line voltage is 3.5 volts so the sensor is performing to specification.

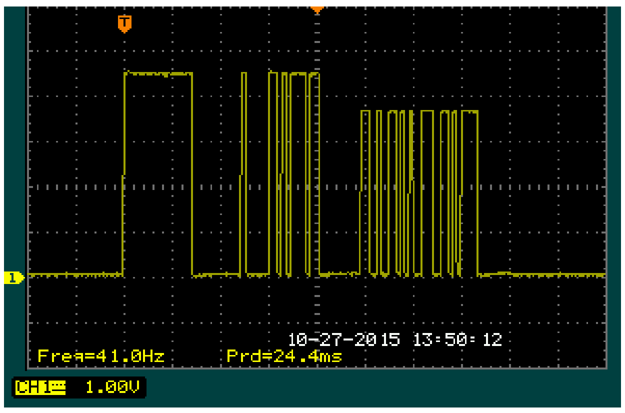

Figure 34 Data line waveform showing transmission of command '?!' and sensor response 6<CR><LF>

The oscilloscope triggered on the first logic HIGH (left side of the screen). The first logic HIGH followed by logic LOW is the wakeup sequence to wake the sensor on the bus. By the end of the wakeup sequence (duration of 14ms + 10 ms) the sensor must be ready to receive the command. From figure 34 the sensor response starts approximately 7.5 seconds after command finishes with a stop bit. The maximum time as specified in the standard is 15 ms and so the sensor is working to specification. Figure 35 shows the command waveform only which includes two 10 bit frames, and figure 36 shows three 10 bit frames of the sensor response only. The main time scale division and voltage scale for the oscilloscope in figure 35 and 36 is 1 ms and 1 V respectively. The waveforms are a match to binary sequence in 5.1 and 5.2.

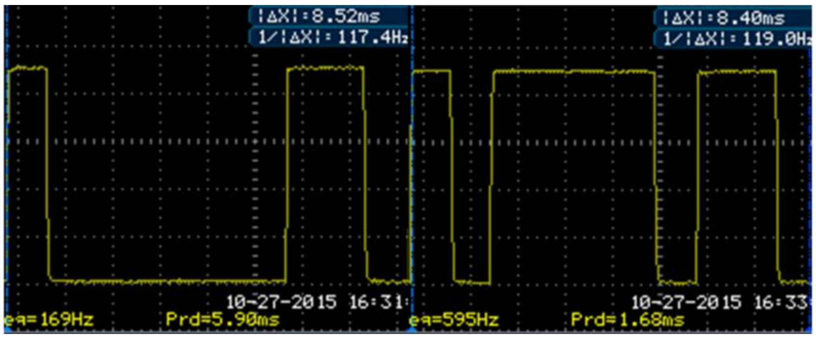

Figure 35 Oscilloscope display capturing transmission of address query command (?!)

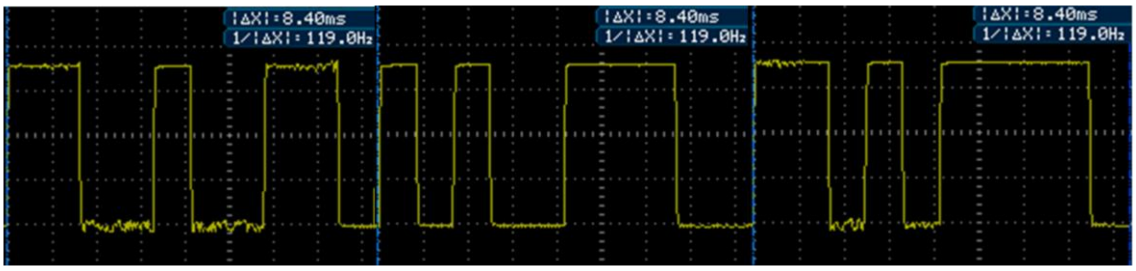

Figure 36 Oscilloscope display capturing sensor response to address query command (6<CR><LF>)

Figure 37 gives both the SDI-12 data line waveform (yellow) and the RXDATAPIN waveform (blue) captured for the address query test. It shows both the command and response.

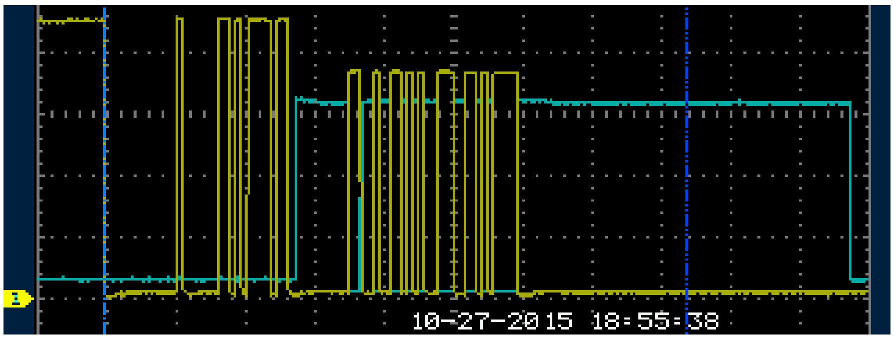

Figure 37 Data line waveform (yellow) and RXDATAPIN waveform (blue)

Figure 37 shows that at the end of the command signal the blue waveform rises and is interpreted as the SN74HCT240 output pin connected to the RXDATAPIN (1Y1) is enabled and so RXDATAPIN is the inverse of the input waveform. The blue line can be seen to return to logic LOW towards the end of the oscilloscope when the state is changed from listening to disabled. Figure 38 shows the sensor response part of the waveform presented in figure 37 only. The RXDATAPIN voltage are the inverse of the bus voltages.

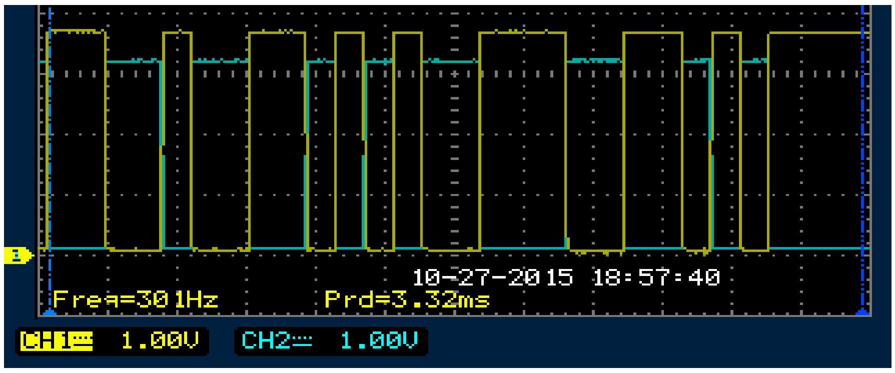

Figure 38 Data line waveform (yellow) and RXDATAPIN waveform (blue) during sensor response

5.1.3 Terminal Command Prompt Analysis

During the development of the software it was necessary to track the program execution for diagnostics purposes and analysis. Key diagnostic are printed to the terminal command prompt. The information captured for the address query test is shown below. To perform the test the user enters option 2 in ‘Main’ menu and option 2 in ‘Device Configuration’ menu.

```
Main Menu - Enter an integer from '0' to '3' and press enter.
 0. Exit Setup
 1. SDI-12 Channels                (not started)
 2. SDI-12 Device Configuration    (partially completed)
 3. Start Logging                  (working - partially complete)
please enter a valid number .....
2
deviceConfiguration() called

SDI-12 Device Configuration Menu
Only one SDI-12 device should be connected to the SDI-12 bus 3
Enter an integer from '0' to '2' and press enter.
 0. Return to Main Menu
 1. Change address of SDI-12 sensor   (not started)
 2. Add SDI-12 device                 (partially complete)
please enter a valid number .....
2
sendAndReceive() called
Constructor() Called
flush() called
begin() Called
setState = INTERRUPTENABLE
String sent to SDI12 bus:?!
sendCommand() called
wakeSensors() Called
SetState = TRANSMITTING
sendCommand() Next Character out is:  ?
writeChar() Next character out is:   ?
sendCommand() Next Character out is:  !
writeChar() Next character out is:   !
SetState = LISTENING
handleInterrupt() called.......................
receiveChar() called
Pin level LOW:
Pin level HIGH:
Pin level HIGH:
Pin level LOW:
Pin level HIGH:
Pin level HIGH:
Pin level LOW:
Pin level LOW:
newChar in ASCII: 6
ReceiveChar() parity of newChar (0 = even, 1 = odd): 0
handleInterrupt() called.......................
receiveChar() called
invalid start bit
handleInterrupt() called.......................
receiveChar() called
Pin level HIGH:
Pin level LOW:
Pin level HIGH:
Pin level HIGH:
Pin level LOW:
Pin level LOW:
Pin level LOW:
Pin level HIGH:
newChar in ASCII:
ReceiveChar() parity of newChar (0 = even, 1 = odd): 0
handleInterrupt() called.......................
receiveChar() called
invalid start bit
handleInterrupt() called.......................
receiveChar() called
Pin level LOW:
Pin level HIGH:
Pin level LOW:
Pin level HIGH:
Pin level LOW:
Pin level LOW:
Pin level LOW:
Pin level LOW:
newChar in ASCII:
ReceiveChar() parity of newChar (0 = even, 1 = odd): 0
```

```
handleInterrupt() called.......................
receiveChar() called
invalid start bit
end() Called
SetState = DISABLED
available() called
Number of characters in buffer:  3
parityError() called
overflowStatus() called
LFCheck() called
CRCheck() called
sendAndReceive() valid response after:  0 resend attempts
flush() called
Destructor() called
SetState = DISABLED
deviceConfiguration() Address of sensor is: 6
checkAddress() called
checkAddress() configFileEntry6 key is: add6

Address of sensor connected is used by another configured sensor. The
address of the sensor must be changed before the sensor can be added.

SDI-12 Device Configuration Menu Options
Only one SDI-12 device should be connected to the SDI-12 bus 3
Enter an integer from '0' to '2' and press enter.
 0. Return to Main Menu
 1. Change address of SDI-12 sensor   (not started)
 2. Add SDI-12 device                 (partially complete)
please enter a valid number:
```

The test where an address query command is sent to a sensor is repeated 30 times and in all cases the correct address was returned without the program having to resend the command. Some of the diagnostics captured (shown above) are highlighted in green for further explanation. After the first character is successfully received from the sensor (ASCII char 6) by the ISR function (receiveChar()), the program returns to the sendAndReceive() function where it continues a predefined delay period. The first handleInterrupt() called is because the wiringPiISR() function clears the interrupt register at the start of an ISR, meaning that if a falling edges transition occurs while servicing an interrupt the interrupt is registered and a subsequent handleInterrupt() is called immediately after finishing the routine. As the first instance of handleInterrupt() finds an invalid start bit it is re-reading the stop bit from the previous frame which indicates that the timing of the receiveChar() function is good. It is preferable to read the pin state at the beginning of the RXDATAPIN bit transitions.

5.2 Testing of Raspberry Pi SDI12 Logger with Three Sensor Attached

The logger is tested with three sensors including two Decagon GS3's measuring soil moisture, temperature, and electrical conductivity and a Decagon 5TM measuring soil moisture and temperature. With all these parameters being logged there are 8 configured channels. The logger was tested for 75 hours with logging interval of 5 minutes. The datafile grew to 880 lines of data over the test period. All three sensor are left sitting on a carpet surface for the duration of the test. As the parameter measurement values are not expected to change dramatically the errors in the datafile can easily be spotted. Channels 2, 5 and 7 (temperature parameters returned by the 3 sensor) are plotted (figure 39). Some erroneous data specifically very large numbers had to be removed before plotting so that the data trend can be observed. These values appear as a zero instead. These results are obviously unacceptable.

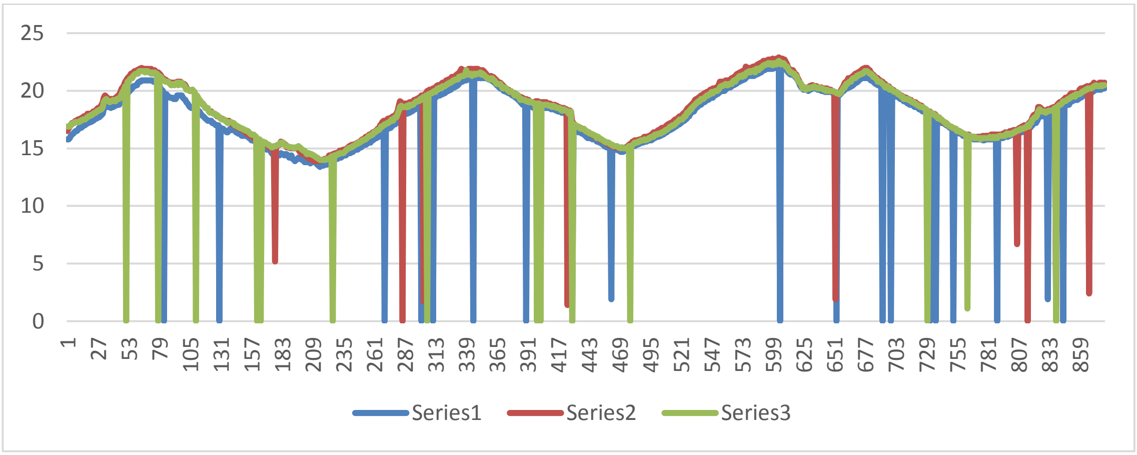

Figure 39 Plot of temperature data from 3 SDI-12 sensor

Of the 880 lines of data entered into the datafile during the test, 63 are found to contain erroneous entries (7%). A mix of erroneous values and diagnostic error messages are present in the datafile. Table 9 gives a modified version of the datafile where all the good lines of data have been removed and erroneous entries highlighted.

Table 9: Erroneous data recorded over 24 hour test period logging at 5 minute intervals

Date	Time	6 Dielec-tric (e)	6 Temper-ature (Deg.C)	6 Electrical Conducti-vity (uS/cm)	A Dielec-tric (e)	A Soil Tempera-ture (Deg.C)	8 Dielect-ric (e)	8 Tempera-ture (Deg.C)	8 Electrical Conductivity (uS/cm)
24/10/2015	13:35:00	1.77	15.8	1	1.12	16.5	1.53	16.9	1
24/10/2015	17:50:00	1.8	19.6	1	1.12	20.9	No Result Error	No Result Error	No Result Error
24/10/2015	20:05:00	1.81	20.6	1	1.11	21.6	1.59	214	1
24/10/2015	20:30:00	Comm. Error	Comm. Error	Comm. Error	1.13	21	1.54	20.9	1
24/10/2015	21:00:00	1.71	19.4	🟥	1.12	20.6	1.56	20.5	1
24/10/2015	22:45:00	1.74	18.5	1	1.11	19.6	1.59	197	1
25/10/2015	0:25:00	1.64	169	1	1.11	17.7	1.55	17.7	1
25/10/2015	2:45:00	1.68	15.7	Q	1.11	16	1.57	16.2	1
25/10/2015	2:50:00	1.74	15.7	2	1.11	15.9	1.59	16.2	1
25/10/2015	3:05:00	1.66	15.6	1	1.1	15.7	No Result Error	No Result Error	No Result Error
25/10/2015	3:20:00	1.69	15.3	1	1.11	15.5	No Result Error	No Result Error	No Result Error
25/10/2015	3:40:00	172	15	1	1.11	15.3	1.55	15.5	1
25/10/2015	4:00:00	1.61	14.8	🟥	1.1	15	1.59	15.2	1
25/10/2015	4:20:00	1.61	14.4	1	1.11	5.2	1.58	15.1	1
25/10/2015	7:45:00	1.7	13.5	1	109	14	1.57	14	1
25/10/2015	8:40:00	1.66	14.1	1	1.1	14.6	No Result Error	No Result Error	No Result Error
25/10/2015	12:05:00	🟥	Corrupt value Error	Corrupt value Error	1.11	17.4	1.56	16.9	1
25/10/2015	13:20:00	1.67	17.9	1	1.11	190	1.57	18.3	1

There are a range of errors captured in the data file. These errors are described below.

Corrupt value Error – this error occurs when the number of polarity signs in the values field of the sensor response to a 'send data' command is not equal to the number of parameters returned by the sensor. The sensors used in testing the logger should return all parameters in the first 'send data' command. For this error to occur there must not have been a parity error or incorrect stop bit, and the last two character <LF> and <CR> valid. The number of these errors in all 2640 sensor queries was 11.

Comms Error – this error occurs after 10 failed attempts at receiving a valid response to the 'start measurement' command (7 character response expected). The error may be due to hardware or due to the operating system interrupting the receiveChar() function. For this error to occur all responses must have been invalid i.e. a parity error, incorrect stop bit or one of the last two character <LF> and <CR> not valid.. The number of these errors in all 2640 sensor queries was 2.

No Result Error – this error occurs after 10 failed attempts at receiving a valid response to the 'send data' command (between 12 and 30 characters in response expected). The error may be due to hardware or due to the operating system interrupting the receiveChar() function. For this error to occur all responses must have been invalid. The number of these errors in all 2640 sensor queries was 12.

One incorrect parameter – this is not a diagnostic message but a single incorrect character is read. For this error to occur there must not have been a parity error or incorrect stop bit, and the last two character <LF> and <CR> valid. The number of these errors in all 2640 sensor queries was 41.

As expected the 'no result error' occurs more than the 'comms error' because the response to 'send data' command is longer than the 'start measurement' command. The fact a valid response cannot be read after 10 resend attempts is a concern because not even the CRC can reduce these errors. The resend attempts can be increased from 10 to 20. The precise timing of the receiveChar() function should be optimised.

Chapter 6: Conclusions and Further Work

6.1 Achievement of Project Objectives

Stage 1 objective - a review of environmental monitoring. It is apparent that SDI-12 is a good choice for many data logging requirements. Setting up a network requires an SDI-12 compatible data logger, SDI-12 compatible sensors and setup of a bus. SDI-12 provides benefits which include plug and play modularity and a growing number of SDI-12 compatible sensors. A problem in the distributed measurement and control (DMC) industry is lack of standardisation which is addressed by IEEE 1451. Sensor manufacturers may opt to produce sensors that can be interfaced to IEEE 1451 compatible transducer interface module (TIM) because of the significant market opportunity. Development of a STIM (smart transducer interface module) for SDI-12 or any other digital or analogue sensor is a potential future project. A sensor plus STIM system for integration into an IEEE 1451 network capable application processor is much more complex then setting up an SDI-12 network. SDI-12 is a good fit for many projects. Another project idea is development of a hybrid wireless SDI-12 host-to-SDI-12 bus to give full flexibility for many common application such as green houses. The selection of a logger is likely to be influenced by whether wired or wireless sensors are needed and on the desired or available protocol.

Stage 2 objectives - conceptual design and background information. All stage 2 objectives are achieved. Adopting the GPIO pin approach for the SDI-12 digital interface, there is an inherent risk of unreliable exchanges as the Raspberry Pi schedules the processes including the daemon programs. A subset of the commands are chosen that will allow configuration of the logger through the HMI and logging of configured sensors. This subset of commands chosen are compatible with all SDI-12 sensors (including those based on earlier versions of the SDI-12 specification). Individual measurement commands are supported. Concurrent measurement commands and cyclic redundancy check (CRC) commands are not supported.

As the sole software developer and no previous experience in programming the conceptual design is a process of experimentation. The conceptual design of the software modules requires some skills and knowledge of the programming language (C/C++) and software structure and also some experience in modular programming would be beneficial. In retrospect a bottom up approach would have worked, starting with the most fundamental concept to understand. In this project, the first concept to understand in the GPIO approach is how to write bits of a variable holding an ASCII character to the sensors and add bits to another variable as read from the asynchronous serial data line. As this concept and other smaller concepts were not obvious an experimental approach was needed.

Stage 3 objectives – development and test. All stage three objectives are achieved. An open source Arduino SDI-12 library was analysed and it was determined that it could be modified for use with the Raspberry Pi based logger. The hardware was designed and tested using an oscilloscope. The configuration file was developed to allow configuration and data logging. The main configuration feature allows a user to add a new SDI-12 sensors through the HMI.

6.2 Evaluation of the Logger

The aim was to create a low cost and reliable data logger that is simple to use. While the Raspberry Pi and the SDI-12 sensors are reliable there is a high rate of bad data being recorded to the datafile. There needs to be further work to assess the suitability of the GPIO method.

The configuration process though the HMI is straight forward for anyone with experience in using the Raspberry Pi command line. The user will need to execute the program using the command prompt and navigate the HMI. The Raspberry Pi logger would be appropriate in a range of application where it can be powered continuously with about 4 watts and can be protected from its environment. In developing the software there was no signs that the Raspberry Pi computer was not up to the challenge of running continuously for long periods of time.

