Open Design Orienteering Timing System

Specification and User Guide

Lawrence Jones

September 2019
revision 1 *working*

Contents

Chapter 1

Introduction

The Open Design Orienteering Timing System (ODOTS) seeks to provide an alternative low-cost electronic timing solution for Orienteering courses. It does this by leveraging the recent development and proliferation of RFID technology and by maintaining an open and accessible design that can be implemented or modified by the wider orienteering community.

In the interests of achieving the goals of the ODOTS it is requested that any further developments made on the design specified here are reported back to the central development team (email: l.jones4243@gmail.com) and made freely available for others to use and further develop.
This document has the following purposes:

- To describe the Open Design Orienteering Timing System (ODOTS) for anyone wishing to use it or assess it.
- Provide a reference for the inner workings of the ODOTS to help developers and debuggers understand the hardware, software and importantly the interfaces between them.
- To act as a user guide for anyone wishing to implement and use the ODOTS, whether this includes building it or not.

For readers wishing to use this document for the first purpose, section 2 contains the quantitative and descriptive specification and performance metrics of the ODOTS. For reader wishing to use this document for the second purpose, Part I is a reference for the hardware design and software structure of the ODOTS. For reader wishing to use the ODOTS for themselves Parts II and III are intended to take you through the process of building and using an ODOTS.

1.1 Version History

9/2018 ... v0 ... Proof of Concept

The ODOTS proof of concept development has been generously supported by the Orienteering Foundation.

Part I

Developer Guide

1.2 Development of the ODOTS

The ODOTS is an open design, all effort has been made in this part of the document to provide the explanation necessary to understand how the ODOTS works and the interactions of its various components. A large portion of this section is inevitably code documentation due to the large software component of the ODOTS. Other elements include performance and hardware specifications an overview of the hardware design, interface specification and some commentary on the various design elements. Every effort will be made to keep this section up to date with the code and design released, however it is possible functions and design decisions will be made after the main documentation write up that will be missed and thus not be explained or included here. If you have any questions please contact the central development team.

1.3 Compatibility and Version Identification

One of the Aims of the ODOTS is to generate community interest in the development of a tailored timing system and it is foreseen that there may end up being multiple development teams producing ODOTS examples. Version identification will then be particularly important, for this reason all ODOTS devices (embedded systems with RFID readers) will be required to identify their Hardware and Firmware version numbers. Version Numbers are 1 byte values, nibble 1 identifies the development team, nibble 2 identifies the design version for that team. For example, the proof of concept software and firmware is version 0 from team 0, therefore both numbers = 0x00. This is clearly a limited name space. New development teams are free to select their own nibble values, if they desire to have their values remain uniquely identifiable it is suggested that they announce them to the central development team, which will attempt to ensure there are no clashes.

It is clear the the name *ODOTS* is a good descriptor of what this project is but does not have the distinctive ring perhaps required for widespread recognition. Therefore the central development team actively encourages design versions to be given names in addition to their ODOTS version numbers, as long as it is clear that it has been built for ODOTS. For this purpose the Proof of Concept fails somewhat, as it is named 'Proof of Concept' (FW version 0x0, HW version 0x0).

It is the nature of community driven projects that there will be several different versions of the ODOTS design developed and presented. It is important, for the image of the project, that the compatibility between designs is well recorded. This will prevent confusion for people discovering the ODOTS and help avoid the nightmare scenario of users' races being wasted due to incompatible racer cards being used. The probability of losing races may be low in practice if card cost is so low that they can be issued to all competitors for each course.

What should the central development team do?

- Provide clear, explicit documentation of the basic ODOTS including the specification of major interfaces (This document).

- Provide clear, explicit documentation of any major design changes and how this impacts any previous specifications.
- Remain available for communication to help new developers.

What should potential developers do?

- Make compatibility information available for their design (or design addition/update) and any other major deployments of the ODOTS or versions of the basic design.
- If possible, make interface and design documentation available for future developers.
- Communicate your work back to the central development team so that records can be kept of ODOTS designs currently active.

This is not a binding list of actions to take, merely a suggestion.