6.3 Further Work

While the logger is functional, the rate of errors in the datafile is unacceptable. Considering an SDI-12 sensor can return 1 to 9 parameters but say typically a sensor returns 3 parameters, the response to a 'send data' command will return between 12 and 30 chars. If the received data is invalid the command is resent to the sensor up to 10 times (until a valid response is received). Adding a diagnostic column to the data file for each sensor where the number of times the 'start measurement' command is sent to each sensor can be recorded would provide a means of assessing the overall performance of the logger. If the 'send data' command is consistently being sent around 10 times, then implementing the 'send data' command with a CRC may not be an appropriate solution. If the commands are consistently being sent around the maximum number of times, then the timing of the receiveChar() function should also be checked and optimised. However, it is likely that the erroneous data issues could be addressed by implementing the CRC 'send data' command. The CRC 'send data' command should be implemented and tested. If the test is unsuccessful the UART approach should be considered. If test is successful the basic features should be finished. Consideration may also be given to developing a more marketable product. Further work includes:

- Finish the HMI to allow full configuration of sensors and changing channels. The HMI needs to display the list of configured channel names.
- Develop a graphical user interface possibly using QT development environment if can successfully install it on the Raspberry Pi.
- Leverage the network capabilities of the Raspberry Pi. Allow remote access to the logger for setting and disabling alarms. Develop an SQL database for remote access of data.
- Look at security applications for the logger, leveraging the video processing and networking capabilities. A camera could be installed and accessed remotely for real-time security to provide assess a security threat.
- Make full use of all commands and use the most appropriate command for any particular sensor. The most appropriate command will depend on what version of the SDI-12 specification the sensor complies with.

References

Arduino 2015, Arduino, viewed 2 March 2015, <http://www.arduino.cc/>.

Bell, C 2013, *Beginning sensor networks with Arduino and Raspberry Pi,* Apress, California.

Boehm, B & Turner, R 2004, 'Balancing agility and discipline: Evaluating and integrating Agile and Plan-driven methods', *Software Engineering, 2004. ICSE 2004. Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, proceedings of a meeting held 23-28 May, IEEE, pp.718-719.

Broadcom Corporation 2012, *Broadcom BCM2835 ARM Peripherals,* datasheet, Broadcom Corporation, Cambridge, viewed 1 July 2015, <http://www.farnell.com/datasheets/1521578.pdf>.

Brooks, FP 1987, 'No silver bullet: Essence and accidents of software engineering', *Computer*, vol. 20, no. 4, pp. 10-19.

Decagon Devices 2015, Decagon Devices, Pullman, Washington, viewed 27 June 2015, <http://www.decagon.com/>.

Decagon Devices n.d., *5TM integrators Guide for Serial and SDI-12 Communications,* Decagon Devices, Pullman, Washington DC, viewed 27 June 2015, <http://manuals.decagon.com/Integration%20Guides/5TM%20Integrators%20Guide.pdf>.

Element14 2015, Element14, Sydney, NSW, viewed 20 June 2015, <http://au.element14.com/>.

Hać, A 2003, *Wireless sensor network designs,* John Wiley & Sons, Chichester, West Sussex.

Henderson, G 2015, Gordon Henderson, viewed 1 July 2015, < https://projects.drogon.net/raspberry-pi/wiringpi/ >

IEEE Standards Board 1997, *IEEE standard for a smart transducer interface for sensors and actuators – transducer to microprocessor communication protocols and transducer electronic data sheet (TEDS) formats,* IEEE std 1451.2-1997, IEEE Standards Board, New York.

IEEE Standards Board 2003, *IEEE standard for a smart transducer interface for sensors and actuators – Digital communication and transducer electronic data sheet (TEDS) formats for distributed multidrop systems,* IEEE std 1451.3-2003, IEEE Standards Board, New York.

Kumar, A, Kim, H & Hancke GP 2013, 'Environmental monitoring systems: A review', *IEEE Sensors Journal,* vol. 13, April, pp. 1329-1339.

Lee, KB & Schneeman, RD 2000, 'Distributed measurement and control based on the IEEE 1451 smart transducer interface standards', *IEEE Transactions on Instrumentation and Measurement,* vol. 49, no. 3, pp. 621-627.

Mackay, DJC 2008, *Sustainable energy without the hot air,* UIT, Cambridge.

OpenEnergyMonitor project 2015, OpenEnergyMonitor project, viewed 28 May 2015, <http://openenergymonitor.org/emon/>.

Priva 2015, Priva, De Lier, Netherlands, viewed 28 May 2015 <http://www.privagroup.com/en>.

Raspberry Pi Foundation 2015, Raspberry Pi Foundation, United Kingdom, viewed 30 May 2015, <https://www.raspberrypi.org/>.

Raspberry Pi Foundation 2015, *Raspberry Pi 2 Model B,* Raspberry Pi Foundation, UK, viewed 28 May 2015, <https://www.raspberrypi.org/>.

Raspberry Pi Foundation 2015, *Raspberry Pi 1 Model A+,* Raspberry Pi Foundation, UK, viewed 28 May 2015, <https://www.raspberrypi.org/>.

Scheiber SF 2001, *Building a successful board test strategy*, 2nd edn, Newnes, Oxford.

SDI-12 Support Group 2013, *SDI-12 a serial-digital interface standard for microprocessor-based sensors version 1.3,* SDI-12 specification, version 1.3, SDI-12 Support Group, Utah, viewed 20 March 2015, <http://www.sdi-12.org/index.php>.

SDI-12 Support Group 2015, SDI-12 Support Group, SDI-12 Support Group, Utah, viewed 26 June 2015 <http://www.sdi-12.org/index.php>.

Sinclair, IR 2001, *Sensors and transducers,* 3rd edn, Newnes, Oxford.

Texas Instruments 2014, *SN74LVC1T45 single-bit dual-supply bus transceiver with configurable voltage translation and 3-state outputs*, datasheet, Texas Instruments, Dallas, Texas, viewed 24 July 2015, < http://www.ti.com/product/sn74lvc1t45 >.

Texas Instruments 2014, *SN54HCT240, SN74HCT240 octal buffers and line drivers with 3-state outputs*, datasheet, Texas Instruments, Dallas, Texas, viewed 24 July 2015, < http://www.ti.com/lit/ds/symlink/sn74hct240.pdf>.

United Nations Environmental Programme - Sustainable Buildings and Climate Initiative 2009, *Buildings and climate change summary for decision-makers,*UNEP-SBCI, Paris, Viewed 31 May 2015, <http://www.unep.org/sbci/pdfs/SBCI-BCCSummary.pdf>.

Write, B & Dillon M n.d., *Application of IEEE P1451 'smart transducer interface standard' in condition based maintenance,* viewed 31 May 2015, <http://www.utdallas.edu/~venky/WP/Smart%20Transducer%20interface%20std%20in%20CBM(very%20good%20paper).pdf>.

YDOC 2015, YDOC, Almere, Netherlands, viewed 29 May 2015, <http://www.your-data-our-care.com/>.

Appendix A: Project Specification

FACULTY OF ENGINEERING AND SURVEYING

ENG4111/4112 Research Project

PROJECT SPECIFICATION

FOR:	**JAMES MACLEAN COPPOCK**
TOPIC:	Development of a Raspberry Pi based, SDI-12 sensor environmental data-logger
SUPERVISORS:	Dr Leslie Bowtell Catherine Hills
ENROLMENT:	ENG 4111 – S1, 2015 ENG 4112 – S2, 2015
PROJECT AIM:	Develop a Raspberry Pi based data-logger which will interface with SDI-12 sensor used in the environmental monitoring industry.

PROGRAMME: (Issue B, 14 March 2015)

1. Research Raspberry Pi operating system, programming languages, capabilities and I/O ports.
2. Research SDI-12 standards.
3. Design and build a hardware interface to implement the SDI-12 protocol. Test.
4. Specify a configuration file format and data storage plan.
5. Design and code software modules for the SDI-12 protocol interface and data logger.
6. Develop a basic HMI for the data logging system.
7. Investigate the use of FTP, SAMBA file server or Apache web server to facilitate network access to data files.
8. Submit an academic dissertation on the research.

As time permits

9. Develop code to configure SDI-12 devices.
10. Develop a graphical user interface for the system.
11. Add the data to an SQL database to allow remote access.

Appendix B: Project Management

Appendix B Contents

Appendix B.1: Risk Assessment

The project requires some hardware modification to an existing computer platform. The Raspberry Pi computer is a low voltage device (5V) and therefore consumers are able to run power cables. Typically the Pi is connected to 240 Vac through other third party devices. A low voltage circuit failure could result in other circuits failing and risk although low, of high voltage at the Raspberry Pi. Providing flexibility in design by allowing battery power will minimise the risk to consumers.

Table B.1: Risks in the development stage of the product

Risk Identification	**Risk Evaluation**	**Risk Control**
Short circuiting components on the Raspberry Pi computer when prototyping to the point of combustion of components or heating, with risk of burns, toxic fume inhalation causing physical health problems and damage to property.	Moderate	- Take precautions when using tools or test equipment not too short components. Use appropriate test leads i.e. small probes with small form for probing small components (separation). - Mount circuit board on a solid prototyping platform (design). - Ensure a clear work bench and area for working on electronics. - Ensure good circuit connections i.e. good jumper lead connections, minimise length of jumper leeds. - Test Pi GPIO voltages are as expected when wiring new circuits. - Disconnect power when not working on the platform. - Have a fume extractor on standby. - If smoke is observed disconnect equipment from power and allow time for smoke to dissipate. - Have a fire extinguisher and smoke alarm if work area contains fire hazards. - Work clear from flammable liquids like chemicals.
Burns to myself directly from soldering or fire from using a soldering iron with risk to others and property.	Low	- Use a clear work space. - Use a fume extractor. - Use a fire extinguisher. - Have a source of cold water to apply to the sight of burn.
Electric shock from using equipment powered from mains 240 Vac power	Low	- Make sure RCD are installed in premise. - Make sure equipment is earthed. - Wear adequate footwear with insulated soles. - Work on rubber or insulated mats to isolate the body from the floor. - Be aware of signs of wear in portable equipment, broken leads, sparks and unusual noise. - Have rescue action plan. Turn of power.

Table B.2: Risks to users of the Raspberry Pi-based SDI-12 logger

Risk Identification	**Risk Evaluation**	**Risk Control**
Mains voltage power adaptor device failure resulting in high voltage on Raspberry Pi circuit board. Risk of electrocution.	Low	- Consider using a plastic case. Install protection equipment such as RCDs.
Installing and powering equipment in field application is more dangerous than on a work bench. It is not easy to route cables so that they are safe and protected. If power cables are damaged there will be a risk of burns, toxic fume inhalation causing physical health problems and damage to property.	Moderate	- Only low voltage installations should be carried out by unqualified consumer. The customer needs to be informed of risks and best practices (Administration). - Low voltage power cables should be protected (separation). - Power supplies should be protected from its installation environment (separation). - Recommend purchasing battery powered sensors and radio interfaces for longer cabling requirements (administration).
Unintended connection of SDI-12 hardware interface may result in component overheating to point of fire and or heating. Fumes may not have an odour or be visible. Risk of burns, toxic fume inhalation causing physical health problems and damage to property.	Low	- Consumer versions of the SDI-12 interface hardware should physically fit to the Raspberry Pi platform in only the intended way (design). - Look at how to minimise issues in software (design).
Connecting GPIO outputs and ground to other systems inputs with a different ground potential causing heating or fire. Risk of burns, toxic fume inhalation causing physical health problems	Low	- GPIO outputs should be optically isolated from other systems (design).
Exposed circuit components on board is at risk of short circuit by conducting materials that come in contact to it resulting in faults causing fire and heating. Risk of burns, toxic fume inhalation causing physical health problems	Moderate	- A case designed to protect components (design).

Appendix B.2: Resource Requirements

The following equipment is needed for development of a prototype Raspberry Pi based, SDI-12 logger. All equipment is easily obtained from suppliers at the begging of the project.

- Raspberry Pi 2 Model B. This is the latest version of the model B which has an upgrade in RAM size and processor speed. This will be important for developing applications on the Raspberry Pi and running multiple processes at once. The system will be more responsive than all previous versions of the Raspberry Pi.
- SD card for Raspberry Pi memory, HDMI computer monitor, USB keyboard, USB mouse and USB WiFi for connecting the Raspberry Pi to the internet.
- Powered USB hub for powering USB devices connected to the Raspberry Pi.
- Prototype board to securely mount the Raspberry Pi. A single platform which includes a bread board and space to mount the Pi will allow effective and safe prototyping.
- 2.54 mm jumper cables terminated in a male and female 2.54mm header connection.
- Serial terminal for engineering and debugging the SDI-12 implementation on the Raspberry Pi from a Windows machine such as ‘Realterm’. ‘Minicom’ is installed on the Raspberry Pi to verify what is received on the serial terminal.
- USB to serial cable for debugging.
- Oscilloscope, multimeter, soldering iron, wire cutters and miscellaneous electronic components.
- SDI-12 sensors for testing.

If the Raspberry Pi was to become hard to source it would threaten the viability of the project. A second Raspberry Pi was purchased as a backup. Both the oscilloscope and USB to serial cable are considered required equipment for testing and debugging.

Appendix B.3: Project Timeline

Table B.3a: Timeline (Semester 1)

Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Starting	2 Mar	9 Mar	16 Mar	23 Mar	30 Mar	6 Apr	13 Apr	20 Apr	27 Apr	4 May	11 May	18 May	25 May	1 Jun	8 Jun	15 Jun	22 Jun	29 Jun	6 Jul	13 Jul
Tasks																				
Project Allocation	🟦 11-Mar	🟦																		
Project Specification		🟦 19-Mar	🟦																	
Research	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
Purchase Equipment	🟦	🟦	🟦	🟦																
Setup Raspberry Pi			🟦	🟦	🟦															
Become Familiar with Linux OS, and commandline				🟦	🟦	🟦	🟦													
Trial IDE's and experiment with programming language						🟦	🟦	🟦	🟦	🟦										
Find Programming Libraries for Raspberry Pi, interrupts and GPIO									🟦	🟦										
Preliminary Repot						🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦						
Preliminary Report Due													🟦 3-Jun	🟦						
Design and code SDI-12															🟦	🟦	🟦	🟦	🟦	🟦
Develop a script and test SDI-12 sensor or Debug and test with RealTherm terminal																				🟦
Design and build hardware															🟦	🟦	🟦	🟦	🟦	🟦
Specify configuration file format																				
Design main program																				
Draft Dissertation Due																				
Dissertation Submission																				
Milestones																				
Equipment Obtained				🟦																
Research & experiment with programming languages	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦											
Write test program in C++ that controls the GPIO								🟦	🟦											
Start writing progress report										🟦										

Table B.3b: Timeline (Semester 2)

Week	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Starting	20 Jul	27 Jul	3 Aug	10 Aug	17 Aug	24 Aug	31 Aug	7 Sep	14 Sep	21 Sep	28 Sep	5 Oct	12 Oct	19 Oct	26 Oct	2 Nov	9 Nov			
Tasks																				
Research	🟦	🟦	🟦	🟦																
Design and code SDI-12	🟦	🟦	🟦	🟦																
Develop a script and test SDI-12 sensor or	🟦	🟦	🟦	🟦																
Design and build hardware	🟦	🟦																		
Specify configuration file format				🟦	🟦	🟦														
Design main program				🟦	🟦	🟦	🟦													
Draft Dissertation Due								🟦 16-Sep	🟦											
Dissertation Submission														🟦 29-Oct	🟦					
Milestones																				
Hardware design complete					🟦	🟦	🟦													
Experiment with SDI12 library and testing	🟦	🟦	🟦	🟦	🟦	🟦	🟦													
Testing logger								🟦	🟦			🟦	🟦	🟦	🟦					
Writing dissertation						🟦	🟦	🟦	🟦			🟦	🟦	🟦	🟦					

Appendix C: SDI12 C++ Library

This appendix gives the modified Arduino SDI-12 C++ library originally authored by Kevin Smith (available at https://github.com/StroudCenter/Arduino-SDI-12) used in the Raspberry Pi SDI-12 logger program which is listed in appendix D. The Author of the code and comments is acknowledge within the comments section of both the header 'SDI12.h' and the source code 'SDI12.cpp'.

Appendix C Contents

Appendix C.1: SDI12 Library Header File (SDI12.h)

```
/* =============================== Raspberry SDI-12 ========================== (James Coppock)

This SDI-12 library was originally authored by Kevin M. Smith for an Arduino based logger. It
has been modified by James Coppock for a Raspberry Pi based logger which according to the
attribution and licences section below is allowed. New SDI12 member functions have been written
for this Raspberry Pi implementation including:
        - CRCheck()
        - LFCheck()
        - advanceBufferHead()
        - overflowStatus()
        - parityErrorStatus()

Modifications were done to the following member functions.
        - setState()    - setstate() defines the state of four pins in this implementation. The
                          original implementation only defined the state of one data pin which
                          is possible because the Arduino digital pins are 5 volt which is
                          compatible with SDI-12.
        - receiveChar() - Added a parity check
        - writeChar()   - Included an algorithm to add a parity bit.

Some of the section heading comments are the original authors and some are written by myself. I
have written the name of the author of the section heading comment in brackets at the top right
hand corner of major section heading. If the section heading comments are both my work and
Kevin’s my words and those of Kevin's are enclosed in brackets with either initials 'JMC:' or
'KMS:' at the beginning.

Where in function comments are my own or parts of the SDI12 class have been modified or new
functions included they are identified through out using my initials. Program modifications are
followed by a comment with my initial (JMC) at the beginning of the comment.


/* ================================ Arduino SDI-12 =============================( Kevin Smith)

Arduino library for SDI-12 communications to a wide variety of environmental sensors. This
library provides a general software solution, without requiring any additional hardware.

================================ Attribution & License ================================

Copyright (C) 2013 Stroud Water Research Centre Available at
https://github.com/StroudCenter/Arduino-SDI-12

Authored initially in August 2013 by:

        Kevin M. Smith (http://ethosengineering.org)
        Inquiries: SDI12@ethosengineering.org

based on the SoftwareSerial library (formerly NewSoftSerial), authored by:
        ladyada (http://ladyada.net)
        Mikal Hart (http://www.arduiniana.org)
        Paul Stoffregen (http://www.pjrc.com)
        Garrett Mace (http://www.macetech.com)
        Brett Hagman (http://www.roguerobotics.com/)

This library is free software; you can redistribute it and/or modify it under the terms of the
GNU Lesser General Public License as published by the Free Software Foundation; either version
2.1 of the License, or (at your option) any later version.

This library 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 Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this
library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor,
Boston, MA  02110-1301  USA
```

```
#ifndef SDI12_h
#define SDI12_h
                                 // Required Libraries
#include <string>                // Use with C++ style strings (std string)
#include <errno.h>
#include <stdlib.h>              // Gives the system function for system calls. system("....");
#include <inttypes.h>            // libraries for uint8_t (unsigned character 8 bits), int8_t
(Signed character 8 bits), int16_t (short int (2 bytes)), int32_t (4 bytes) int64 (long int (8
bytes))
#include <iostream>              // uses cin and cout for input and output

// Include the wiringPi library Authored by Gordon Henderson. The library give some simple
functions for controlling the setting the state of the Raspberry Pi pins, writing a HIGH or LOW
to the pin or reading digital voltage.
#include <wiringPi.h>

class SDI12 {
private:
  static void setState(uint8_t state);          // set the state of the SDI12 objects
  void wakeSensors();                           // Used to wake up all sensors on the SDI12 bus
  void writeChar(uint8_t out);                  // sends a char out on the data line
  static inline void receiveChar();             // used by the ISR to grab a char from data line

public:
  SDI12(uint8_t txEnable, uint8_t txDataPin, uint8_t rxEnable, uint8_t rxDataPin);        //
constructor
  ~SDI12();                                      // destructor
  void begin();                                  // enable SDI-12 object
  static void end();                             // disable SDI-12 object
  void forceHold();                              // sets line state to HOLDING
  void sendCommand(std::string cmd);             // sends the String cmd out on the data line
  bool overflowStatus();                         // (JMC: returns the overflow status)
  bool parityErrorStatus();                      // (JMC: returns parity error status)
  int available();                               // returns the number of bytes available in
buffer
  bool LFCheck();                                // (JMC: Checks the last character in the
buffer is a <LF>)
  bool CRCheck();                                // (JMC: Checks the last character in the
buffer is a <CR>)
  int peek();                                    // reveals next byte in buffer without
consuming)
  void flush();                                  // resets the circular buffer head and tail,
resets the Overflow and parity error status
  int read();                                    // returns next byte in the buffer (consumes)
  void advanceBufHead(int advance);              // (JMC: advance the buffer head)
  static inline void handleInterrupt();          // intermediary ISR function

};

#endif
```

Appendix C.2: SDI12.cpp

```
* =================================== Raspberry SDI-12 ======================= (James Coppock)

This SDI-12 library was originally authored by Kevin M. Smith for an Arduino based logger. It
has been modified by James Coppock for a Raspberry Pi based logger which according to the
attribution and licences section below is allowed. New SDI12 member functions have been written
for this Raspberry Pi implementation including:
        - CRCheck()
        - LFCheck()
        - advanceBufferHead()
        - overflowStatus()
        - parityErrorStatus()

Modifications were done to the following member functions.
        - setState()     - setstate defines the state of four pins in this implementation. The
                           original implementation only defined the state of one data pin which
                           is possible because the Arduino digital pins are 5 volt which is
                           compatible with SDI-12.
        - receiveChar() - Added a parity check
        - writeChar()    - Included an algorithm to add a parity bit.

Some of the section heading comments are the original authors and some are written by myself. I
have written the name of the author of the section heading comment in brackets at the top right
hand corner of major section heading. If the section heading comments are both my work and
Kevin’s my words and those of Kevin's are enclosed in brackets with either initials 'JMC:' or
'KMS:' at the beginning.

Where in function comments are my own or parts of the SDI12 class have been modified or new
functions included they are identified through out using my initials. Program modifications are
followed by a comment with my initial (JMC) at the beginning of the comment.

*/

/*================================= Arduino SDI-12 ============================== (Kevin Smith)

Arduino library for SDI-12 communications to a wide variety of environmental sensors. This
library provides a general software solution, without requiring any additional hardware.

============================ Original Attribution & License ===================== (Kevin Smith)

Copyright (C) 2013 Stroud Water Research Centre Available at
https://github.com/StroudCenter/Arduino-SDI-12

Authored initially in August 2013 by:

        Kevin M. Smith (http://ethosengineering.org)
        Inquiries: SDI12@ethosengineering.org

based on the SoftwareSerial library (formerly NewSoftSerial), authored by:
        ladyada (http://ladyada.net)
        Mikal Hart (http://www.arduiniana.org)
        Paul Stoffregen (http://www.pjrc.com)
        Garrett Mace (http://www.macetech.com)
        Brett Hagman (http://www.roguerobotics.com/)

This library is free software; you can redistribute it and/or modify it under the terms of the
GNU Lesser General Public License as published by the Free Software Foundation; either version
2.1 of the License, or (at your option) any later version.

This library 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 Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this
library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor,
Boston, MA  02110-1301 USA
```

```
==================================== Code Organization ========================= (Kevin Smith)

0. Includes, Defines, & Variable Declarations
1. Buffer Setup
2. Data Line States, Overview of Interrupts
3. Constructor, Destructor, SDI12.begin(), and SDI12.end()
4. Waking up, and talking to, the sensors.
5. Reading from the SDI-12 object. available(), peek(), read(), flush()
6. Interrupt Service Routine (getting the data into the buffer)
*/



/* ===== 0. Includes, Defines, and Variable Declarations ======= (Kevin Smith and James Coppock)
(KMS:
          0.1 - Include the header file for this library.
          0.2 - defines the size of the buffer
          0.3 - defines value for DISABLED state
          0.4 - defines value for ENABLED state
          0.5 - defines value for DISABLED state
          0.6 - defines value for TRANSMITTING state
          0.7 - defines value for LISTENING state
)
(JMC:     0.8 - defines value for ENABLEINTERRUPT state.)

(KMS:     0.9 - Specifies the delay for transmitting bits. (JMC: 1200 Baud equates to 833us but
                with system calls the actual time was measured using an oscilloscope to be
                805us.)
)
(JMC: 0.10 to 0.13 are new reference variables.
          0.10 - a reference to the pin that connects to the SN74HCT240 output enable pin. This
                pin puts the TX data pin on the output of the SN74HCT240 in a high impedance
                state.)
          0.11 - a reference to the TX data pin. This pin is connected to an input of the
                SN74HCT240.)
          0.12 - a reference to the pin that connects to one of the SN74HCT240 output enable
                pin. This pin puts the output of the SN74HCT240 connected to the RX data pin of
                the Pi in a high impedance state.)
          0.13 - a reference to the RX data pin. This pin is connected to an output of the
                SN74HCT240.)
)
(KMS:     0.14 - holds the buffer overflow status.)
(JMC:     0.15 - a new reference variable which holds the parity error status.
)
*/


#include </home/pi/Desktop/SDI12.h>      // 0.1 header file for this library
#define _BUFFER_SIZE 75                  // 0.2 - max buffer size
#define DISABLED 0                       // 0.3 value for DISABLED state
#define ENABLED 1                        // 0.4 value for ENABLED state
#define HOLDING 2                        // 0.5 value for DISABLED state
#define TRANSMITTING 3                   // 0.6 value for TRANSMITTING state
#define LISTENING 4                      // 0.7 value for LISTENING state
#define INTERRUPTENABLE 5                // (JMC: 0.8 value for ENABLEINTERRUPT state)
#define SPACING 805                      // 0.9 bit timing in microseconds

uint8_t _txEnable;               // (JMC: 0.10 reference to the pin that connects to one of
                                    the SN74HCT240 output enable pins)
uint8_t _txDataPin;              // (JMC: 0.11 reference to the tx data pin)
uint8_t _rxEnable;               // (JMC: 0.12 reference to the pin that connects to one of the
                                   SN74HCT240 output enable pins)
uint8_t _rxDataPin;              // (JMC: 0.13 reference to the rx data pin)
bool _bufferOverflow;            // 0.14 buffer overflow status
bool _parityError;               // (JMC: 0.15 parity error status)
```

```
/* ================================ 1. Buffer Setup ============================ ( Kevin Smith)

The buffer holds the ascii characters from the SDI-12 bus. Characters are read into the buffer
when an interrupt is received on the data line. The buffer uses a circular implementation with
pointers to both the head and the tail. The circular buffer is defined with the size of the
buffer and two pointers;
        - One to the start of data
        - One to end of data
When a pointer reaches the end of the buffer it jumps back to the start.

1.1 - Define a maximum buffer size (in number of characters).
1.2 - Declare a character array of the specified size.
1.3 - Index to buffer head. (JMC: Buffer head is the index to first character in. The index is
      declared as uint8_t type which is an unsigned 8-bit integer, can map from 0-255.)
1.4 - Index to buffer tail. (JMC: Buffer tail is the index to 1 position advanced of the last
      character unless empty.)
*/

// See section 0.2 above.                 // 1.1 - max buffer size
char _rxBuffer[_BUFFER_SIZE];             // 1.2 - declare an array (buffer for incoming ascii
characters)
uint8_t _rxBufferHead = 0;                // 1.3 - index of buff head
uint8_t _rxBufferTail = 0;                // 1.4 - index of buff tail


/* ==================================== 2. Data Line States ================== ( James Coppock)

The original library specified 4 states. The original library used 1 single digital I/O pin of
an Arduino. The Arduino pins are 5 volts and can supply 20 mA of current. The voltage of the
Arduino pin is compatible with SDI-12 voltage and 20 mA is plenty to drive at least 10 sensor.
The Raspberry Pi pins are 3.3 volt and can supply 16 mA from a single pin safely (the pins are
not current limited). The data pin(s) must be connected to some hardware to interface the
Raspberry Pi to the SDI-12 bus. The best way to do this is by having two data pins, one TX data
pin and one RX data pin and connecting to a SN74HCT240 inverting tristate buffer line driver.

All communications are initiated by the Raspberry Pi. The state of the line is set to 1 of 5
states.

                    RXDATAPIN        RXDATAPIN    RXENABLE     TXDATAPIN     TXENABLE
                   (BCM 22)         (BCM 22)     (BCM 27)     (BCM 17)     (BCM 4)
                    Input            Input        Output       Output        Output
State               Interrupt Mode   LEVEL        Level        Level         Level
                   (240 1Y1)       (240 1Y1)     (240 1OE)    (240 2A1)     (240 2OE)
HOLDING             Falling          pulup        HIGH         HIGH          LOW
TRANSMITTING        Falling          pulup        HIGH         Vary          LOW
LISTENING           Falling          pulup        LOW          Dont Care     HIGH
DISABLED           Disable          pulup        HIGH          Dont care    HIGH
INTERRUPTENABLE     Enable Fall      Pulup        HIGH         HIGH          LOW


* NOTE: OE HIGH means output of SN74HCT240 in high impedance state.

------------------------------------------| Sequencing |------------------------------------

INTERRUPTENABLE --> TRANSMITTING --> LISTENING --> TRANSMITTING --> LISTENING --> ..... -->
DISABLED

-----------------------------------| Function Descriptions |-----------------------------------

2.1 - A private function, sets the state of 4 pins that connect to the tri state buffer and line
driver with separate output enable pins. The 5 states are given in the table above which are
HOLDING, TRANSMITTING, LISTENING, DISABLED and INTERRUPTENABLE. 4 of the five states were
defined in original code but all the code of this member function is changed. The functions used
to write a HIGH or LOW on any pin, and the interrupt enable and disable are from the wiringPi
library.

2.2 - A public function which forces a "HOLDING" state. This function is called after a failed
communication due to noise or to place line into a low impedance state before initiating
communication with a sensor.
```

```
// 2.1 - sets the state of the SDI-12 object. (JMC: All setState() function code has been
modified to control SN74HCT240 using wiringPi libraries as mentioned in section 2 comments
above)
void SDI12::setState(uint8_t state){
  if(state == HOLDING){                                        // If HOLDING
    //std::cout << "SetState = HOLDING" << "\n";
    digitalWrite(_rxEnable, HIGH);       // State of 240 output 1 in high impedance
    digitalWrite(_txDataPin, HIGH);      // Set TX pin HIGH level (txDataPin = BCM17)
    digitalWrite(_txEnable, LOW);        // Set State of 240 output 2 'driving' state
    return;
  }
  if(state == TRANSMITTING){                          // If Transmitting
    //std::cout << "SetState = TRANSMITTING" << "\n";
    digitalWrite(_rxEnable, HIGH);       // State of 240 output 1 high impedance
    digitalWrite(_txEnable, LOW);        // State of 240 output 2 is driving state
    return;
  }
  if(state == LISTENING) { // If LISTENING - All interrupts enabled also set the pullup resistor
    //std::cout << "SetState = LISTENING" << "\n";
    digitalWrite(_txEnable, HIGH);       // State of 240 output 2 (TX output) is driving state
    digitalWrite(_rxEnable, LOW);        // State of 240 output 1 (RX input) is high impedance
    return;
  }
  if(state == DISABLED) {        // If state==DISABLED.  Pin interrupt disabled.
    //std::cout << "SetState = DISABLED" << "\n";
    // Only necessary to disable if using ISR routine.
    system ("gpio edge 22 none");        // Disable rising edge interrupt detection on RXDATAPIN
    digitalWrite(_rxEnable, HIGH);       // State of 240 output 1 (RX input) is high impedance
    digitalWrite(_txEnable, HIGH);       // State of 240 output 2 (TX output) is driving state
        return;
  }
  if(state == INTERRUPTENABLE) {         // If state==INTERRUPT.  Enables pin interrupt.
    //std::cout << "SetState = INTERRUPTENABLE"<< "\n";//
    pullUpDnControl(_rxDataPin, PUD_UP); // Set RX Pin with pull down resistors enabled
    system ("gpio edge 22 falling");     // Enable rising edge interrupt detection on RXDATAPIN
    pullUpDnControl(_rxDataPin, PUD_DOWN);// Triggers the first interrupt which has a bug and
triggers two. Both these need to be ignored.
    pullUpDnControl(_rxDataPin, PUD_UP);// Triggers the first interrupt which has a bug and
triggers two. Both these need to be ignored.
    delay(1);
    digitalWrite(_rxEnable, HIGH);       // State of 240 output 1 in high impedance
    digitalWrite(_txDataPin, HIGH);      // Set TX pin HIGH level (txDataPin = BCM17)
    digitalWrite(_txEnable, LOW);        // Set State of 240 output 2 'driving' state
    return;
  }
  else {                                 // Error message due to unexpected value.
    std::cout << "Error: SetState = unknown check script for mistake" << "\n";//
    //std::cout << state << "\n";
    std::cout << unsigned(state) << "\n";
  }
}

// 2.2 - forces a HOLDING state.
void SDI12::forceHold(){
  //std::cout << "ForceHold() called\n";
  setState(HOLDING);
}
```

```
================================= 3. Constructor, Destructor, SDI12.begin(), SDI12.end(),
parityErrorStatus(), and overflowStatus()====================== (Kevin Smith and James Coppock)


(KMS: 3.1 - The constructor requires a four parameter, which are the four pins to be used for
            the data line. When the constructor is called it resets the buffer overflow status
            to FALSE (JMC: and resets the parity overflow status to false and assigns the pin
            numbers txEnable, txDataPin, rxEnable and rxDataPin to private variables
            "_txEnable", "_txDataPin", "_rxEnable" and "_rxDataPin".)
)

(KMS: 3.2 - When the destructor is called, it's main task is to disable any interrupts that had
            been previously assigned to the pin, so that the pin will behave as expected when
            used for other purposes. This is achieved by putting the SDI-12 object in the
            DISABLED state.

          3.3 - begin() - public function called to begin the functionality of the
                SDI-12 object. It has no parameters as the SDI-12 protocol is fully
                specified (e.g. the baud rate is set).

          3.4 - end() - public function called to temporarily cease all functionality
                of the SDI-12 object. It is not as harsh as destroying the object with the
                destructor, as it will maintain the memory buffer.
)
(JMC: 3.5 - parityErrorStatur() - new public function called to return the parity error status.
)

(KMS: 3.6 - overflowStatus() - public function called to return the overflow error status
)

//       3.1      Constructor (JMC: Modified function parameters)
SDI12::SDI12(uint8_t txEnable, uint8_t txDataPin, uint8_t rxEnable, uint8_t rxDataPin) {
  //std::cout << "Constructor() Called\n";
  _rxBufferHead = _rxBufferTail = 0;      // initialise buffer pointer
  _bufferOverflow = false;                // initialise buffer overflow
  _parityError = false;                   // (JMC: initialise parity error)
  _txEnable = txEnable;                   // (JMC: assign pin number to private variables)
  _txDataPin = txDataPin;                 // (JMC: assign pin number to private variables)
  _rxEnable = rxEnable;                   // (JMC: assign pin number to private variables)
  _rxDataPin = rxDataPin;                 // (JMC: assign pin number to private variables)
}

//       3.2      Destructor
SDI12::~SDI12(){
  //std::cout << "Destructor () called\n";
  setState(DISABLED);
}

//  3.3 Begin - public function sets rising edge interrupt on RX datapin.
void SDI12::begin() {
  //std::cout << "begin() Called\n";
  setState(INTERRUPTENABLE);
}

//  3.4 End - public function
void SDI12::end() {
  //std::cout << "end() Called\n";
   SDI12::setState(DISABLED);
}

// (JMC: 3.5 - new public function returns the parity error status).
bool SDI12::parityErrorStatus(){
  //std::cout << "parityError() called\n";
  return(_parityError);
}

//3.6 - public function returns the overflow status.
bool SDI12::overflowStatus() {
  //std::cout << "overflowStatus() called\n";
  return(_bufferOverflow);
}
```

```
/* ========= 4. Waking up, and talking to, the sensors. =========(Kevin Smith and James Coppock)


------------------------------------|  Function Descriptions |----------------------------------

(JMC: 4.1 - wakeSensors() - original function (private) that is called by the public sendCommand
            function. wakeSensors() will wake the sensors on the SDI-12 bus by placing spacing
            (HIGH voltage level) for a minimum of 12 milliseconds (no upper limit specified in
            standard). This is followed by a marking (logic LOW) for at least 8.33 ms (the upper
            limit to marking is about 90 ms). As the SDI-12 sensors are permitted to sleep after
            100 ms of marking. Allowing some extended time on the minimum, the break is held for
            14.161 ms and the marking for 10 ms. The state is initially set to the transmitting
            state.
)

(JMC: 4.2 - writeChar(uint8_t out) - slightly modified private function, that outputs a single
            ASCII character to the SDI-12 bus. Each 10 bit frame that is sent out has 7 data
            bits, (LSB first) 1 start bit, 1 parity bit (even parity), and 1 stop bit. The SDI-
            12 protocol uses negative logic. An example a transmission of character 'a' is
            shown below. The binary representation of ASCII 'a' is 110 0001.

 The SDI12 line voltage for char 'a' is;
            _   _ _ _ _         _
           |s|d|d d d d|d d e f|
          _| |_|       |_ _ _ _|

 d = 7 data bits
 s = 1 start bit
 e = 1 parity bit
 f = 1 stop bit
)

There are four steps to the transmission

(JMC: 4.2.1 -The original code used an a function from the parity.h header to calculate the
           parity. I have used an alternate algorithm. The algorithm calculates the number of
           1's in the 8 bit frame with one parity bit and seven data bits. The algorithm returns
           _evenParityBit is either 0 (even number of 1's) or 1 (odd number of 1's). The frame
           should have an even number of ones for even parity.

          The code merges the first 4 bits with the last 4 bits using an XOR operation. As can
          be seen below the parity of two bits is computed with an XOR operation.
          (0 XOR 0) -> 0
          (0 XOR 1) -> 1
          (1 XOR 0) -> 1
          (1 XOR 1) -> 0
          Now with four bits we are left with 16 possible values for _evenParityBit. Shifting
          0x6996 to the right by _evenParityBit number of times leaves the relevant bit in bit
          position 0. A 0 for even and a 1 for odd parity.
)

(JMC: 4.2.2 - slightly modified code to send the start bit. The start bit is a 1 on the SDI12
           data line. The original code sent HIGH however writing a LOW to the TX data pin will
           cause the SN74HCT240 to output a HIGH, so a LOW is written to the TX data pin for 820
           us.
)
(KMS: 4.2.3 - (JMC: Slightly modified code switches the HIGH for LOW and LOW for HIGH)
           Send the payload (the 7 character bits and the parity bit) least significant bit
           first. This is accomplished bitwise AND operations on a moving mask (00000001) -->
           (00000010) --> (00000100)... and so on. This functionality makes use of the '<<='
           operator which stores the result of the bit-shift back into the left hand side.

           If the result of (out & mask) determines whether a 1 or 0 should be sent.

           if(out & mask){
                 digitalWrite(_txDataPin, HIGH);
           }
           else{
         digitalWrite(_txDataPin, LOW);
           }

    4.2.4 - Slightly modified code to send the stop bit. The stop bit is always a '0', so we
```

```
           simply write the dataPin HIGH for 820 microseconds.

    4.3 - sendCommand(String cmd) - original public function that sends out a String byte by
          byte the data line.
)
*/

// 4.1 - private function that wakes up the entire sensor bus.
void SDI12::wakeSensors(){
  setState(TRANSMITTING);
  digitalWrite(_txDataPin, LOW);
  delayMicroseconds(14161);
  digitalWrite(_txDataPin, HIGH);
  delayMicroseconds(10000);
}

// 4.2 - private functionp that writes a character out on the data line (JMC: function modified)
void SDI12::writeChar(uint8_t out)      {
  //std::cout << "Next Character out writeChar\n";
  //std::cout << out << "\n";            // 4.2.1 - 1 byte with 7 bit ASCII character and with
even parity bit in MSB
  uint8_t _evenParityBit;                // (JMC: code written to calculate the parity bit)
  _evenParityBit = out;                  // (JMC: code written to calculate the parity bit)
  _evenParityBit ^= out >> 4;            // (JMC: code written to calculate the parity bit)
  _evenParityBit &= 0x0F;                // (JMC: code written to calculate the parity bit)
  _evenParityBit = ((0x6996 >> _evenParityBit) & 1);// (JMC: code written to calculate the
parity bit)
  out |= (_evenParityBit<<7);

  digitalWrite(_txDataPin, LOW);                          // 4.2.2 - start bit
  delayMicroseconds(820);

  for (uint8_t mask = 0x01; mask; mask<<=1){              // 4.2.3 - send payload
    if(out & mask){
      digitalWrite(_txDataPin, HIGH);
    }
    else{
      digitalWrite(_txDataPin, LOW);
    }
    delayMicroseconds(SPACING);
  }
  digitalWrite(_txDataPin, HIGH);                         // 4.2.4 - stop bit
  delayMicroseconds(820);
}

//      4.3     - Public function that sends out the characters of the String cmd, one by one
void SDI12::sendCommand(std::string cmd){
  //std::cout << "sendCommand Called\n";
  wakeSensors();                                          // wake up sensors
  for (unsigned i = 0; i < cmd.length(); i++){
    //std::cout << "Next Character out - sendCommand() \n";
    //std::cout << cmd[i] << "\n";       // outputs variable (Note cout << (unsigned char))
    writeChar(cmd[i]);                           // write each characters
  }
  setState(LISTENING);                           // listen for reply
}
```

```
/* ============= 5. Reading from the SDI-12 object.  ============(Kevin Smith and James Coppock)


(KMS: 5.1 - available() - (JMC: original public function that) returns the number of characters
                available in the buffer. To understand how:
                _rxBufferTail + _BUFFER_SIZE - _rxBufferHead) % _BUFFER_SIZE;
                accomplishes this task, we will use a few examples.

                To start take the buffer below that has _BUFFER_SIZE = 10. The message "abc" has
                been wrapped around (circular buffer).

                _rxBufferTail = 1 // points to the '-' after c
                _rxBufferHead = 8 // points to 'a'

                        [ c ] [ - ] [ - ] [ - ] [ - ] [ - ] [ - ] [ - ]  [ a ] [ b ]

                The number of available characters is (1 + 10 - 8) % 10 = 3

                The '%' or modulo operator finds the remainder of division of one number by
                another. In integer arithmetic 3 / 10 = 0, but has a remainder of 3. We can only
                get the remainder by using the the modulo '%'. 3 % 10 = 3. This next case
                demonstrates more clearly why the modulo is used.
                rxBufferTail = 4 // points to the '-' after c
                _rxBufferHead = 1 // points to 'a'

                [ a ] [ b ] [ c ] [ - ] [ - ] [ - ] [ - ] [ - ]  [ - ] [ - ]
                The number of available characters is (4 + 10 - 1) % 10 = 3

                If we did not use the modulo we would get either ( 4 + 10 - 1 ) = 13 characters
                or ( 4 + 10 - 1 ) / 10 = 1 character. Obviously neither is correct.

                If there has been a buffer overflow, available() will return -1.
)
(JMC: 5.2 - new public function that checks the last character in the buffer is a <LF> without
                consuming. The buffer tail is the index to 1 position advanced of the last
                character in thus the last char is _rxBufferTail-1 LF = 0000 1010  = 10(dec)
)
(JMC: 5.3 - new public function that checks the second last character in the buffer is a <CR>
                without consuming. The buffer tail is the index to 1 position advanced of the
                last character in thus the second last char is:
                _rxBufferTail-2 CR = 0000 1101  = 13(dec)
)
(KMS: 5.4 - peek() - (JMC: original public function that) allows the user to look at the
                character that is at the head of the buffer. Unlike read() it does not consume
                the character (i.e. the  index addressed by _rxBufferHead is not changed).
                peek() returns -1 if there are no characters to show.

      5.5 - flush() is a modified public function that clears the buffers contents by setting
                the index for both buffer head and tail back to zero. (JMC: new code also resets
                the status of the buffer overflow and parity error variables.)

      5.6 - read() returns the character at the current head in the buffer after incrementing
                the index of the buffer head. This action 'consumes' the character, meaning it
                cannot be read from the buffer again. If you would rather see the character, but
                leave the index to head intact, you should use peek();
)
(JMC: 5.7 - advanceBufHead() - new public function that advances the buffer head. This function
                is used if only a certain part of the response from the sensor is needed. It
                saves reading all characters into the program and then discarding them.
)
*/

// 5.1 - public function that reveals the number of characters available in the buffer -
int SDI12::available() {
  //std::cout << "available() called\n";
  if(_bufferOverflow) return -1;
  return (_rxBufferTail + _BUFFER_SIZE - _rxBufferHead) % _BUFFER_SIZE;
}

// (JMC: 5.2 - new public function that checks the last character in the buffer is a <LF>
without consuming)
bool SDI12::LFCheck() {
  //std::cout << "LFCheck() called\n";
  if (_rxBufferHead == _rxBufferTail) return false;       // Empty buffer? If yes, 0
  int LF = _rxBuffer[_rxBufferTail-1];
  // Otherwise, read from "tail" (last character in)
  if (LF == 10) {
    //std::cout << "Last character is a linefeed <LF>! \n";
    return true;
  }
  return false;
}

// (JMC: 5.3 - new public function that checks the second last character in the buffer is a <CR>
without consuming)
bool SDI12::CRCheck() {
  //std::cout << "CRCheck() called\n";
  if (_rxBufferHead == _rxBufferTail) return false;       // Empty buffer? If yes, 0
    int CR = _rxBuffer[_rxBufferTail-2];
    if (CR == 13) {               // Otherwise, check the second last character in buffer)
      return true;
    }
  return false;
  }
}

// 5.4 - public function that reveals the next character in the buffer without consuming
int SDI12::peek() {
  if (_rxBufferHead == _rxBufferTail) return -1;  // Empty buffer? If yes, -1
  return _rxBuffer[_rxBufferHead];                // Otherwise, read from "head"
}

// 5.5 - a public function that clears the buffer contents, resets the status of the buffer
overflow and parrity error variables.
void SDI12::flush() {
  //std::cout << "flush() called\n";
  _rxBufferHead = _rxBufferTail = 0;
  _bufferOverflow = false;
  _parityError = false;
}

// 5.6 - reads in the next character from the buffer and moves the index ahead. (JMC: This is
FIFO opperation)
int SDI12::read() {
  _bufferOverflow = false;
  //reading makes room in the buffer
  if (_rxBufferHead == _rxBufferTail) return -1;         // Empty buffer? If yes, -1
  uint8_t nextChar = _rxBuffer[_rxBufferHead];           // Otherwise, grab char at head
  _rxBufferHead = (_rxBufferHead + 1) % _BUFFER_SIZE;    // increment head. modulo will reset
the _rxBufferHead to 0 when _rxBufferHead = bufferSize - 1
  return nextChar;                                       // return the char
}

// (JMC: 5.7 - new public function that advances the buffer head)
void SDI12::advanceBufHead(int advance) {
  //std::cout << "advanceBufHead() called \n";
  //std::cout << "initial buffer head position" << (int)_rxBufferHead << "\n";
  _rxBufferHead = _rxBufferHead + advance;
  //std::cout << "new buffer head position" << (int)_rxBufferHead << "\n";
 }
```

```
/* ================== 6. Interrupt Service Routine ============= ( James Coppock & Kevin Smith )

(JMC:
        The original receiveChar() function did not include a parity check. I have modified the
        code to include a parity error check. Most of the timing delays were changed to decrease
        the chance of missing a bit when the operating system de-scheduled the thread.

        wiringPi library functions used:
        int digitalRead(int pin) - function returns the value read on the given pin. It will be
        HIGH or LOW (1 or 0) depending on the logic level at the pin (Gordons Projects 2015).
)

(KMS:
        We have received an interrupt signal, what should we do?

         6.1 - function passes of responsibility to the receiveChar() function.

         6.2 - This function quickly reads a new character from the data line in to the buffer.
                It takes place over a series of key steps.

        6.2.1 - Check for the start bit. If it is not there, interrupt may be from interference
                or an interrupt we are not interested in, so return.

        6.2.2 - Make space in memory for the new character "newChar".

        6.2.3 - Wait half of a SPACING to help centre on the next bit. It will not actually be
                centred, or even approximately so until delayMicroseconds(SPACING) is called
                again.

        6.2.4 - For each of the 8 bits in the payload, read whether or not the line state is
                HIGH or LOW. We use a moving mask here, as was previously demonstrated in the
                writeByte() function.

                The loop runs from i=0x1 (hexadecimal notation for 00000001) to i<0x80
                (hexadecimal notation for 10000000). So the loop effectively uses the
                masks following masks: 00000001
                00000010
                00000100
                00001000
                00010000
                00100000
                01000000 and their inverses.

                Here we use an if / else structure that helps to balance the time it takes to
                either a HIGH vs a LOW, and helps maintain a constant timing.

        6.2.5 - Skip the stop bit.

(JMC:
         6.2.6 - The original code skipped the parity bit with a delay of 830 microseconds. Due
                to the number of parity errors that would come up the parity check was needed.

                 The algorithm calculates the number of 1's in the 8 bit frame with one parity
                bit and seven data bits. The algorithm returns _evenParityBit is either 0 (even
                number of 1's) or 1 (odd number of 1's). The frame should have an even number of
                ones for even parity.

                 The code merges the first 4 bits with the last 4 bits using an XOR
                 operation. As can be seen below the parity of two bits is computed with
                 an XOR operation.
                 (0 XOR 0) -> 0
                 (0 XOR 1) -> 1
                 (1 XOR 0) -> 1
                 (1 XOR 1) -> 0
                 Now with four bits we are left with 16 possible values for _evenParityBit
                 Shifting 0x6996 to the right by _evenParityBit number of times leaves the
                relevant bit in bit position 0. 6996 (Hex) = 0110 1001 1001 0110 (bin) which
                gives the 16 possibilities for the parity. A 0 for even and a 1 for odd parity

                 If a parity error is picked up parityError status is set to true and the state
                is set to disabled. The interrupts will be disabled also.
)
```

```
(KMS:
         6.2.7 - Check for an overflow. Check if advancing the index to most recent buffer entry
                (tail) will make it have the same index as the head (in a circular fashion). If
                there is an overflow a character will not be stored and hence will not overwrite
                buffer head

         6.2.8 - Save the byte into the buffer if there has not been an overflow, and then
                advance the tail index.
)

// 6.1 - public static function that passes off responsibility for an interrupt to the
receiveChar() function.
inline void SDI12::handleInterrupt(){
  if (_parityError == true) {
    std::cout << "handleInterrupt() error: parity error is true: \n";
        return;
  }
  receiveChar();
}

// 6.2 - private function that reads a new character into the buffer (JMC: function modified
// to do parity error check on each received).
inline void SDI12::receiveChar() {
  //std::cout << "receiveChar() called \n";

  if (digitalRead(_rxDataPin)==0) {               // 6.2.1 - Is the start bit LOW? a HIGH
indicates a false trigger of interrupt

        uint8_t newChar = 0;                      // 6.2.2 - Declare and initialise variable for
char.

    delayMicroseconds(20);                        // 6.2.3 - sets a small delay period after the
falling edge of the start bit was detected

    for (uint16_t i=0x1; i<=0x80; i <<= 1) {      // 6.2.4 - read the 7 data bits (for i = 1 to
0100 0000 (<<= bitshift assignment))
      delayMicroseconds(800);    // (800)         // Delay 800 us. This seems to work better than
a full symbol period of 830 us.
      uint8_t noti = ~i;                          // ~ Bitwise NOT operator
      if (!digitalRead(_rxDataPin)) {             // If pin level is LOW (NOTE ! is Logical NOT
operator)
        std::cout << "Pin level LOW: " << "\n";
        newChar &= noti;
         }
      else {                                      // Else pin level is HIGH
         std::cout << "Pin level HIGH: " << "\n";
         newChar |= i;                            // |= Bitwise inclusive OR assignment operator
        }
    }

    uint8_t newChar2 = newChar;
    newChar2 &= 0x7F;

        delayMicroseconds(650);  //(650)          // 6.2.5 - Skip the stop bit.

    // New code. Experiment with delay above.
    if (digitalRead(_rxDataPin)==0) {             // JMC: 6.2.5 - Is the stop bit LOW? a LOW
indicates an incorrect stop bit (inverted logic)
    std::cout << "receiveChar() Incorrect stop bit: - parityError set to true and interrupt
disabled \n";
    _parityError = true;                          // JMC:
    SDI12::end();                                 // JMC: Disable interrupt
return;                                           // JMC:
}


    // (JMC: 6.2.6 - Check for parity error.
    uint8_t _evenOrOdd;                           // JMC
    _evenOrOdd = newChar;                         // JMC
    _evenOrOdd ^= newChar >> 4;                   // JMC
    _evenOrOdd &= 0x0F;                           // JMC
    _evenOrOdd = ((0x6996 >> _evenOrOdd) & 1);    // JMC
    std::cout << "ReceiveChar() calculated parity of newChar (0 = even, 1 = odd): " <<
unsigned(_evenOrOdd) << "\n";

  if (_evenOrOdd == 1) {                        // (JMC: Check for parity error)
    std::cout << "receiveChar() Parity error: - parityError set to true - check parityError()
\n";
    _parityError = true;
    SDI12::end();
    return;
  }

    newChar &= 0x7F;                            // Set the most significant bit (Parity bit) to
0 leaving the 7 bit ASCII character.

    if ((_rxBufferTail + 1) == _rxBufferHead) {  // 6.2.7 - Overflow? If not, proceed.
      _bufferOverflow = true;                   // bufferOverflow status set and newChar is not
stored
      std::cout << "Buffer full - check overflowStatus() " << "\n";
    } else {                                    // 6.2.8 - Save char, advance tail.
      _rxBuffer[_rxBufferTail] = newChar;
      _rxBufferTail = (_rxBufferTail + 1) % _BUFFER_SIZE;// increments buffer tail and resets to
0 if _rxBufferTail+1 == BUFFERSIZE
    }
  }
}
```