1.4 Version History

1.4.1 Hardware

v0 ... 9/2019 ... Proof of concept

1.4.2 Software

v0 ... 9/2019 ... Proof of concept

1.4.3 Documentation

User Guide and Specification
(As per introduction)
v0 ... 9/2019 ... Proof of concept

Chapter 2

Performance and Specification

This chapter provides information of the performance of the ODOTS and some brief technical specifications for use in assessing the ODOTS for an application. These values have been derived from the proof of concept hardware, so improvements in most characteristics may be expected in future design iterations particularly with regards to power consumption.

Technical Characteristics

Table 2.1: Specimin Technical Characteristics

Characteristic	**min**	**typ**	**max**	**unit**
Device Input Supply Voltage	4.9V	5.0	5.1	V
Device Current Draw (IDLE)[1]		41		mA
Device Current Draw (RFID IN USE)[2]	50		65	mA
Firmware Flash Memory Usage[3]		12.2		kBytes
Firmware RAM Memory Usage (Global Reserved)[3]		0.6		kBytes
Firmware Persistent Memory Usage (EEPROM)		4		Bytes

[1] Measured from Strip board prototype with no outstanding serial or RFID actions.
[2] Measured from Strip board prototype with RFID card present, higher value indicates draw after Capacitor reservoir drained.
[3] Measured with Arduino Compiler.

Performance Evaluation Metrics

The basic hardware for any ODOTS unit will be roughly the same, therefore most ODOTS units will be able to fulfil any function required of any ODOTS unit. All units can act as Checkpoint units, ClearCard Reset units, or Card information download units. The only difference between these units will be a serial to USB bridge added to the download unit.

Table 2.2: Specimin Performance Characteristics

Characteristic	**min**	**typ**	**max**	**unit**
Time to Power Drain[1]	10		52	Hours
Visit Process Time[2]	6	20	45	ms
Clock Drift	1		100	sec/Week
Punches Per Card[3]	92		180	Visit Records
PCD Cost	21.5			£
PICC Cost	0.5			£

[1] Assumes fully charged 18650 cell with full dicharge achieved.
[2] Time will vary depending on timing of visit record with relation to power preserving sleep period.
[3] Will vary with card type, full capacity not yet achieved on MIFARE 4k card.

At the moment the ODOTS does not employ many power saving measures, hence the rather short endurance (Time to Power Drain). The current firmware will allow an ODOTS unit to comfortably run for two days on a single fully charged 18650 cell, enough for the Proof of Concept. It is possible that with greater care to paid to power consumption and the implementation of sleep modes this figure could be stretched out a great deal.

Competitor visit process time is hard limited to just more than 5ms by the RF standard in use. Currently this value may be reached with lucky timing (a competitor arriving just at the end of a sleep period). The hard maximum time has been calculated as Minimum time + sleep time (40ms) + clock update time (<1ms). In testing the difference was not noticeable.

Chapter 3

Firmware Specification and Guide

This section will look at the firmware running on the embedded processor, an AtMega328p in the proof of concept. The firmware has three purposes, to maintain an accurate clock, to write checkpoint visit records to passing MIFARE cards and to interact with a Serial host to fulfil download and configuration functions.

As mentioned previously the processor used in the proof of concept is an AtMega328p chosen due to its compatibility with the Arduino bootloader and option for Dual Inline Packaging (DIP). Long term accurate time keeping is achieved with the DS1337 real time clock chip while RFID functionality is achieved with the MRFC522 which also contains extra hardware used to handle the authentication and encryption required by the MIFARE standard.

In general every effort has been made to try and ensure that source file interactions make sense and can be best described with a multitier architecture where lower tiers get closer to hardware function while higher tiers get progressively more abstract and, hopefully, readable. Sitting outside of this structure is a configuration file 'Config.h' that contains many definitions (and a few global variable definitions and declarations) that determine firmware and hardware functionality. Centralising these definitions will hopefully simplify any debugging or augmentation. Table 3.1 has a brief outline of the firmware source files.