Appendix D: Complete SDI12 Logger Program Listing (Excluding SDI12 Library)

Appendix D Contents

Appendix D.1: Organisation and Description of SDI12 Logger Program

```
/* ========================= Raspberry SDI12Logger Program Organisation =======================
This file ‘SDI12Logger.cpp’ and three non-standard C++ libraries make up the Raspberry Pi
logger. SDI12Logger.cpp has working functions that perform measurement handling and partially
complete functions to perform device configuration including adding a new SDI-12 device to the
logger configuration. Functions are grouped under four sections. Functions that are common to
both the measurement handling and device configuration processes are put into the generic
functions section.

1. main Function
        1)  main()


2. Measurement Handling Functions include:
        1)  dataHeadings()
        2)  measurementDelay()
        3)  takeMeasurement()

  Functions in the measurement handling section with addition of the generic ‘sendAndReceive’
  function performs the task of initiating a measurement sequence and logging of data. The
  sequence of menu options to initiate the logging session from the ‘main’ menu (assuming the
  logger has been pre-configured) takes place with a single user input, the user enter keys 3
  when in the ‘main’ menu.
  Functions called for a logging session include:
        1)  main()                              → see section: main() Function
        2)  dataHeadings()
        3)  measurementDelay()
        4)  takeMeasurement()
        5)  sendAndReceive()        → see section: Generic Functions (returns measurement time)
        6)  sendAndReceive          → see section: Generic Functions (returns measurement data)


3. SDI-12 Device Configuration Functions include
        1)  deviceConfiguration()
        2)  checkAddress()
        3)  getSensorModel()
        4)  addChannels()

  Functions in the SDI-12 device configuration section perform two main tasks however only 1 is
  Complete. The first task with addition of the generic ‘sendAndRecieve’ function performs the
  task of adding an SDI-12 device to the logger configuration. Only one sensor must be connected
  to the logger. The sequence of menu options to add an SDI-12 device to the configuration from
  the main menu takes place over two user inputs, the user enter keys 2 when in the ‘main’ menu
  followed by key 2 when in the ‘device configuration’ menu.
  Functions called to add an SDI-12 device include:
        1)  main()                  → see section: ‘main() Function’
        2)  deviceConfiguration()
        3)  sendAndReceive()        → see section: Generic Functions (returns address)
        4)  checkAddress()
        5)  getSensorModel()
        6)  addChannels()


4. Generic Functions include:
        1)  sendAndReceive()
        2)  getInteger()

========================================= Attribution =========================================
Authored in 2015 by James Coppock for major project undertaken as part of the requirements of a
Bachelor of Engineering. Project titled:
Development of a Raspberry Pi based, SDI-12 sensor environmental data logger



---------------------------| SDI12 Library Function To Write and Send | -----------------------
```

```
To write characters to the SDI-12 data line and receive chars the sequence of functions called
are:
        1. begin() → setState(INTERRUPTENABLE)

        2. sendCommand(string: cmd)      → wakeSensors()         → setState(TRANSMITTING)
                                         → writeChar(uint8_t)
                                         → writeChar(uint8_t)
                                         → ..........
                                         → writeChar(uint8_t)     → setState(LISTENING)
If sensor replies:
        1. handleInterrupt() →  receiveChar()
        2. handleInterrupt() → receiveChar()
        ..
        n. handleInterrupt() → receiveChar()

The receiveChars() function puts the characters into the buffer.

After delay period.

        1. available()
        2. parityError()
        3. overflowStatus()
        4. LFCheck()
        5. CRCheck()
        6. read()

---------------------- Reference for Interpreting Oscilloscope Display -------------------------

 NOTE when reading from the oscilloscope left of screen is the start bit. Some common binary
 responses that can be checked on the oscilloscope.

The address query command (?!) on an oscilloscope screen is.
1 0000 0011 0 (?)       1 0111 1011 0 (!)
Where
   ? = 33 (Dec) = 011 1111 ← LSB        (parity bit not shown)
   ! = 63 (Dec) = 010 0001 ← LSB        (parity bit not shown)

GS3 response to an address query command (a<CR><LF>) on oscilloscope screen (assuming sensor
address ‘a’ = 6 is:
 1 1001 0011 0 (Dec. = 6)       1 0100 1110 0 (carriage return)       1 1010 1111 0 (line feed)
Where;
   6  = 54 (Dec) = 011 0110 ← LSB       (parity bit not shown)
   CR = 13 (Dec) = 000 1101 ← LSB       (parity bit not shown)
   LF = 10 (Dec) = 000 1010 ← LSB       (parity bit not shown)
*/

/* ====================== 0. Includes, Defines, and Variable Declarations ====================

0.1 - Include the SDI12 library Authored by Kevin Smith for the Arduino and modified by myself
      for use with the Raspberry Pi.
0.2 - Include the file parser library shared on dream in code web site
     <http://www.dreamincode.net/forums/topic/183191-create-a-simple-configuration-file-parser/>
0.3 - Include the wiringPi library Authored by Gordon Henderson. The library give some simple
      functions for controlling the setting the state of the Raspberry Pi pins, writing a HIGH
      or LOW to the pin or reading digital voltage.

0.4 - Standard C++ libraries.
0.4.1 - Use with C++ style strings (std string)
0.4.2 - Defines macro for reporting error conditions.
0.4.3 - Gives the system function for system calls. system("....");
0.4.4 - libraries for uint8_t (unsigned character 8 bits), int8_t (Signed character 8 bits),
         int16_t (short int (2 bytes)), int32_t (4 bytes) int64 (long int (8 bytes))
0.4.5 - gives cin and cout for input and output
0.4.6 - Stream class to operate on strings

0.5 - Defines the BCM pin number connected to output enable of SN74HCT240 (2OE)
0.6 - Defines the BCM pin number connected to input 2A1 of the SN74HCT240
0.7 - Defines the BCM pin number connected to output enable of SN74HCT240 (1OE)
0.8 - Defines the BCM pin number connected to output 1Y1 of the SN74HCT240
0.9 - Global variable that is a number of characters entered into the character array for a
      given sensor response. The name of the characters array is 'response'. 'response' and
      '_charsAvailable' are initialised in sendAndReceive().
```

```
*/



// Required Libraries
#include "SDI12.cpp"                        // 0.1
#include </home/pi/Desktop/Parser.h>        // 0.2
#include <wiringPi.h>                       // 0.3
                                            // 0.4
#include <string>                           // 0.4.1
#include <errno.h>                          // 0.4.2
#include <stdlib.h>                         // 0.4.3
#include <inttypes.h>                       // 0.4.4
#include <iostream>                         // 0.4.5
#include <sstream>                          // 0.4.6
// Variable declarations and defines
#define TXENABLE 4                          // 0.5
#define TXDATAPIN 17                        // 0.6
#define RXENABLE 27                         // 0.7
#define RXDATAPIN 23                        // 0.8

int _charsAvailable;                        // 0.9
```