DS1337.cpp, DS1337.h, ODOTSRTC.cpp and ODOTSRTC.h are used to control the real time clock function and provide time stamps to the RFID module when required. MFRC522.cpp, MFRC522.h, ODOTS.cpp and ODOTS.h manage communications with the RFID module and the handling of frames being passed to the RFID card. require_cpp11.h and deprecated.h are required by MRFC522.h to maintain compatibility with the Arduino compiler. ODOTS.h and

Software Layer	**Source Files**
Application	ODOTSGeneric.ino
Library Management	ODOTS.h
Function Management	ODOTSSerialUtils.h, ODOTS.h, ODOTSRTC.h, LowPower.cpp, LowPower.h
Peripheral Device Drivers	BuzzerUtils.h, MFRC522.h, DS1337.h
Basic Hardware Abstraction	Arduino HAL

Table 3.1: Brief structure of firmware source files

.cpp also acts as the top layer of the library with glue code to pass non-RFID functionality to the application. ODOTSSerialUtils.cpp and ODOTSSerialUtils.h manage serial communications between the Microcontroller and a UART host. BuzzerUtils.cpp and BuzzerUtils.h contain code used to drive the notification peripherals, a buzzer and LEDs.

DS1337.cpp and .h is largely unchanged from when it was cloned from https://github.com/richard-clark/ds1337, with some minor changes to allow for better integration with the rest of the code. MRFC522.cpp and .h (and require_cpp11.h, deprecated.h) are largely unchanged copies of source code available at https://github.com/miguelbalboa/rfid, where is has been released under the Unlicense.

3.1 The Software and You, an overview for newcomers

If the software running in the ODOTS device is a recipe the 'ODOTSGeneric.ino' file holds the page you would read with instruction like 'stir' and 'turn oven on'. The actual nitty gritty of how to turn the oven on is described in ever nittier and grittier detail as you move down the levels outlined above. It is assumed that anyone looking at the software has google to hand to help with syntax comprehension, but suffice to say while Arduino can describe the 'ino' files, the rest of them have been written for C++.

The top level architecture relies on polling to satisfy functionality requirements, it checks each potential action required of it in turn and takes action where required. This is a very safe way of handling multiple tasks, if inelegant and resource intensive. The flags that notify the central thread of execution about required tasks are generally generated by the Arduino HAL or peripheral device drivers. The exception to this is the clock alarm, used to maintain an internal time reference accurate to the RTC. This is driven by an interrupt service routine which you may find in *ODOTSRTC.cpp*, which is used as the falling edge of the alarm interrupt occurs on a second boundary, which is useful to know for keeping precise time. Future development should look at moving the firmware into a more interrupt driven architecture as this will allow for power saving measures in the future, such as sleep modes, which polling has difficulty implementing.

It was intended that the libraries of the ODOTS firmware should be easily swappable to allow for hardware to be changed with a minimum of fuss when it comes to driver changes.

3.2 File by File guide

3.2.1 ODOTS.cpp, ODOTS.h

As the Library front source file there is a lot of glue code. There is also some EEPROM and ODOTS specific RFID functionality defined.

ODOTS.h

Imports: *[Arduino Builtin]* [library specific]

ODOTSRTC.h, ODOTSSerialUtils.h, BuzzerUtils.h, DS1337.h, Config.h

Declares:*[Variables, Class Instances]* [Functions]

struct time_t PunchTime,
bool CheckIsUnitConfiguredForDownload(),
bool CheckIsUnitConfiguredForClear(),
void InitRFID(),
bool CheckIfNEWRFIDCardPresent(),
bool SelectCard(),
uint16_t ParseCardIDandInfo(),
void HaltCard(),
bool ResetCard(),
bool WriteVisitStamp(struct time_t* VisitTime, uint16_t WriteBlock),
void UpdateStatusFromEEPROM(),
bool DumpCardToSerial(),
void SendRFIDModuleToSleep(),
void WakeRFIDModule(),
void SendUnitToSleep().