Appendix D.2: main() Function

```
/* main() - function that gives a menu of configuring options. The menu options are:
        0. Exit setup
        1. SDI-12 channels                      (not started)
        2. SDI-12 device configuration          (partially complete)
        3. Start Logging                        (Complete)

*/
int main ()
{

  // INITIALISATION
  // Setup wiring pi
  if (wiringPiSetupGpio () < 0) {                        //Setup using BCM pin numbers
    std::cout << "Error: Unable to setup wiringPi\n";
    return 1;
  }
  //Initialise state of two Output enable control pin of the SN74HCT240 (OE1 and OE2).
  pinMode(RXENABLE, OUTPUT);                    // RXENABLE (BCM 27) set output
  digitalWrite(RXENABLE, HIGH);                 // State of 244 output 1 (connected to RPi RX
input) in high impedance state (RXENABLE = BCM 27)
  pinMode(TXENABLE, OUTPUT);                    // TXENABLE (BCM 4) set output
  digitalWrite(TXENABLE, HIGH);                 /State of 240 output 2 (connected to SDI-12 bus
                                                // in high impedance state (TXENABLE=BCM 4)
  pinMode(TXDATAPIN, OUTPUT);                   // TXDATAPIN (BCM 17) set output

Initialise the interrupt service routine function on the 'RXDATAPIN' - The interrupt function
handleInterrupt() is a static member function from the SDI-12 class library. The function is
called after detecting a falling edge on RXDATAPIN.
  wiringPiISR (RXDATAPIN, INT_EDGE_FALLING, SDI12::handleInterrupt);
  pullUpDnControl(RXDATAPIN, PUD_UP);           // Set RX Pin with pull down resistors enabled
  system ("gpio edge 23 none");                 // Note: system calls use BCM pins numbers.
This disables interrupt on RXDATAPIN

  // Main menu loop
  for (;;) {
    if (piHiPri(1) < 0)  {        // Programming Priority between 1 and 99 (99 is the highest)
      std::cout << "Error: Unable to set priority low in main() \n”;
    }
    std::cout << "\nSDI-12 Device Configuration Menu Options\n";
    // Output 'Main Menu' options.
    std::cout << "Enter an integer from '0' to '3' and press enter.\n";
    std::cout << “0. Exit Setup\n 1. SDI-12 Channels (not started)\n 2. SDI-12 Device
Configuration (partially completed)\n 3. Start Logging\n";
    // getInteger() - a function that waits for a user to enter a valid input '0' to '3'
    int myNumber = getInteger(0, 3);

    if (myNumber == 0) {                        // Exit setup
      std::cout << "You entered 0\n";
      return 0;
    }
    if (myNumber == 1) {                        // SDI-12 channels
          std::cout << "You entered 1\n";
          viewChannels();
    }
    if (myNumber == 2) {                        // SDI-12 device configuration
          std::cout << "You entered 2\n";
          deviceConfiguration();
    }
    if (myNumber == 3) {                        // Start Logging
          std::cout << "You entered 3\n";
          dataFileHeadings();
    }
  }

return 0;

}
```

Appendix D.3: Measurement Handling Functions

```
/* dataFileHeadings() - a function called from main() when menu option 3 is selected. This
function takes measurements from the sensor addresses within the loggerconfiguration.txt file.
The results returned from the sensor will be stored in the appropriate channel.

When this function is first called this function checks to see if a datafile exists and if not
creates one. If the datafile exists it checks the loggerconfiguration.txt to see if any channels
have been added since the last logging session. When a change in channel configuration has been
made the value of the key ChanConfigChange is 'yes'. If the value is 'yes' a new channel names
are written to the .csv at the bottom of the existing data file and the value 'y1' is reset to
'no'.
*/

void dataFileHeadings() {
  std::cout << "measurementHandler() called\n";
  std::string line;
  // Declare myfilein as ifstream and this file can be read from the same as using cin
  std::ifstream myfilein ("loggerconfiguration.txt");            // read and write.
  size_t lineNo = 0;

  if (myfilein.is_open())  {
    // the while loop keeps extracting lines, until EOF is found.
    while ( std::getline (myfilein, line)) {
      lineNo++;
      std::string temp = line;
      // Checks line for 'ChanConfigChange' and if it exists outputs line with ChanConfigChange'
      // exists.
      if (line.find("ChanConfigChange") != line.npos) {
        // Check if line = 'chanConfigChange=yes'. If true new channel names are printed at the
        // bottom of the existing datafile and 'ChanConfigChange' is given a new value = 'no'.
        // Logging is started
        if (line.find("ChanConfigChange=yes") != line.npos) {
          myfilein.close();
          // Rewrite loggerconfiguration.txt file changing line 'ChanConfigChange=yes' to
          ChanConfigChange=no
          std::string strReplace = "ChanConfigChange=yes";       // line to replace
          std::string strNew = "ChanConfigChange=no";            // New line
          std::ifstream myfilein ("loggerconfiguration.txt");    // file to read from.
          std::ofstream myfileout ("temp.txt");                  // temporary file to write too
          if (myfilein.is_open() || myfilein.is_open())  {
            std::string strTemp;
            // the while loop extracts lines of loggerconfiguration.txt and prints each line to
            a new file 'temp.txt'. When line = "ChanConfigChange=yes" is found the new line =
            "ChanConfigChange=no" is printed.
            while ( std::getline (myfilein,strTemp) )  {
              if(strTemp == strReplace){
                strTemp = strNew;
              }
              strTemp += "\n";
              myfileout << strTemp;
            }
            myfilein.close();
            myfileout.close();
            // Delete original config file "loggerconfiguration.txt".
            remove("loggerconfiguration.txt");
            // rename "temp.txt" to "loggerconfiguration.txt".
            rename("temp.txt", "loggerconfiguration.txt");
            std::cout << "value for key 'ChanConfigChange' is set to yes \n";

            // Check if there are channels configured in the logger config file
'loggerconfiguration.txt'.
            ConfigFile lccfg("loggerconfiguration.txt");
            // getValueOfKey() - function from the parse.h library will return a key value for
            // the key 'NoOfConfigChannels'. The key value will be an integer representing the
            // number of configured channels. The value is returned as type 'int'.
            int noConfigChannels = lccfg.getValueOfKey<int>("NoOfConfigChannels");
            if (noConfigChannels <= 0) {
              std::cout << "No channels currently configured. Add channels in the main menu
through option 2. 'SDI-12 device configuration' \n";
```

```
              return;
            }
            // Channel exist in configuration list
            else {
              std::cout << "Number of configured channels is: " << noConfigChannels<< "\n";
              std::ofstream myfileout ("datafile.csv", std::ios::app); // datafile to write to
Create a string of channel names seperated by commas and write to the bottom of the data file.
              if (myfileout.is_open())  {
                // Initialise string variable of channel names with date and time.
                std::string chanNames = "Date,Time";
                // append additional channel names.
                for(int x=1; x <= noConfigChannels; x++) {
                  // convert channel number (int) to a string.
                  std::stringstream ss;
                  ss << x;
                  // Create a key for ConfigFileEntry8 that specifies a channel name and unit in
                  // the config file 'loggerconfiguration.txt'.
                  std::string str = "CH"+ss.str()+"n";
                  // getValueOfKey() - will return a key value = channel name and unit
                  std::string chanNames_new = lccfg.getValueOfKey<std::string>(str);
                  chanNames = chanNames + "," + chanNames_new;
                  //std::cout << "Channel Name: " << chanNames << "\n";
                }
                // Prints heading to datafile.csv
                myfileout << chanNames << std::endl;
                myfileout.close();
                // call measurementDelay() function which initiates a measurement on time
                measurementDelay();
                return;
              }
              else {
                std::cout << "Error: Unable to open datafile.csv in measurementHandler() \n";
                myfileout.close();
                return;
              }
            }
          }
          else {
            std::cout << "Error: Unable to open file 'temp.txt' or 'loggerconfiguration.txt'  \n
            myfilein.close();
            myfileout.close();
            return;
          }
        }


        // Check if line = 'ChanConfigChange=no'. If true data can be logged with current
        // channel headings.
        if (line.find("ChanConfigChange=no") != line.npos) {
          std::cout << "ChanConfigChange=no \n";
          myfilein.close();
          // call measurementDelay() function
          measurementDelay();
          return;
        }
        else{
          std::cout << "Error: key = value error in loggerconfiguration.txt. Key =
'ChanConfigChange' but value should be 'yes' or 'no' with no white space.  \n";
          myfilein.close();
          return;
        }
      }
    }
    std::cout << "Error: Did not find the key 'ChanConfigChange' in the configuration file) \n";
    return;
  }

  else {
    std::cout << "Error: Unable to open file loggerconfiguration.txt in measurementHandler() \n"
    return;
  }
}
```

```
/* measurementDelay() - a function called from dataFileHeadings() that delays the measurement
so that it occurs at a specific minute and second of the hour depending on what the measurement
interval is. The measurement will always occur referenced from hh:mm:ss = hh:00:00. Valid
measurement intervals are 2, 5, 10, and 20 minutes. These measurement intervals are specified in
configFileEntry 3 within 'loggerconfiguration.txt'.
*/
void measurementDelay() {
  std::cout << "measurementDelay() called         \n";
  // Get the measurement interval from the logger config file 'loggerconfiguration.txt'.
  ConfigFile lccfg("loggerconfiguration.txt");
  // getValueOfKey() - function from the parse.h library will return a
  // key value for the key 'MeasurementInt'. The key value will be an
  // integer representing measurement interval in minutes. Possible values
  // are 2, 5, 10 or 20 minutes.
  int measurementInt = lccfg.getValueOfKey<int>("MeasurementInt");

  time_t rawtime;
  struct tm * timeinfo;
  char hms_buffer [50];          // buffer for holding the current time returned from
localtime() in hours:minutes:second hh:mm:ss
  char date_buffer [50]; // buffer holding the date returned from localtime() in format
DD/MM/YYYY
  int sec;
  int secs_to_next_minute;
  int min;
  int minute_delay;

  std::string data;                         // Variable that holds the next line of the csv file.

  for(;;) {
    // Get an initial time in seconds and minutes to set a sleep count
    time (&rawtime);
    timeinfo = localtime (&rawtime);
    sec = timeinfo->tm_sec;                       //
    min = timeinfo->tm_min;
    secs_to_next_minute = 60 - sec;



    // For 2 minutes measurement interval
    if (measurementInt == 2) {
      // measurement points are done on every second minute from reference mm:ss = mm:00, where
mm can be 00, 02, 04 .., 58
      if(min%2 == 1) {
        // Minutes is divisible by 2 with a remainder. (delay = secs_to_next_minute)
        std::cout << "measureInt = 2 -1 \n";
        // Put the process in a sleep state while waiting to take next measurement.
        sleep(secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);                         // Get current time and date
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);            // Store time
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);           // Store date
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        std::cout << "new data: " << data << "\n";
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app); // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);      // sleep for 20 seconds incase of inaccuracy in the sleep command.
        piHiPri(1)      // Programming priority between 1 and 99 (99 the highest)
        continue;
      }
      else {
        // Minutes is an divisable by 2. (Delay = 1 minutes + secs_to_next_minute)
        std::cout << "measureInt = 2 -2 \n";
        // Put the process in a sleep state while waiting to take next measurement.
        sleep(60 + secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        std::cout << "new data: " << data << "\n";
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app); // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);      // sleep for 20 seconds incase of inaccuracy in the sleep command.
        piHiPri(1)      // Programming priority between 1 and 99 (99 the highest)
        continue;
      }
    }



    // For 5 minutes measurement interval
    if (measurementInt == 5) {
      // measurement points are done on every 5th minute from reference mm:ss = mm:00, i.e. mm
can be 00, 05, 10 .., 55
      if (min%5 != 0) {
        std::cout << "measureInt = 5 -1 \n";
        // Minutes is divisable by 5 with a remainder. (delay = 4-(min Mod 5) +
secs_to_next_minute)
        minute_delay = (4 - (min%5))*60;
        // Put the process in a sleep state while waiting to take next measurement.
        sleep(minute_delay + secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        std::cout << "new data: " << data << "\n";
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app); // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);       // sleep for 20 seconds incase of inaccuracy in the sleep command.
        if (piHiPri(1) < 0){     // Programming Priority between 0 and 99 (99 is the highest)
          std::cout << "Error: Unable to set priority low in main() \n" ;
        }
        continue;
      }
      if (min%5 == 0){
        std::cout << "measureInt = 5 -2 \n";
        // Minutes is an divisable by 5. (Delay = 4 minutes + secs_to_next_minute)
        // Put the process in a sleep state while waiting to take next measurement.
        sleep(240 + secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        std::cout << "new data: " << data << "\n";
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app); // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);       // sleep for 20 seconds incase of inaccuracy in the sleep command.
        if (piHiPri(1) < 0){     // Programming Priority between 0 and 99 (99 is the highest)
          std::cout << "Error: Unable to set priority low in main() \n" ;
        }
        continue;
      }
    }



    // For 10 minutes measurement interval
    if (measurementInt == 10) {
      // measurement points are done on every 10th minute from reference mm:ss = mm:00, i.e mm
can be 00, 10, 20 .., 50
      if (min%10 != 0) {
        std::cout << "measureInt = 10 -1 \n";
        // Minutes is divisable by 10 with a remainder. (delay = 9-(min Mod 10) +
secs_to_next_minute)
        minute_delay = (9 - (min%10))*60;
        // Put the process in a sleep state while waiting to take next measurement.
        sleep(minute_delay + secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        std::cout << "new data: " << data << "\n";
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app); // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);      // sleep for 20 seconds incase of inaccuracy in the sleep command.
        piHiPri(1)      // Programming Priority between 0 and 99 (99 is the highest)
        continue;
      }
      if (min%10 == 0){
        std::cout << "measureInt = 10 -2 \n";
        // Minutes is divisable by 10. (Delay = 9 minutes + secs_to_next_minute) Put the process
        // in a sleep state while waiting to take next measurement.
        sleep(540 + secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        std::cout << "new data: " << data << "\n";
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app);     // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);      // sleep for 20 seconds incase of inaccuracy in the sleep command.
        piHiPri(1)      // Programming Priority between 0 and 99 (99 is the highest)
        continue;
      }
    }


    // For 20 minutes measurement interval
    if (measurementInt == 20) {
      // measurement points are done on every 20th minute from reference mm:ss = mm:00, where mm
can be 00, 20, 40
      if (min%20 != 0) {
        std::cout << "measureInt = 20 -1 \n";
        // Minutes is divisable by 20 with a remainder. (delay = 19-(min Mod 20) +
secs_to_next_minute)
        minute_delay = (19 - (min%20))*60;
        // Put the process in a sleep state while waiting to take next measurement.
        sleep(minute_delay + secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app);      // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);      // sleep for 20 seconds incase of inaccuracy in the sleep command.
        piHiPri(1)   // Programming Priority between 0 and 99 (99 is the highest)
        continue;
      }
      if (min%20 == 0){
        std::cout << "measureInt = 20 -2 \n";
        // Minutes is divisable by 29. (Delay = 19 minutes + secs_to_next_minute) Put the
        // process in a sleep state while waiting to take next measurement.
        sleep(1140 + secs_to_next_minute);
        time (&rawtime);
        timeinfo = localtime (&rawtime);
        strftime (hms_buffer,50,"%H:%M:%S",timeinfo);
        strftime (date_buffer,50,"%d/%m/%Y",timeinfo);
        // Create string with date and time seperated by a comma
        data = (std::string)date_buffer + "," + (std::string)hms_buffer;
        // Get data from each channel
        std::string data2 = takeMeasurement();
        // data = new line to append to csv file
        data = data + "," + data2;
        std::cout << "new data: " << data << "\n";
        // Open data file
        std::ofstream myfileout ("datafile.csv", std::ios::app);      // datafile to write to
        // Print csv data to data file.
        if (myfileout.is_open())  {
          std::cout << "Opened datafile.csv. New data is written to the file. \n";
          // print data to file
          myfileout << data << std::endl;
          myfileout.close();
        }
        else {
          std::cout << "Error: Unable to open datafile.csv from measurementDelay() \n";
          myfileout.close();
          exit(EXIT_FAILURE);
        }
        sleep(20);      // sleep for 20 seconds incase of inaccuracy in the sleep command.
        piHiPri(1)      // Programming Priority between 0 and 99 (99 is the highest)
        continue;
      }
    }



    else {
      std::cout <<"Error: Measurement interval specified in ConfigFileEntry3 should be 2, 5, 10
or 20.”
      return;
    }
  }
}
```

```
// takeMeasurement() – CURRENTLY LIMITED TO SENSORS THAT RETURN ALL PARAMETERS IN RESPONSE
/* takeMeasurement() - a function called from measurementDelay() that reads the addresses of
sensors and channel information that is stored in loggerconfiguration.txt after through the
deviceConfiguration menu. Each address is sent a measurement command. This function returns a
string of channel parameter values separated by commas to measurementDelay() printing to
datafile.csv.
*/

std::string takeMeasurement() {
  std::cout << "measurementDelay() called         \n";

  if (piHiPri(99) < 0)   {       // Programming Priority between 0 and 99 (99 is the highest)
        std::cout << "Unable to set priority High in takeMeasurement()\n" ;
  }

  // Declare variable for storing csv results
  std::string channelData = "";           // csv channel data from all sensor addresses
  std::string channelDataN = "";          // csv channel data from a single sensor address

  // Construct ConfigFile object named lccfg for parsing loggerconfiguration.txt.
  ConfigFile lccfg("loggerconfiguration.txt");

  // keyExists() - function from the Parse.h library that checks if ConfigFileEntry2 key
'ConfiguredAddresses' exists in the loggerconfiguration.txt file.
  if (lccfg.keyExists("ConfiguredAddresses")) {
    // getValueOfKey() - function from the parse.h library will return a key value for the key
    // 'ConfiguredAddresses'. The key value will be an integer representing the number of
    // configured addresses.
    int noAddresses = lccfg.getValueOfKey<int>("ConfiguredAddresses");
    std::cout << "Number of configured address is:" << noAddresses << "\n";
    // loop executed for each unique sensor address
    for (int x=1; x<=noAddresses; x++) {
      // convert integer x to a string.
      std::stringstream ss;
      ss << x;
      // Create a key for ConfigFileEntry5 that specifies address of sensors
      std::string str = "a"+ss.str();

      if(lccfg.keyExists(str)) {
        // getValueOfKey() - will return a key values 'y5' = addresses of sensors in order they
        // were added to config file.
        std::string address = lccfg.getValueOfKey<std::string>(str);
        // Create a key for ConfigFileEntry6 that specifies the number of parameters returned by
        // the each sensor address
        str = "add"+address;
        if(lccfg.keyExists(str)) {
          // getValueOfKey() - will return a key values 'y6' = number of parameters for the
          current sensor address.
          int parameters = lccfg.getValueOfKey<int>(str);         // NOT USED
          // Construct  start measurement command (aM!) where 'a' is the sensor address
          std::string myCommand = address+"M!";

          // The response from sensor is atttn<CR><LF> where
          // a - is the sensor address
          // ttt - the time in seconds, until the sensor will have the measurements ready
          // n - the number of measurement values the sensor will return in one or more send
          // data commands.

          // The number of character returned for a start measurement command is 7.
          int noChars = 7;               // number of character in the response.

          // The delay must allow enough time for 7 characters to be received
          // The minimum delay is (833us x 10 X 7) + 15ms = 73.31ms.
          int delaymSec = 84;            // Delay in seconds
          // sendAndReceive() - a function that that sends the command and returns the sensor
          // responses. It takes the command, a minimum delay time for the response in
          // milliseconds, the minimum number of characters in the response and the sensor
          // address as inputs.
          // Pointer to array
          char *startMeasurement;
          startMeasurement = sendAndReceive(myCommand, delaymSec, noChars, address);
          // This if statement breaks the loop if a "Comms Error" occurred. Error may be due to
          // hardware or noise. It may also occur if the sensor is not connected or if there is
          // short circuit on the SDI-12 bus. It may also occur if the time value was read
          // incorrectly. It is not likely to be the OS scheduling as the command is resent
          // multiple times. The error message is printed to the data file, and the 'start
          // measurement' command is sent to the next sensor address.
          if (_charsAvailable == -1) {
            channelDataN = "Comm. Error";
            for (int i=1; i <= (parameters-1); i++) {
              channelDataN  = channelDataN + ",Comm. Error";
          }
          if (x == 1){
            channelData = channelDataN;
          }
          else {
            channelData = channelData + "," + channelDataN;
          }
          // std::cout << "channel data 1: " << channelData << "\n";
          std::cout << "Error communicating with sensor at address: " << address << "\n";
          continue;
          }

          // Get the first three character out of the array of received characters. The received
          // characters are the response to the 'start measurement' command and are put into a
          // string. The first three characters excluding the address character of a 'start
          // measurement' command are 'ttt', which is the time in seconds till data is available
          // from the sensor. After a delay of ttt seconds the data can be retrieved with a
          // 'send data' command.
          std::string ttt_string = "";
          for (int i=0; i<=2; i++) {
            char c = startMeasurement[i];
            ttt_string += c;
          }
          // Convert 'ttt' to an integer representing the time in seconds until the measurement
is ready.
          std::string str = ttt_string;
          std::istringstream ss(str);
          int ttt_int;
          ss >> ttt_int;

          // Convert the 'n' part of the 'start measurement' response from a 'char' to an 'int'
          // and check that it corresponds to the number of character that is specified in the
          // config file for that sensor. If not an error is sent to display and the number of
          // parameter specified in the config file is used.
          char n_char = startMeasurement[3];
          int n_int = n_char - '0';
          // Check if the number of
          if (n_int != parameters) {
            std::cout << "Error: The number of parameters returned by sensor at address '" <<
address << "' is not equal to the number specified in the config file <takeMeasurement()>\n";
          }
          // Set a delay of 'ttt' seconds.
          sleep(ttt_int);

          // Send data command (aD<x>!) is sent to the sensor. The 'x' value is initially 0 but
          // subsequent send data commands may be needed if the values of all parameter do not
          // fit within the maximum of 35 characters available for <value> field. The maximum
          // number of characters in a data value is 9 therefore 3 parameters will fit in one
          // response. If a sensor returns more than 3 parameters 'x' will be incremented and
          // the new send data command sent. The for loop sends the 'send data' command with
          // incrementing 'x'. When all the data is received the loop is exited.
          for (int y=0; y <= parameters-1; y++){                 // use this line
            // Convert incrementing 'send data' command number 'x' to a 'string' type.
            std::stringstream ss1;
            ss1 << y;
            // Construct a send data command aD<x>! with sensor address.
            myCommand = address+"D"+ss1.str()+"!";
            std::cout << "sendMeasurement command: " << myCommand << "\n";
/*
            //The response of a sensor to a 'send data' command is
            // a<values><CR><LF> where
            // a - is the sensor address
            // values - pd.d
            // p - the polarity sign (+ or -)
            // d - numeric digits before the decimal place
            // . - the decimal place (optional)
            // d - numeric digits after the decimal point.

            // The maximum number of characters in the <values> field is 35.
            // The maximum nuber of character for a single parameter is 9
            //       (includes a polarity sign + 7 digits + the decimal point)
            // The minimum number of characters for a single parameter is 2
            //       (includes a polarity sign + 1 digit (decimal is optional))

            // A multi-paramneter sensor will return at least three parameters within a single
            // 'send data' command.

            // the maximum number of character returned from send data command is 38. If a valid
            /response is returned but no data is returned the sensor has aborted the measurement
*/
            // the minimum number of character returned is 3. The response
            // by the sensor when no results are available is a<CR><LF>
            // noChars is the minimum number of chars returned.
            noChars = 3;

            // The delay must allow enough time for up to 35 characters to be received + the
            // address and <CR><LF>. The maximum time to receive 41 possible chars is 357 ms
            if ( (8.33 * ((parameters * 9) + 3)) + 15 <= 357   ) {
              delaymSec = (0.833 * 10 * ((parameters * 9) + 3)) + 15;
            }
            else {
              delaymSec = 357;   // Delay in milliseconds
            }

            // sendAndReceive() - a function that that sends the command and returns the sensor
            // responses. It takes the command, a minimum delay time for the response in
            // milliseconds, the minimum number of characters in the response and the sensor
            // address as inputs.
            // Pointer to array
            char *measurementRX;
            measurementRX = sendAndReceive(myCommand, delaymSec, noChars, address);

            //this if statement breaks the loop if a "No Result Error" occurred. Error may occur
            // due to bad communication due to hardware issues or noise. It is not likely to be
            // the OS scheduling as the command is resent multiple times. The error message is
            // printed to the data file.
            if (_charsAvailable == -1) {
              channelDataN = "No Result Error";
              for (int i=1; i <= (parameters-1); i++) {
                channelDataN  = channelDataN + ",No Result Error";
              }
              if (x == 1){
                channelData = channelDataN;
              }
              else {
                channelData = channelData + "," + channelDataN;
              }
              std::cout << "channel data 0: " << channelData << "\n";
              break;
            }


            // ********************* this part of code is not complete ***********************
            // If this if statement is true the sensor is returning a valid response without the
            // <values> field on second or higher iteration. There may have been a corruption of
            // values specifically polarity signs in the previous iterations that has resulted
            // in further iteration of the 'send data' command, which would result in no further
            // vales being returned. If true the loop is broken.
            // In current implementation the program should never get here and is not complete.
            if (_charsAvailable == 0 && y > 0) {
              std::cout << "Error: Unexpected iteration attempt for test sensors used \n";
              break;
            }
            // If this if statement is true the sensor is returning a valid response without
            // returning the <values> field on the first iteration. The logger may have obtained
            // the wrong time value in the 'start measurement' command, or the sensor may not
            // have results ready by the time it indicated.
            if (_charsAvailable == 0 && y == 0) {
              std::cout << "Error: No <values> field after the first iteration of a 'send data'
command \n";
              channelDataN = "No value returned";
              for (int i=1; i <= (parameters-1); i++) {
                channelDataN  = channelDataN + ",No value returned";
              }
              // This if statement checks if this is the result from the first configured sensor
address
              if (x == 1){
                channelData = channelDataN;
              }
              else {
                channelData = channelData + "," + channelDataN;
              }
              std::cout << "channel data 1: " << channelData << "\n";
              break;
            }


            int polaityCount = 0;
            int polarityPosition[parameters+2];
            std::string value_string = "";
            char c;
            // If this if statement is true the sensor is returning a valid response with a
            // <values> field on the first 'send data' command iteration. If true the values are
            // extracted and appended to the new datafile line separated by a commas.
            if (_charsAvailable > 0 && y == 0) {
              std::cout << "chars are available in first iteration. \n";
              // This if statement tests if the first character is a polarity sign
              if (measurementRX[0] == '+' || measurementRX[0] == '-') {
                polaityCount = 1;
                polarityPosition[polaityCount] = 0;
                // This for loop counts the number of polarity signs and the positions of the
                // polarity signs within the array of received characters.
                for (int i=1; i<= _charsAvailable-1; i++){
                  if (measurementRX[i] == '+' || measurementRX[i] == '-') {
                    polaityCount = polaityCount+1;
                    polarityPosition[polaityCount] = i;
                  }
                  if(measurementRX[i] == 0x0D){
                    // std::cout << "carriage return at element i = " << i << "\n";
                    polaityCount = polaityCount+1;
                    polarityPosition[polaityCount] = i;
                  }
                }
                // this if statement tests if the number of polarity signs is equal to the
                // number of parameter returned by the sensor. If true all parameters have been
                // returned in the first send data iteration. If true no further 'send data'
                // command iterations are needed.
                if (polaityCount-1 == parameters){
                  // this for loop extracts the values from the array of character received from
                  // the sensor. It will extract all parameter values that are expected to be
                  // returned by the sensor.
                  for (int i=1; i<=parameters; i++) { // for 1 to 3
                    value_string = "";
                    // this if statement tests if a parameter values polarity signs is equal to
                    // '-'. If the polarity is equal to '-' the polarity sign is printed to the
                    // data file.
                    if (measurementRX[polarityPosition[i]] == '-') {// if measurementRX[i] = '-'
                      if (i==1){
                        // if measureementRX[0] = '-'
                        for (int k=0; k<=polarityPosition[2]-1; k++) {
                          c = startMeasurement[k];
                          value_string += c;
                        }
                        channelDataN = value_string;
                      }
                      else {                            // measurementRX[i>0] = '-'
                        for (int k=polarityPosition[i]; k<=polarityPosition[i+1]-1; k++) {
                          c = startMeasurement[k];
                          value_string += c;
                        }
                        channelDataN  = channelDataN + "," + value_string;
                      }
                    }

                    // Else the polarity signs is equal to '+'. When the polarity is equal to
                    // '+' the polarity sign is not printed to the data file.
                    else {
                      // measureementRX[i] = '+'
                      if (i==1) {
                        for (int k=1; k<=polarityPosition[2]-1; k++) {
                          c = startMeasurement[k];
                          value_string += c;
                        }
                        channelDataN = value_string;
                        //std::cout << "value_string 3: " << value_string << " channelDataN 3:
" << channelDataN << "\n";
                      }
                      else {
                        for (int k=polarityPosition[i]+1; k<=polarityPosition[i+1]-1; k++) {
                          c = startMeasurement[k];
                          value_string += c;
                        }
                        channelDataN  = channelDataN + "," + value_string;
                        //std::cout << "value_string 4: " << value_string << " channelDataN 4:
" << channelDataN << "\n";
                      }
                    }
                  }
                  if (x == 1){
                    channelData = channelDataN;
                  }
                  else {
                    channelData = channelData + "," + channelDataN;
                  }
                  std::cout << "channel data 2: " << channelData << "\n";
                  break;
                }

                // ******************** this part of code is not complete ********************
                // else the number of polarity signs in the first iteration of the 'send data'
                // command is not equal to the number of parameters expected to be returned by
                // the sensor. Values should be extracted from the array of received characters
                // and further 'send data' command iterations are initiated by continuing the
                // for loop. The sensor used for testing measurement functions should return
                // all parameter values in the first 'send data' iteration.
                else {
                  std::cout << "Error: The number of polarity signs extracted in the first
iteration is not equal to the number \n";
                  std::cout << "       parameters returned by the sensor. The sensors used in
testing this logger should return \n";
                  std::cout << "       all parameters in the first iteration, therefore a
corrupt value has been received.\n";
                  for (int i=1; i <= (parameters-1); i++) {
                    channelDataN  = channelDataN + ",Corrupt value Error";
                  }
                  // This if statement checks if this is the result from the first configured
sensor address
                  if (x == 1){
                    channelData = channelDataN;
                  }
                  else {
                    channelData = channelData + "," + channelDataN;
                  }
                  std::cout << "channel data 3: " << channelData << "\n";
                  break;
                }
```

```
              }

              // else the first character was not a '+' or '-' character, therefore corrupt
              // characters are present. No result are recorded in this implementation.
              else {
                channelDataN = "Error: The first character was not a '+' or '-' character,
therefore a corrupt value has been receive";
                channelDataN = "Corrupt value Error";
                for (int i=1; i <= (parameters-1); i++) {
                  channelDataN = channelDataN + ",Corrupt value Error";
                }
                // This if statement checks if this is the result from the first configured
sensor address
                if (x == 1){
                  channelData = channelDataN;
                }
                else {
                  channelData = channelData + "," + channelDataN;
                }
                std::cout << "channel data 4: " << channelData << "\n";
                break;
              }
            }


            // ******************* This part of code is not complete ************************
            // If this if statement is true the sensor is returning a valid response with the
            // <values> field on the second or higher iteration. Values should be extracted from
            // the array of received characters. The sensor used for testing measurement
            // functions should return all parameter values in the first 'send data' iteration.
            // This section of code is not yet complete. It is not needed unless sensor return
            // <values> over multiple 'send data' command iterations. \
            if (_charsAvailable > 0 && y > 0) {
              std::cout << "Chars are available in second iteration \n";
              std::cout << "Error: All values should be received in the first 'send data'
iteration for the GS3 and 5TM. \n";
              std::cout << "Error: Even if there is data corruption it should not reach this
point. \n";
            }
          }
        }

        else {
          // Unable to find configFileEntry6 key add<x6>
          std::cout << "Error: Unable to find ConfigFileEntry6 key 'add<x6>' = '" << str << "'
in loggerconfiguration.txt in takeMeasurement() \n";
          exit(EXIT_FAILURE);
        }
      }
      else {
        // Unable to find ConfigFileEntry5 key a<x5>
        std::cout << "Error: Unable to find ConfigFileEntry5 key 'a<x5>' = '" << str << "' in
loggerconfiguration.txt from takeMeasurement() \n";
        exit(EXIT_FAILURE);
      }
    }
  }
  else {
    // Unable to find ConfigFileEntry2 key 'ConfiguredAddresses'
    std::cout << "Error: Unable to find ConfigFileEntry2 Key 'ConfiguredAddresses' in
loggerconfiguration.txt. \n";
    exit(EXIT_FAILURE);
  }
  std::cout << "channel data 5: " << channelData << "\n";
  return channelData;
}
```