ODOTS.cpp

Imports: *[Arduino Builtin]* [library specific]

Arduino.h,*EEPROM.h*, MRFC522.h, LowPower.h, ODOTSRTC.h, Config.h

Declares:*[Variables, Class Instances]* [Functions]

uint8_t CheckIfTrailerClash(uint8_t Block),
MFRC522 mfrc522(SSPIN, RSTPIN),
MFRC522::MIFARE_Key key

Variables, Class Instances:

MFRC522 mfrc522(SSPIN, RSTPIN)

Instance of MRFC522 class used by functions to interact with MRFC522 library.

MFRC522::MIFARE_Key key

Declares key variables for use in MIFARE authentication.

Functions

uint8_t CheckIfTrailerClash(uint8_t Block)) [local]

Checks whether block specified is a trailer block. Currently only supports the 4 block sectors of MIFARE Classik 1k and the first half of 4k
Parameter: Block, block number to be checked.
Returns: Block , returns next block (counting up) if the block passed was a trailer block.

---

bool CheckIsUnitConfiguredForDownload()

Glue function.
Returns: Returns value of IsUnitConfiguredForDownload.

---

bool CheckIsUnitConfiguredForClear()

Glue function.
Returns: Returns value of IsUnitConfiguredForClear.

---

void InitRFID()

Initialises MFRC522 peripheral and SPI interface, also used to generate MIFARE access key, currently set to the default 0xffffffffffff.

---

bool CheckIfNEWRFIDCardPresent()

Glue code, calls mfrc522 method PICC_IsNewCardPresent() to determine whether there is a fresh card in the readers RF field. Note that an 'old' card can become new if it is left near the reader for too long.
Returns: True if there is a new card in the field, false if there is not.

---

bool SelectCard()

Glue code, calls mfrc522 method PICC_ReadCardSerial() to select a card previously detected with PICC_IsNewCardPresent. This operation includes anticollision measures to prevent multiple cards trying to talk at once.
Returns: True if card selection successful, false if it is not.

---

uint16_t ParseCardIDandInfo()

ODOTS information block interpretation. Reads the information block (as specified in Config.h) and determines the next block to be used for time visit stamps as well as whether block reuse has become necessary.
Returns: Bodged two byte value, the first byte is the next block value, the second byte = 0 if block reuse is not occuring, =1 if it is, =0xff if the card is full.

---

void HaltCard()

Calls MRFC522 methods to deselect and turn off selected Card (PICC). Must be called every time a card is selected to deselect it again.

---

bool ResetCard()

Calls MRFC522 methods to reset ODOTS PICC card state, that is resetting next-visit block pointer and block reuse flag to 0.
Returns: True if interaction successful, false if error.

---

bool WriteVisitStamp(struct time_t VisitTime, uint16_t WriteBlock)*

Generates new time stamp from RTC reference time (kept accurate elsewhere) and generates a visit stamp to be written to the card, then writes it to the ODOTS card. Handles block reuse, though this requires an extra read operation.
Parameter: VisitTime, time struct (from DS1337 source) containing time the punch occured (allowing this time to be recorded earlier and therefore be slightly more accurate).
Parameter: WriteBlock, output of ParseCardIDandInfo (see that function for full description)
Returns: True if interaction successful, false if error.

---

void UpdateStatusFromEEPROM()

Routine to update card operation flags from EEPROM memory, allowing persistent memory of device information. This means replacing the battery does not then require the device to be reprogrammed.

---

bool DumpCardToSerial()

Dumps Card memory to UART serial bus, data format specified in chapter 6.
Returns: True if interaction successful, false if error.

---

void SendRFIDModuleToSleep()

Turns RFID RF field off a good way to save power.

---

void WakeRFIDModule()

Turns RFID field on, useful if trying to detect cards.

---

void SendUnitToSleep()

Turns off most Microcontroller peripheral devices for a short period of time, a power saving tactic.

3.2.2 ODOTSRTC.cpp, ODOTSRTC.h

These files contain the ODTODS specific time functionality, interacting with the Microcontrollers internal oscillator and the DS1337 to maintain and produce accurate time stamps.