Appendix D.4: SDI-12 Device Configuration Functions

Appendix D.4.1 Main Configuration Handler Function

```
/* deviceConfiguration() - a function called from main() that gives a menu of configuring
options and performs the tasks by calling other functions which return . Menu options are:
        0. Return to Main Menu
        1. Change address of SDI-12 sensor       (Feature not written)
        2. Add SDI-12 device                     (Partially Complete)
*/
void deviceConfiguration() {
  std::cout << "deviceConfiguration() called \n";
  piHiPri(99)    // Set thread to the highest priority(99)

  for (;;) {
    std::cout << "\nSDI-12 Device Configuration Menu Options\n";
    std::cout << "Only one SDI-12 device should be connected to the SDI-12 bus 3\n";
    // Outputs Device configuration options.
    std::cout << "Enter an integer from '0' to '2' and press enter.\n";
    std::cout << "0. Return to Main Menu\n 1. Change address of SDI-12 sensor (not started)\n 2.
Add SDI-12 device (partially complete)\n";
    // getInteger() waits for user to enter a valid input between '0' to '2'
    int myNumber = getInteger(0, 2);

    // If "0" - return to main menu
    if (myNumber == 0) {
      std::cout << "You entered 0\n";
      return;
    }



    // If "1" - proceed with SDI-12 sensor address change
    if (myNumber == 1) {
      std::cout << "You entered 1\n";
    }



    // if "2" - proceed with adding SDI-12 device
    if (myNumber == 2) {
      std::cout << "You entered '2': Add SDI-12 Device\n";




      // getAddress()-function that gets the address of a SDI-12 sensor that is connected to bus
      // the electrical interface of the logger
      char address = getAddress();
      // if "address = |" - a valid sensor address was not returned. Program waits for user to
      // re-select from menu options.
      if (address == '|') {
        std::cout << "Unable to get a valid sensor address after 6 retry\n";
      }
      // if address is not = | - a valid sensor address was returned. Program continues.
      else {
        std::cout << "Address of sensor from deviceConfiguration() is: " << address << "\n";
        // if a valid address was returned the config file is checked to see if it in use by
        // another sensor on a configured channel. checkAddress() - function will return a
        // 'true' if address is not in use or 'false' if address is in use.
        if (checkAddress(address) == true) {
          std::cout << "Address of sensor connected is not used by another previously configured
sensor. \n" ;
          // getSensorModel() - function that sends a "send identification" command to the
          // sensor address and returns the model of the sensor
          std::string model = getSensorModel(address);
          if (model == "Error") {
            std::cout << "Unable to get a valid sensor model number after 6 retry attempts\n";
          }
          else {
            for (;;) {
              // User confirms sensor model output to command window.
              std::cout << "\n Is the sensor model '" <<model<< "' the correct sensor model?
Enter '0' or '1' and press enter.\n\n";
              std::cout << "0. No\n 1. Yes\n\n";
              // Wait for user to enter a valid input between '0' to '1'.
              int myNumber = getInteger(0, 1);
                if (myNumber == 1) {             // User pressed '1'.
                  // addChannels() - a function that finds information on the sensor in the
                  // sensorinformation.txt file and adds the channels to the logger
                  // configuration.txt file.
                  // Convert 'char' type address to a 'string' type address.
                  std::stringstream ss;
                  std::string add; // variable 'add' will hold the address as a 'char' type
                  ss << address;
                  ss >> add;
                  int finishAdd = addChannels(model,add);
                  if (finishAdd == 1) {
                    // back to deviceConfiguration()
                    std::cout << "Sensor channels successfully added \n";
                  }
                  else if (finishAdd == 0)  {
                    std::cout << "Sensor model was not found in the database
(SensorInformation.txt) \n";
                    std::cout << "Information about the sensor should be entered in the
SensorInformation.txt \n";
                    std::cout << "file before the sensor can be configured to a channel.
Information should be \n";
                    std::cout << "available from the sensor manufacturer";
                  }
                  else {        // implies finishAdd = "-1" -Error occurred in adding the sensor
                    std::cout << "Error occurred in adding sensor \n";
                  }
                  break;
                }
                else {                      // User pressed '0'.
                  std::cout << "return to Device configuration menu \n";
                  break;
                }
              }
            }
          }
          else {
            std::cout << "\n Address of sensor connected is used by another configured sensor.
The address of the sensor must be changed before the sensor can be added. \n";
          }
        }
      }
    }
  }
```

Appendix D.4.2: Add SDI-12 Device

```
/* checkAddress() – second function called from deviceConfiguration() when option 2 is
selected (Add SDI-12 Device). This function checks if the address of a new added SDI-12 sensor
is in the configured list before adding. This function searches for the key:
        add<a>
where 'a' is the sensor address. If it is found the address must be changed before adding the
sensor and if not the sensor can be added. Function will return a 'true' if address is not in
use or 'false' if address is in use.
*/

bool checkAddress(char address) {
  std::cout << "checkAddress() called \n";
  // Construct and ConfigFile object named cfg for parsing loggerconfiguration.txt.
  ConfigFile lccfg("loggerconfiguration.txt");

  // Convert 'char' type address to a 'string' type address.
  std::stringstream ss;
  std::string a;                // a is the variable that will hold the converted char address
  ss << address;
  ss >> a;
  std::string key = "add"+a;
  std::cout << "key" <<  key << "\n";
  // keyExists() - function from the Parse.h library that searches for
  // the loggerconfiguration.txt file for 'add<x>' where x is the address
  // the new sensor.
  if (lccfg.keyExists(key)) {
    return false;
  }
  else {
  return true;
  }
}
```

```
/* getSensorModel() - third function called from deviceConfiguration() that sends an
Identification command (<a>I!) where 'a' is a sensor address returned from getAddress(). The
sensor will respond with:
        allccccccccmmmmmmvvvxxx<CR><LF>

To configure the sensor from the database of SDI-12 sensors automatically the six character
model number is extracted from the response. The model number will be character 12 through to
17.

All checks are made to minimise the chance of an error due to the scheduling system of the Linux
OS. All characters received are checked for parity errors plus the carriage return and line feed
characters are checked. If there is an error detected the command is re-sent to the sensor

*/
std::string getSensorModel(char address) {
  std::cout << "getSensorModel() called\n";
  std::cout << "Address of sensor from getSensorModel() is: " << address << "\n";
  // Covert a 'char' type address to a 'string' type address.
  std::stringstream ss;
  std::string a;                 // ‘a’ is the ‘char’ address converted to 'string' type.
  ss << address;
  ss >> a;

  // Command aI! is the send identification command where 'a' is the sensor address
  std::string myCommand = a+"I!";

  // Construct an SDI12 object named mySDI12
  SDI12 mySDI12(TXENABLE, TXDATAPIN, RXENABLE, RXDATAPIN);
  // Initialise variable to count the number of times the command is sent to the sensor.
  int retryAttempt = 0;

  //Allow 6 attempts at retrieving a valid SDI-12 model. A valid address is registered when no
  //parity errors exist in the sensor response and the <LF> and <CR> characters end the response
  //as specified in the SDI-12 standard.
  while (retryAttempt <= 5) {
  // begin() - function from the SDI-12 library that sets the state prior to outputting a
  // command sequence. The rising edge interrupt is set to enabled but the receive pin is
  // isolated from the bus.
  mySDI12.begin();
  // outputs variable (Note cout << (unsigned char))
  std::cout << "String sent to SDI12 bus:" << myCommand << "\n";
  mySDI12.sendCommand(myCommand);
  // The maximum number of character returned for a send identification command is 35. The delay
  // must allow enough time for 35 characters to be received so that the carriage return and
  // line feed can be checked. The minimum delay is 833us x 10 X 35 = 291ms.
  delay(310);

  // available() - function from the SDI12 library that checks the number of character in the
  // buffer. It is expected to be between 22 and 35 characters returned for a send
  // identification command. 13 character including those for the sensor serial number field are
  // optional.
  int charsInBuffer = mySDI12.available();
  if (charsInBuffer > 20) {
    std::cout << "Number of characters in buffer" << mySDI12.available() << "\n";
    // Checks for a parity error (false = no parity error)
    if (mySDI12.parityErrorStatus() == false) {
      std::cout << "No parity error \n";
      // Checks for a buffer overflow (false = no overflow)
      if (mySDI12.overflowStatus() == false) {
        std::cout << "No buffer overflow\n";
        // Checks that a line feed <LF> character ended the sensor response
        if (mySDI12.LFCheck() == true) {
          std::cout << "Last character is a linefeed (LF)! \n";
          // Checks that a carriage return <CR> character came before the line feed character
          if (mySDI12.CRCheck() == true) {
            std::cout << "Second last character is a carriage return (CR)! \n";
            // Reads the address returned
            char add = mySDI12.read();
            if (add == address) {
              std::cout << "Address is a match with that sent \n";
              std::cout << "Address Query Command Retry Attempt Number: " << retryAttempt
              // advance buffer head 10 positions to position of first 6 character "sensor
        model" field entry in the buffer.
              mySDI12.advanceBufHead(10);
              std::string model = "";
              for (int i=0; i<=5; i++) {
                char c = mySDI12.read();
                // checks the next character is a space if space do not add to string.
                if (c == ' ')    {
                  std::cout << "Model number character not added: " << c<< "\n";
                }
                else {
                  std::cout << "Model number character added: " << c<< "\n";
                  model += c;
                  std::cout << "buffer: " << model << "\n";
                }
              }
              return model;              // Need to change this.
            }
            else {
              ++retryAttempt;
              std::cout << "address received is not the sensor address sent out with command! \
              std::cout << "Address Query Command Retry Attempt Number: " << retryAttempt << "\n
              mySDI12.flush();
            }
          }
           else {
             ++retryAttempt;
             std::cout << "Second last character is not a carriage return (CR)! \n";
             std::cout << "Address Query Command Retry Attempt Number: " << retryAttempt << "\n’
             mySDI12.flush();
           }
         }
         else {
           ++retryAttempt;
           std::cout << "Last character is not a linefeed (LF)! \n";
           std::cout << "Address Query Command Retry Attempt Number: " << retryAttempt << "\n";
           mySDI12.flush();
         }
       }
       else {
         ++retryAttempt;
         std::cout << "overflowStatus error\n";
         std::cout << "Address Query Command Retry Attempt Number: " << retryAttempt << "\n";
         mySDI12.flush();
       }
     }
     else{
       ++retryAttempt;
       std::cout << "parityErrorStatus error \n";
       std::cout << "Address Query Command Retry Attempt Number: " << retryAttempt << "\n";
       mySDI12.flush();
     }
   }
   else {
   ++retryAttempt;
  std::cout << "Not enough characters available\n";
  std::cout << "Address Query Command Retry Attempt Number: " << retryAttempt << "\n";
  mySDI12.flush();
  }
  }

  std::cout << "Unable to get sensor address after 6 resend attempts! \n";
  return "Error";
}
```

```
// addChannels() - NOT COMPLETE - START HERE TO FINISH ADD SDI-12 DEVICE
/* addChannels() - a function called from deviceConfiguration() when option 2 is selected (Add
SDI-12 Device). This function searches a configuration file "sensorinformation.txt" which
contains SDI-12 sensor information on the sensor model. The sensor model was found using the
getSensorModel() function. The model parameter is used in this function. The program searches
the file and if the sensor model is found the sensor parameters can be assigned to a channel.
The information needed:
        1) Number of parameter returned by the sensor
        2) The order of results returned from sensor
        3) The name of the parameters returned
        4) The units of the parameters returned

These details for a sensor would be available from sensor manufacturers. The information on a
sensor can be added to the sensorinformation.txt file by manually by any user. The .txt file
provides a means of configuring the sensor without having to enter information every time a new
sensor is added to the logger.
*/

int addChannels(std::string model, std::string address) {
  std::cout << "addChannels() called\n";
  // Construct and ConfigFile object named sicfg for parsing sensorinformation.txt.
  ConfigFile sicfg("sensorinformation.txt");

  // Construct ConfigFile object named lccfg for parsing loggerconfiguration.txt.
  ConfigFile lccfg("loggerconfiguration.txt");

  // keyExists() - function from the Parse.h library that checks if the sensor model name exists
  // as a key in the sensorinformation.txt file.
  if (sicfg.keyExists(model)) {
    std::cout << "Sensor model exists! \n";
    // getValueOfKey() - function from the parse.h library will return a key value for the key =
    // sensor model in the sensorinformation.txt file. The key value will be an integer
    // representing the number of parameters returned from the attached sensor model. The value
    // is returned as an 'int' type.
    int noParameters = sicfg.getValueOfKey<int>(model);
    // Note the value is returned as an int and not a string here
    // getValueOfKey() - will return a key value where key is the "NoOfConfigChannels" from the
    // loggerconfiguration.txt file. The key value will be an integer number representing the
    // number of channels the logger has configured.
    int noChannels = lccfg.getValueOfKey<int>("NoOfConfigChannels");

    std::cout << "Channels are not added to the config file for sensor model '" <<model<<"' at
address '" << address << "' because the addChannels() function is not finished.\n";


    // Important....... Need to set the value of ChanConfigChange to yes in
    // loggerconfiguration.txt after adding the channel.
  }
  else {
    std::cout << "Attached sensor model does not exist within 'sensorinformation.txt' \n"
  }
  return 1;
}
```

Appendix D.5: Generic Functions

```
/* sendAndReceive() - a function called from takeMeasurements() that sends the command and
returns the sensor response as a char array. It takes the command, a minimum delay time for the
response in milliseconds, the minimum number of characters in the response and the sensor
address as inputs.

A valid response is registered when no parity errors exist in the sensor response and the <LF>
and <CR> characters end the response as specified in the SDI-12 standard.
*/
  char* sendAndReceive(std::string myCommand, int delaymSec, int noChars, std::string address) {
  std::cout << "sendAndReceive() called \n";

  // Construct an SDI12 object named mySDI12
  SDI12 mySDI12(TXENABLE, TXDATAPIN, RXENABLE, RXDATAPIN);
  // Initialise variable to count the number of times the command is sent to the sensor.
  int retryAttempt = 0;
  // Allow 10 attempts at retrieving a valid response
  while (retryAttempt <= 10) {
  // begin() - function from the SDI-12 library that sets the state prior to outputting a
  // command sequence. The rising edge interrupt is set to enabled
  mySDI12.begin();
  if(retryAttempt >=1) {
    // Delay 30ms to clear any noise
    delay(30);
  }
  // outputs the SDI-12 command sent to the sensor
  std::cout << "String sent to SDI12 bus:" << myCommand << "\n";
  mySDI12.sendCommand(myCommand);
  // set delay in milliseconds using the function input parameter delaymSec
  delay(delaymSec);

  // available() -function from the SDI12 library that checks the number of character in buffer
  int charsInBuffer = mySDI12.available();

  if (charsInBuffer >= noChars) {
    std::cout << "Number of characters in buffer" << charsInBuffer << "\n";
    // Checks for a parity error (false = no parity error)
    if (mySDI12.parityErrorStatus() == false) {
      // Checks for a buffer overflow (false = no overflow)
      if (mySDI12.overflowStatus() == false) {
        // Checks that a line feed <LF> character ended the sensor response
        if (mySDI12.LFCheck() == true) {
          // Checks that a carriage return <CR> character came before the line feed character
          if (mySDI12.CRCheck() == true) {
            // Reads the address returned
            char add = mySDI12.read();
            // Convert variable add from char type to string type
            std::stringstream ss;
            std::string addN;    // ‘addN’ is the address (type 'char') to be converted to a
'string' type.
            ss << add;
            ss >> addN;
            // Check the first character is the address that the command was sent to.
            if (addN == address) {
              std::cout << "Address is a match with that sent \n";
              // If this if statement evaluates to true the send command and receive characters
              // sequence is started again as the sensor measurement may have been aborted
              if (charsInBuffer == 3 && retryAttempt <= 2) {
                ++retryAttempt;
                std::cout << "Error: Sensor response has no <values> field indicating the and
may have been aborted - Sequence restarted \n";
                std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
                mySDI12.flush();
                continue;
              }

              if (charsInBuffer == 3 && retryAttempt <= 5) {
                std::cout << "Error: Sensor response has no <values> field after 2 attempts. May
not be any data left or measurement aborted by sensor. \n";
```

```
                std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
                mySDI12.flush();
                _charsAvailable = 0;
                static char response[1] = {'@'};
                return response;
              }
              std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
              // store response excluding the address in a character array (max response is 75
chars)
              static char response[75];
              std::string response_str = "";
              for (int i=0; i<=(charsInBuffer-2); i++) {
                //for (int i=0; i<=(noChars-4); i++) {
                char c = mySDI12.read();
                //std::cout << "Character output from buffer are: " << c << "\n";
                response_str += c;
                response[i] = c;
              }
              // Global variable that is an integer number representing the number of characters
              // in the array 'response'.
              _charsAvailable = charsInBuffer-1;
              return response;
            }
            else {
              ++retryAttempt;
             std::cout << "Error: address received is not the sensor address sent in command!\n”
              std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
              mySDI12.flush();
            }
          }
          else {
            ++retryAttempt;
            std::cout << "Error: Second last character is not a carriage return (CR)! \n";
            std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
            mySDI12.flush();
          }
        }
        else {
          ++retryAttempt;
          std::cout << "Error: Last character is not a linefeed (LF)! \n";
          std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
          mySDI12.flush();
        }
      }
      else {
        ++retryAttempt;
        std::cout << "Error: overflowStatus error\n";
        std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
        mySDI12.flush();
      }
    }
    else{
      ++retryAttempt;
      std::cout << "Error: parityErrorStatus error \n";
      std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
      mySDI12.flush();
    }
  }
  else {
    ++retryAttempt;
    std::cout << "Error: Not enough characters available\n";
    std::cout << "sendAndReceive Retry Attempt Number: " << retryAttempt << "\n";
    mySDI12.flush();
  }
  }
  std::cout << "Error: Unable to get valid response after 6 resend attempts! \n";
  // return the error response
  _charsAvailable = -1;
  static char response[1] = {'?'};
  return response;
}
```

```
/* getInteger() - a function called after a menu option is displayed. The function accepts a
minimum and maximum integer as function parameters. Any character that is an integer outside of
'min' and 'max' or any other entered string of characters will result in invalid entry and
program will continue to wait for a valid entry
*/
int getInteger(int min, int max){
  //std::cout << "getInteger() called\n";
  // Get a valid input
  int myNumber = 0;
  std::string input = "";

  while (true){
    std::cout << "please enter a valid number: \n";
    std::getline(std::cin, input);
    // This code converts from a string to a number safely
    std::stringstream myStream(input);
    //
    if (myStream >> myNumber && myNumber >= min && myNumber <=max){
          return myNumber;
    }
  }
}
```

Appendix E: Flowcharts for SDI-12 Logger Functions

Appendix E Contents_Toc433562070

Appendix E List of Figures

Appendix E.1: Flowchart for main()

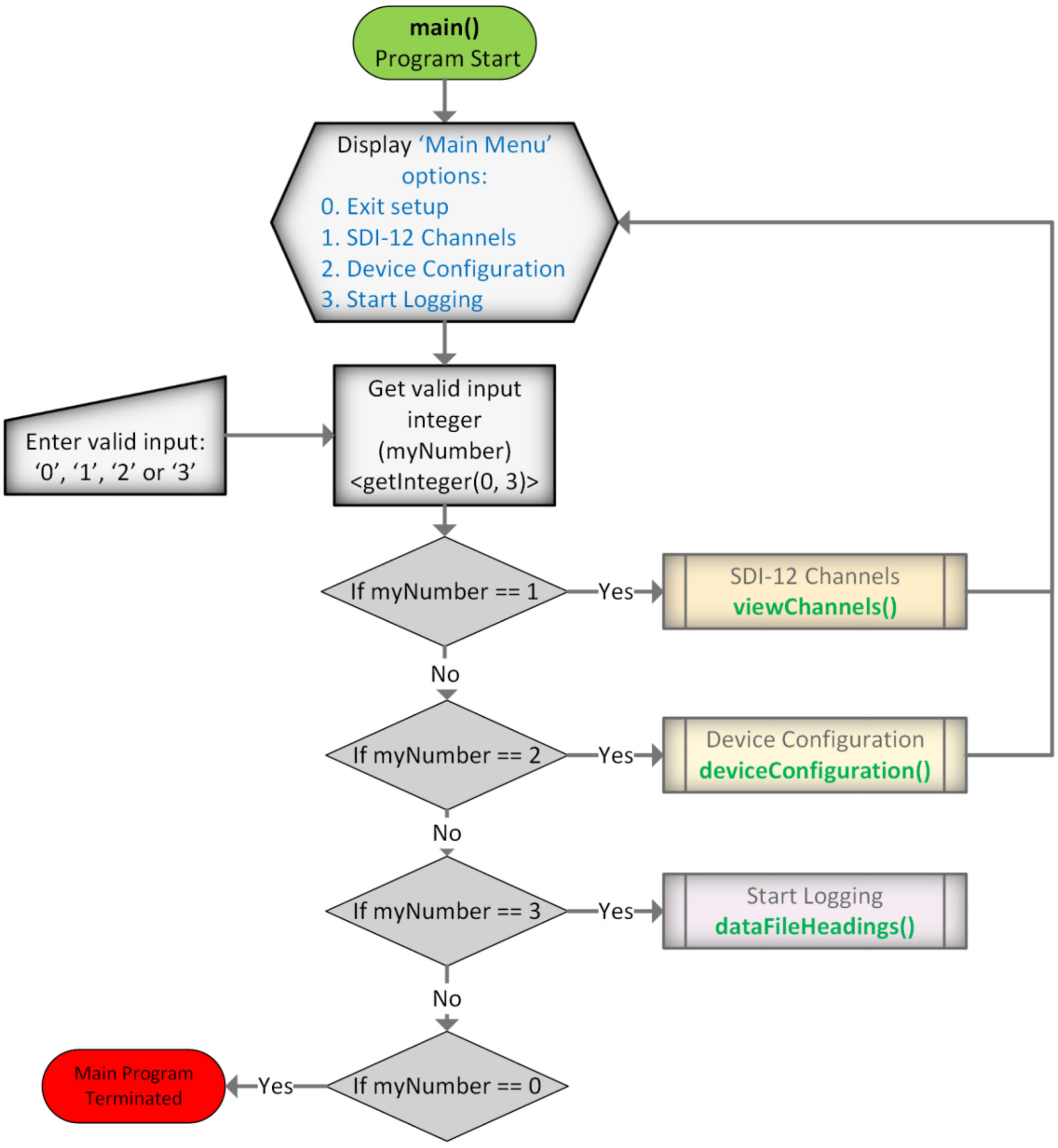

Figure E.1 Flowchart for main()

Appendix E.2: Flowchart for dataFileHeadings()

This function is called from main() when menu option 3 is selected. This functions checks ConfigFileEntry1 of loggerconfiguration.txt file to see if a channel configuration change has been made since the last logging session. If a channel configuration change has been made new channel headings (which include 'Date' and 'Time' followed by '<Channel name>+<unit>') are printed to datafile.csv and then the values field of ConfigFileEntry1 is changed from 'yes' to 'no'. New headings are extracted from the values field of ConfigFileEntry 8 .
With correct data file heading the channel data the measurementDelay() function is called to which retrieves measurements from sensor

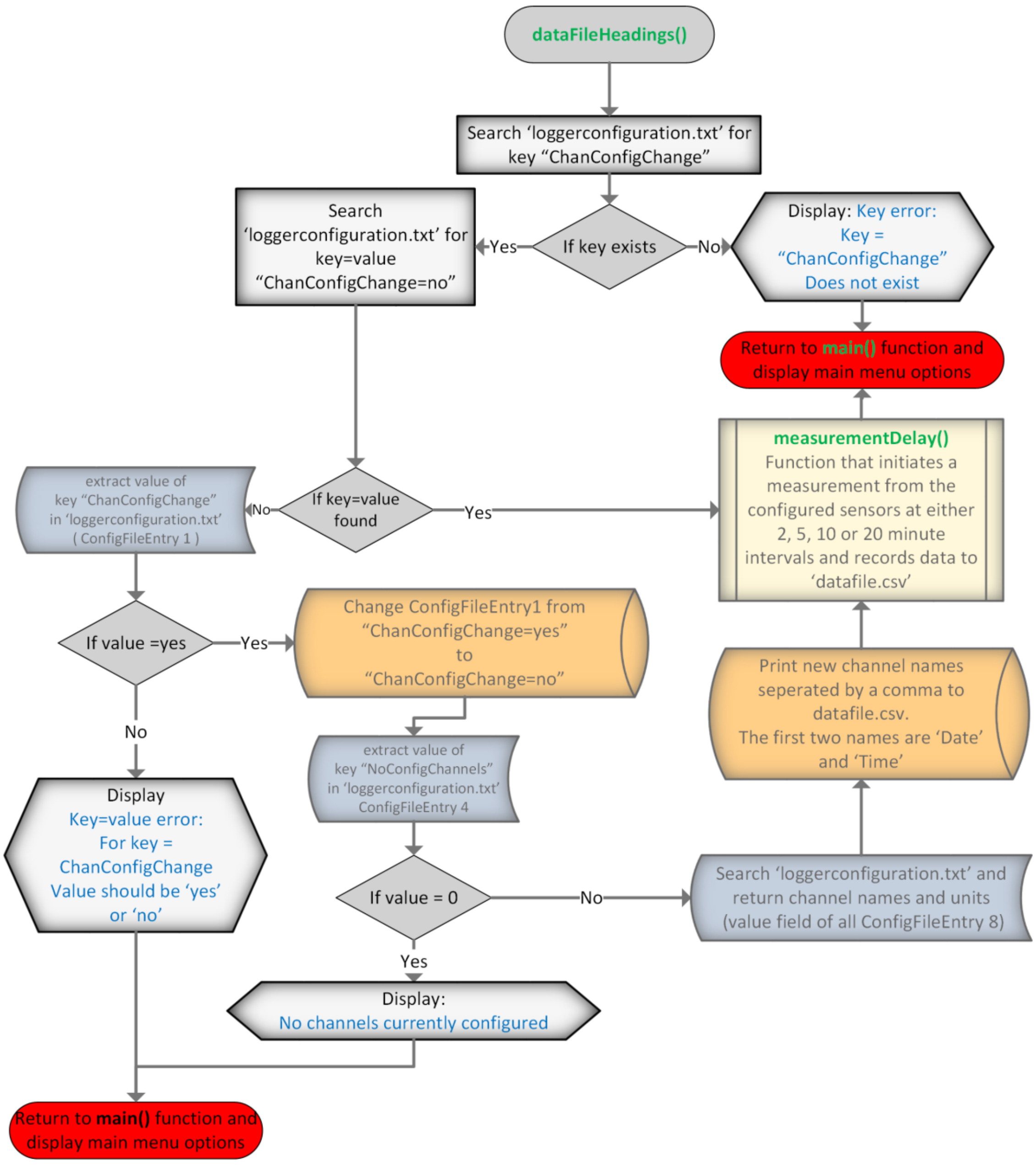
dataFileHeadings()
Search 'loggerconfiguration.txt' for key "ChanConfigChange"
If key exists
Yes
No
Display: Key error: Key = "ChanConfigChange" Does not exist
Return to main() function and display main menu options
Search 'loggerconfiguration.txt' for key=value "ChanConfigChange=no"
If key=value found
Yes
No
measurementDelay() Function that initiates a measurement from the configured sensors at either 2, 5, 10 or 20 minute intervals and records data to 'datafile.csv'
extract value of key "ChanConfigChange" in 'loggerconfiguration.txt' (ConfigFileEntry 1)
If value =yes
Yes
No
Change ConfigFileEntry1 from "ChanConfigChange=yes" to "ChanConfigChange=no"
Print new channel names seperated by a comma to datafile.csv. The first two names are 'Date' and 'Time'
extract value of key "NoConfigChannels" in 'loggerconfiguration.txt' ConfigFileEntry 4
If value = 0
No
Yes
Search 'loggerconfiguration.txt' and return channel names and units (value field of all ConfigFileEntry 8)
Display Key=value error: For key = ChanConfigChange Value should be 'yes' or 'no'
Display: No channels currently configured
Return to main() function and display main menu options

Figure E.2 Flowchart for dataFileHeadings()

Appendix E.3: Flowchart for measurementDelay()

This function is called from dataFileHeadings(). This functions gets the measurement interval from ConfigFileEntry3 of loggerconfiguration.txt file and delays the measurement so that it occurs at a specific minute and second of the hour depending on what the measurement interval is. The measurement will always occur referenced from 0 minutes and 0 seconds with valid measurement intervals are 2, 5, 10 or 20 minutes.