ODOTSRTC.h

Imports: *[Arduino Builtin]* [library specific]

DS1337.h

Declares:*[Variables, Class Instances]* [Functions]

void InitRTC()
void RTCEnableAlarm()
struct time_t CalculateTimeStamp()
bool CreateNewTimeStampString(struct time_t* PunchTime, uint8_t* OutputBuffer)
uint8_t UpdateRTCReference()
void RTC_SetTime(uint8_t Year, uint8_t Month, uint8_t Day,uint8_t Hour,uint8_t Minute, uint8_t Second)
bool RTC_ReadTime(uint8_t* Buffer)
bool RTCAlarm()

ODOTSRTC.cpp

Imports: *[Arduino Builtin]* [library specific]

Arduino.h, Config.h,DS1337.h

Declares:*[Variables, Class Instances]* [Functions]

unsigned long RefMillis = 0,
volatile unsigned long InterruptMillis = 0,
unsigned long RTCMillisOffset = 0,
volatile bool RTCFlag = false,
struct time_t RTCReference,,
void RTC_InterruptServiceRoutine()

Variables, Class Instances:

unsigned long RTCOffsetMillis = 0

Value of millisecond timer when last the last RTC reference was taken, time is calculated milliseconds since this reference.

---

volatile unsigned long InterruptMillis = 0

Value of millisecond timer when interrupt was called, it is assumed that the interrupt is serviced on the second edge, therefore this variable is used to determine the millisecond value of the RTC reference time (which is generated after the interrupt is serviced.

---

volatile bool RTCFlag = false

Used to enable RTC update following interrupt from RTC.

---

struct time_t RTCReference

RTC time struct, time at the last time the RTC was checked, time for a punch is calculated from this value.

Functions:

void RTC_InterruptServiceRoutine() [Local]

RTC reference update interrupt service routine, enables RTC update (and sets millisecond calculation value) when interrupt is generated by external RTC.

---

void InitRTC()

Initialises RTC, starts I2C interface.

---

void RTCEnableAlarm()

Initialises and enables RTC driven interrupt. Sets RTc alarm every minute at 0 seconds and sets internal config register to enable interrupt.

---

struct time_t CalculateTimeStamp()

Calculates a time stamp for the time of the function call. Calculates time (in msecs) since the last time the RTC time was checked then uses this to estimate present time.
Returns: Time_T struct of time.

---

bool CreateNewTimeStampString(struct time_t PunchTime, uint8_t* OutputBuffer)*

Creates an ODOTS visit record string ready for passing to a card including the Device ID number and a time string. DOES NOT write this string to the card.
Parameter: PunchTime, Pointer to a time_t struct containing the punch time recorded.
Parameter: OutputBuffer, Pointer to a char array that can be used to store the string before it is passed to the card.
Returns: True.

---

uint8_t UpdateRTCReference())

Follow up function to the Interrupt service routine, pulls time from real time clock and calculates time in milliseconds by assuming the ihnterrupt was called at Xh:Xm:Xs:0ms and then counting milliseconds since the interrupt was called.
Returns: 0.

---

bool RTC_ReadTime(uint8_t Buffer)*

Reads time from internal reference, and writes it to the buffer provided.
Parameter: Buffer, pointer to buffer where time can be stored (6 bytes long ideally).
Returns: True.

---

void RTC_SetTime(uint8_t Year, uint8_t Month, uint8_t Day,uint8_t Hour,uint8_t Minute, uint8_t Second)

Writes the provided time to the real time clock allowing growing inaccuracies to be corrected (generally through a host with access to internet UTC time). The function call assumes that it occurs on a second boundary. The RTC has no way of tracking milliseconds. TIP: to prevent daylight savings confusion it is suggested that the RTC time is set to GMT which does not change.
Parameter: Year, year (since 2000).
Parameter: Month, month (number).
Parameter: Day, of month.
Parameter: Hour, 24 hour time used.
Parameter: Minute.
Parameter: Second.

---

bool RTCAlarm()

Glue code.
Returns: Value of RTCFlag.

3.2.3 ODOTSSerialUtils.cpp, ODOTSSerialUtils.h

These Source files handle the UART communications with a host PC including ferrying configuration information and commands. The interface expected over the UART connection is described in detail in the dedicated software interface chapter, chapter **??**.