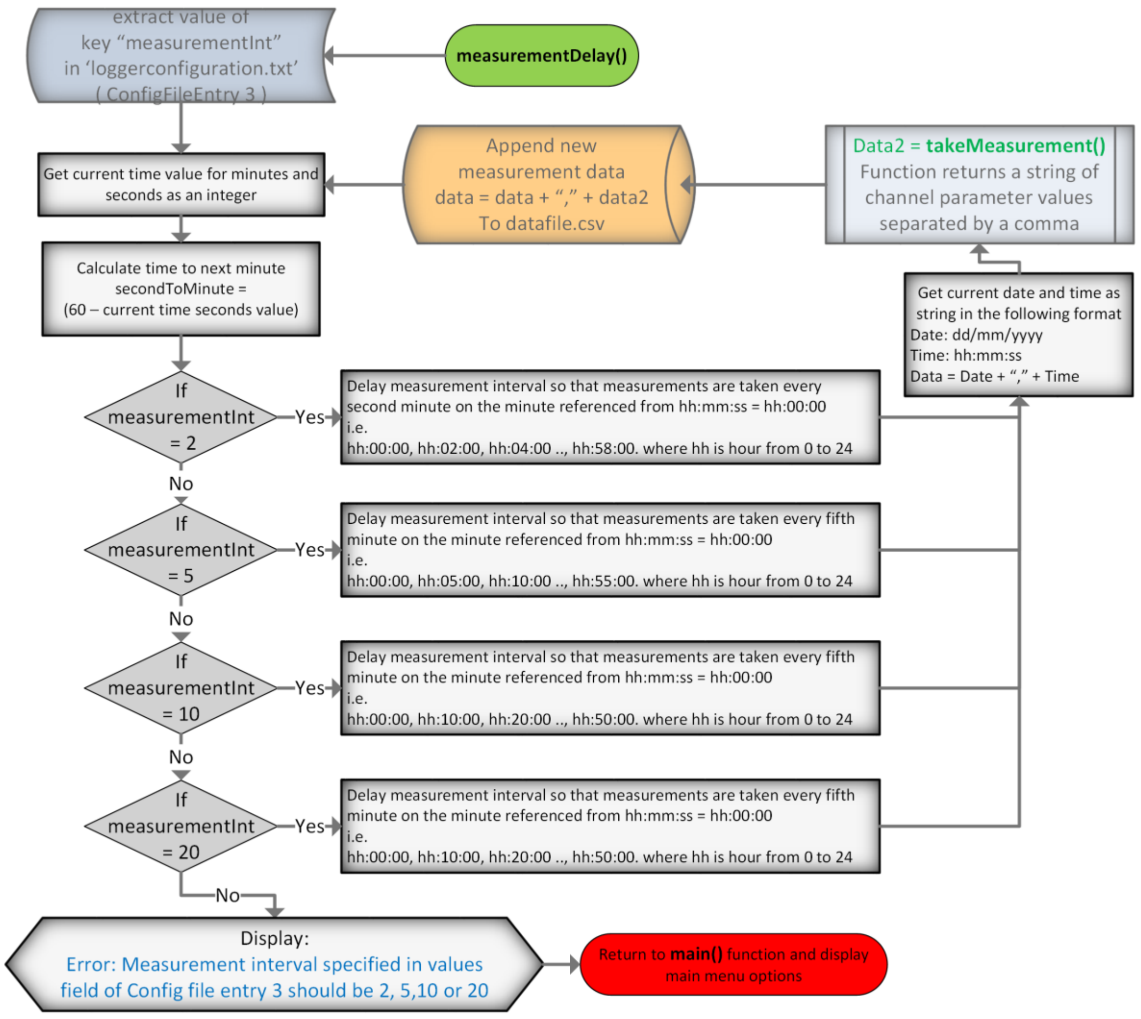
measurementDelay()
extract value of key "measurementInt" in 'loggerconfiguration.txt' (ConfigFileEntry 3)
Get current time value for minutes and seconds as an integer
Calculate time to next minute secondToMinute = (60 – current time seconds value)
If measurementInt = 2
Yes
Delay measurement interval so that measurements are taken every second minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:02:00, hh:04:00 .., hh:58:00. where hh is hour from 0 to 24
No
If measurementInt = 5
Yes
Delay measurement interval so that measurements are taken every fifth minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:05:00, hh:10:00 .., hh:55:00. where hh is hour from 0 to 24
No
If measurementInt = 10
Yes
Delay measurement interval so that measurements are taken every fifth minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:10:00, hh:20:00 .., hh:50:00. where hh is hour from 0 to 24
No
If measurementInt = 20
Yes
Delay measurement interval so that measurements are taken every fifth minute on the minute referenced from hh:mm:ss = hh:00:00 i.e. hh:00:00, hh:10:00, hh:20:00 .., hh:50:00. where hh is hour from 0 to 24
No
Display: Error: Measurement interval specified in values field of Config file entry 3 should be 2, 5,10 or 20
Return to main() function and display main menu options
Get current date and time as string in the following format Date: dd/mm/yyyy Time: hh:mm:ss Data = Date + "," + Time
Data2 = takeMeasurement() Function returns a string of channel parameter values separated by a comma
Append new measurement data data = data + "," + data2 To datafile.csv

Figure E.3 Flowchart for measurementDelay()

Appendix E.4: Flowchart for takeMeasurment()

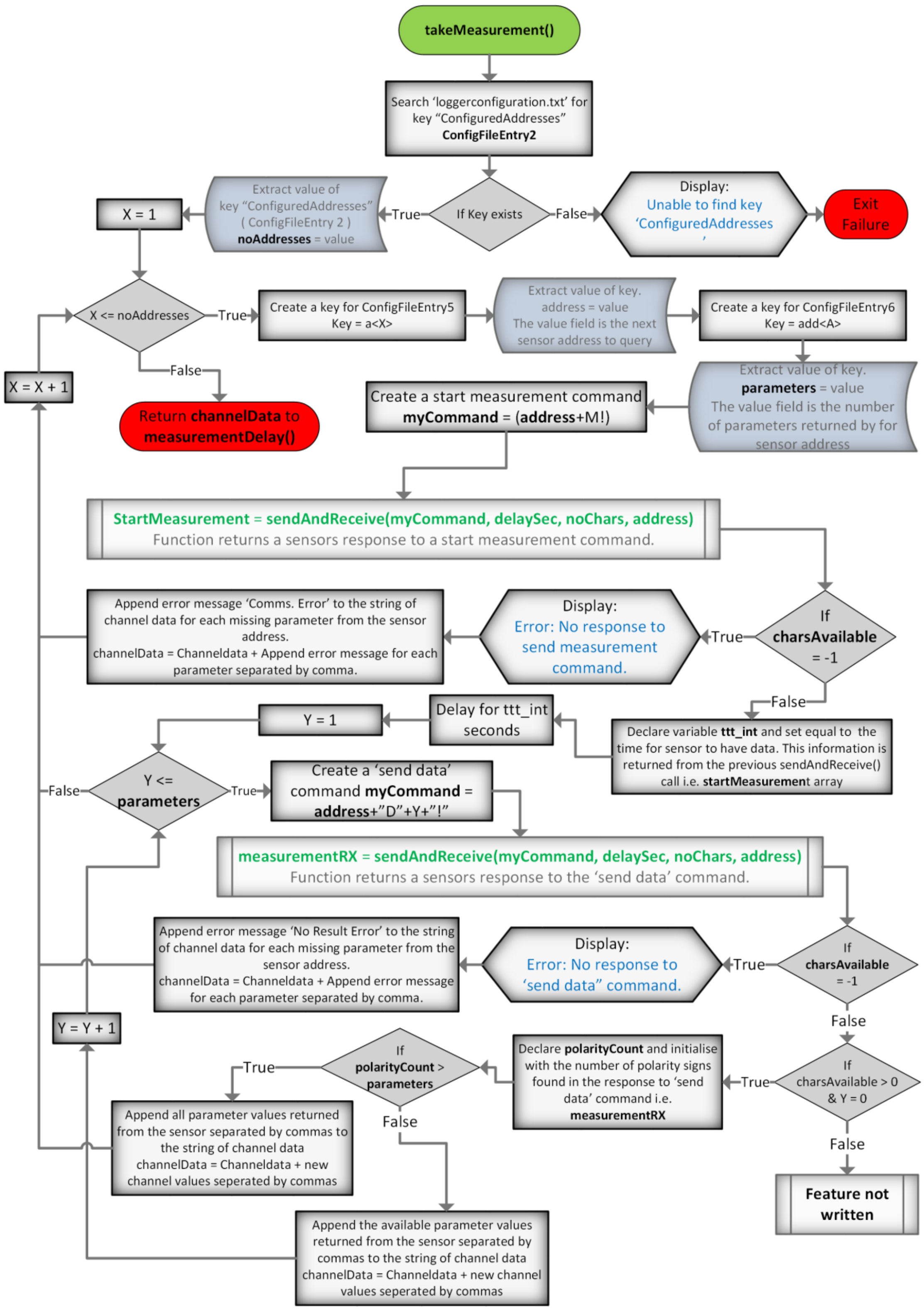
takeMeasurement()
Search 'loggerconfiguration.txt' for key "ConfiguredAddresses" ConfigFileEntry2
If Key exists
True
False
Display: Unable to find key 'ConfiguredAddresses'
Exit Failure
Extract value of key "ConfiguredAddresses" (ConfigFileEntry 2) noAddresses = value
X = 1
X <= noAddresses
True
False
Create a key for ConfigFileEntry5 Key = a<X>
Extract value of key. address = value The value field is the next sensor address to query
Create a key for ConfigFileEntry6 Key = add<A>
Extract value of key. parameters = value The value field is the number of parameters returned by for sensor address
Create a start measurement command myCommand = (address+M!)
Return channelData to measurementDelay()
X = X + 1
StartMeasurement = sendAndReceive(myCommand, delaySec, noChars, address)
Function returns a sensors response to a start measurement command.
If charsAvailable = -1
True
False
Display: Error: No response to send measurement command.
Append error message 'Comms. Error' to the string of channel data for each missing parameter from the sensor address. channelData = Channeldata + Append error message for each parameter separated by comma.
Declare variable ttt_int and set equal to the time for sensor to have data. This information is returned from the previous sendAndReceive() call i.e. startMeasurement array
Delay for ttt_int seconds
Y = 1
Y <= parameters
True
False
Create a 'send data' command myCommand = address+"D"+Y+"!"
measurementRX = sendAndReceive(myCommand, delaySec, noChars, address)
Function returns a sensors response to the 'send data' command.
If charsAvailable = -1
True
False
Display: Error: No response to 'send data" command.
Append error message 'No Result Error' to the string of channel data for each missing parameter from the sensor address. channelData = Channeldata + Append error message for each parameter separated by comma.
Y = Y + 1
If charsAvailable > 0 & Y = 0
True
False
Feature not written
Declare polarityCount and initialise with the number of polarity signs found in the response to 'send data' command i.e. measurementRX
If polarityCount > parameters
True
False
Append all parameter values returned from the sensor separated by commas to the string of channel data channelData = Channeldata + new channel values seperated by commas
Append the available parameter values returned from the sensor separated by commas to the string of channel data channelData = Channeldata + new channel values seperated by commas

Figure E.4 Flowchart for takeMeasurement()

Appendix E.5: Flowchart for sendAndReceive()

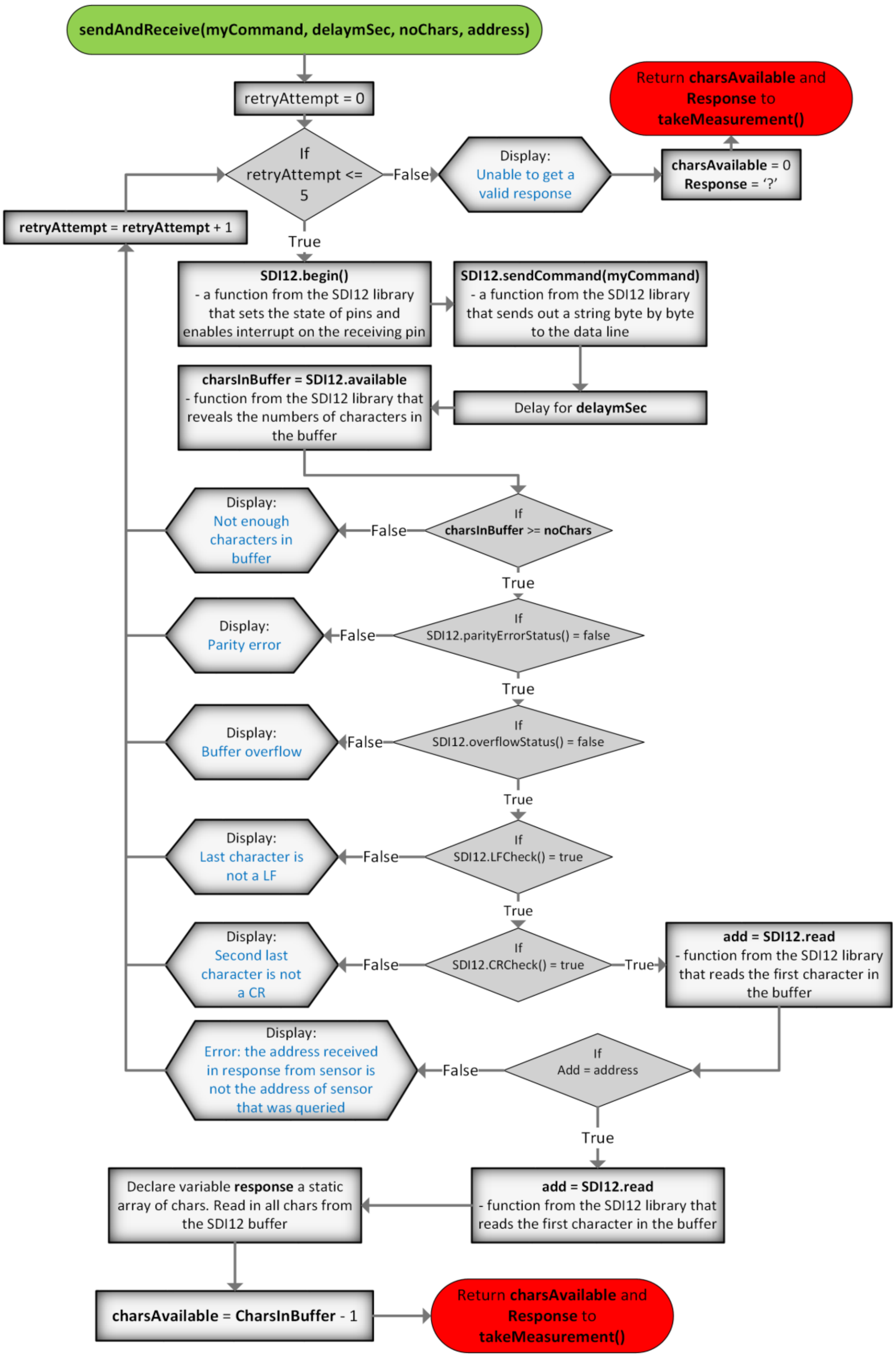
sendAndReceive(myCommand, delaymSec, noChars, address)
retryAttempt = 0
If retryAttempt <= 5
False
Display: Unable to get a valid response
charsAvailable = 0
Response = '?'
Return charsAvailable and Response to takeMeasurement()
retryAttempt = retryAttempt + 1
True
SDI12.begin() - a function from the SDI12 library that sets the state of pins and enables interrupt on the receiving pin
SDI12.sendCommand(myCommand) - a function from the SDI12 library that sends out a string byte by byte to the data line
Delay for delaymSec
charsInBuffer = SDI12.available - function from the SDI12 library that reveals the numbers of characters in the buffer
If charsInBuffer >= noChars
Display: Not enough characters in buffer
If SDI12.parityErrorStatus() = false
Display: Parity error
If SDI12.overflowStatus() = false
Display: Buffer overflow
If SDI12.LFCheck() = true
Display: Last character is not a LF
If SDI12.CRCheck() = true
Display: Second last character is not a CR
add = SDI12.read - function from the SDI12 library that reads the first character in the buffer
If Add = address
Display: Error: the address received in response from sensor is not the address of sensor that was queried
add = SDI12.read - function from the SDI12 library that reads the first character in the buffer
Declare variable response a static array of chars. Read in all chars from the SDI12 buffer
charsAvailable = CharsInBuffer - 1
Return charsAvailable and Response to takeMeasurement()

Figure E.5 Flowchart for sendAndReceive()

Appendix E.6: Flowchart for deviceConfiguration()

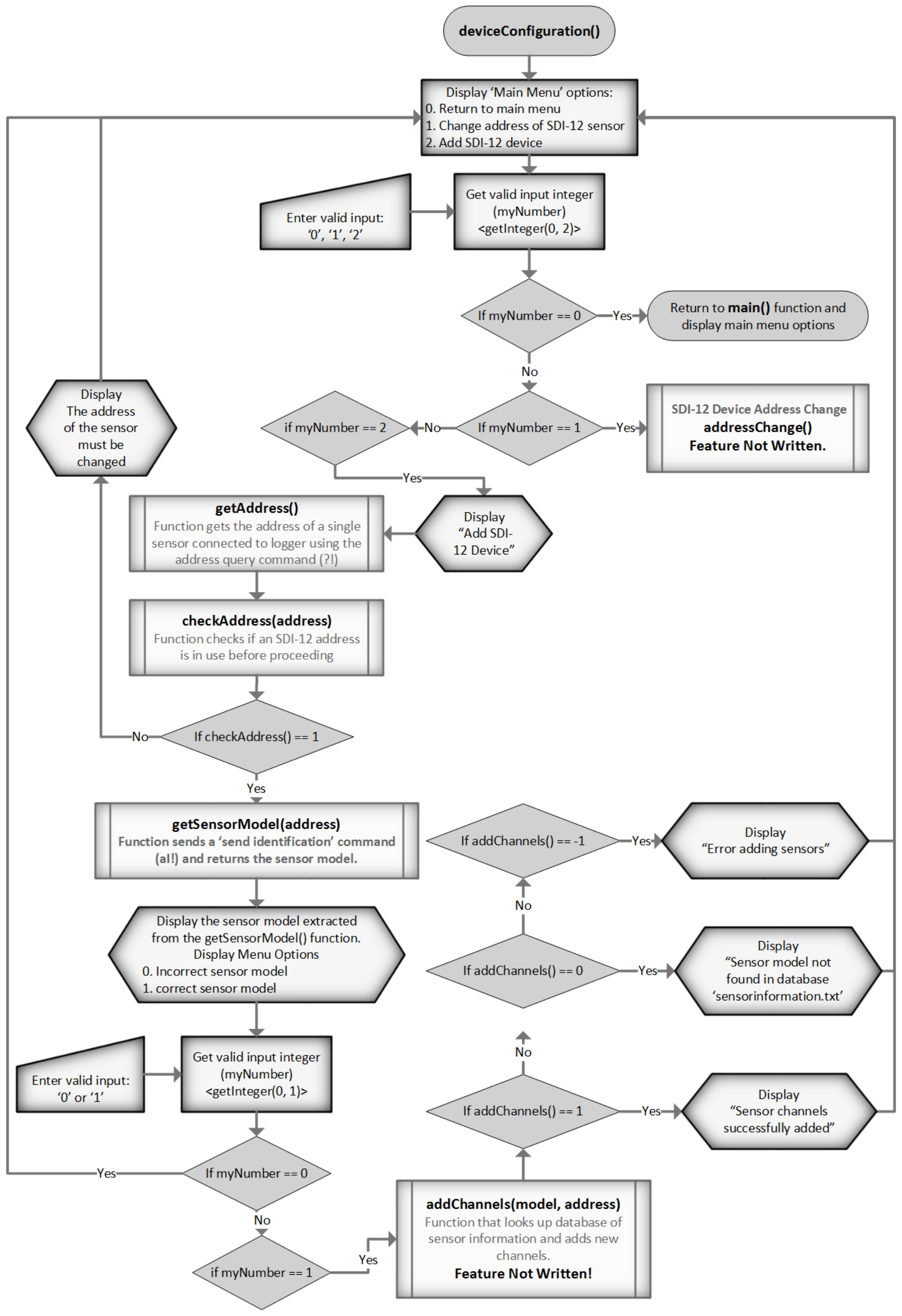
deviceConfiguration()
Display 'Main Menu' options:
0. Return to main menu
1. Change address of SDI-12 sensor
2. Add SDI-12 device
Enter valid input: '0', '1', '2'
Get valid input integer (myNumber) <getInteger(0, 2)>
If myNumber == 0
Yes
Return to main() function and display main menu options
No
If myNumber == 1
Yes
SDI-12 Device Address Change
addressChange()
Feature Not Written.
No
if myNumber == 2
Yes
Display "Add SDI-12 Device"
getAddress()
Function gets the address of a single sensor connected to logger using the address query command (?!)
checkAddress(address)
Function checks if an SDI-12 address is in use before proceeding
If checkAddress() == 1
No
Display The address of the sensor must be changed
Yes
getSensorModel(address)
Function sends a 'send identification' command (aI!) and returns the sensor model.
Display the sensor model extracted from the getSensorModel() function.
Display Menu Options
0. Incorrect sensor model
1. correct sensor model
Enter valid input: '0' or '1'
Get valid input integer (myNumber) <getInteger(0, 1)>
If myNumber == 0
Yes
No
if myNumber == 1
Yes
addChannels(model, address)
Function that looks up database of sensor information and adds new channels.
Feature Not Written!
If addChannels() == 1
Yes
Display "Sensor channels successfully added"
No
If addChannels() == 0
Yes
Display "Sensor model not found in database 'sensorinformation.txt'
No
If addChannels() == -1
Yes
Display "Error adding sensors"

Figure E.6 Flowchart for deviceConfiguration()