ODOTSSerialUtils.h

Imports: *[Arduino Builtin]* [library specific]

Arduino.h
Declares:*[Variables, Class Instances]* [Functions]

void SerialInit()
uint8_t CheckSerialForContents()
uint8_t SerialHandleRequest()

ODOTSSerialUtils.cpp

Imports: *[Arduino Builtin]* [library specific]

Arduino.h,*EEPROM.h*,Config.h,ODOTSRTC.h

Declares:*[Variables, Class Instances]* [Functions]

uint8_t WaitForSerialWithTimeOut(uint8_t Bytes)
,uint8_t ReadADC(),
uint8_t SendADCToSleep(),
uint8_t WakeADC()
Functions:

void SerialInit()

Initialises UART serial interface, with a baudrate specified in config.h

---

uint8_t CheckSerialForContents()

Glue code.
Returns: value returned by Serial.available(), the number of bytes sitting in the serial buffer.

---

uint8_t SerialHandleRequest()

Interpretation function, decides which functionality is desired (based on byte provided by serial link) and executes required behaviour. There are no inputs as the function takes the bytes directly from the serial buffer.

---

uint8_t WaitForSerialWithTimeOut(uint8_t Bytes)

Provides timeout functionality to the serial input readers, provides ”»Timeout” message to serial interface on timeout.
Parameter: Bytes, the number of bytes that are required to be read. *Returns:* uint8_t value ’0’ if timeout occured, ’1’ if the required number of bytes was detected before timeout, these can then be read elsewhere (THIS FUNCTION DOES NOT RETURN THE BYTES IN THE SERIAL BUFFER)

---

uint8_t ReadADC()

Turns On ADC module and enables the battery voltage reader Mosfet, waits for a bit (to let the ADC voltage settle) then reads the battery voltage. Finally turns the battery voltage reader mosfet back off.
Returns: ADC reading, first 8 MSBs as the ADC normally gives a 10 bit values (which does not fit in a byte).

---

uint8_t SendADCToSleep()

Disables ADC peripheral and updates control registers.

---

uint8_t WakeADC()

Enables ADC and updates control registers.

3.2.4 BuzzerUtils.cpp, BuzzerUtils.h

These source files manage the buzzer and LED user notifications. At the moment these are blocking processes that prevent other microcontroller operations. It is possible in the future that they may be rewritten to allow for other operations to be completed in the background. This may require a hardware upgrade.

BuzzerUtils.h

Imports: *[Arduino Builtin]* [library specific]

None.

Declares:*[Variables, Class Instances]* [Functions]

void BuzzerInit()
void BuzzerBleep()
void ErrorBuzz()
void BuzzerBlip()

BuzzerUtils.cpp

Imports: *[Arduino Builtin]* [library specific]

Arduino.h,Config.h.

Declares:*[Variables, Class Instances]* [Functions]

None.
Functions:

void BuzzerInit()

Puts buzzer pin into output mode.

---

void BuzzerBleep()

Procedure for standard operation complete notification.

---

void ErrorBuzz()

Procedure for standard error notification, for example due to incomplete PICC interaction.

---

void BuzzerBlip()

Procedure for non-standard notification, good for debugging purposes!

3.2.5 Config.cpp, Config.h

These source files contain many of the definitions used for configuring how the device operates including microcontroller pin numbers, UART characteristics, and EEPROM memory locations. It also contains some global variables that are used through many source files.

Config.h

Imports: *[Arduino Builtin]* [library specific]

None.
Declares:*[Variables, Class Instances]* [Functions]

None. (Many preprocessor instructions though.)

Config.cpp

Imports: *[Arduino Builtin]* [library specific]

Arduino.h

Declares:*[Variables, Class Instances]* [Functions]

uint8_t ODOTS_DeviceID[2],
bool IsUnitConfiguredForDownload,
bool IsUnitConfiguredForClear

Variables, Class Instances
uint8_t ODOTS_DeviceID[2]

The device ID, used to distinguish which checkpoint produced each visit stamp.

---

bool IsUnitConfiguredForDownload

Simple flag used to switch device behaviour in different functions.

---

bool IsUnitConfiguredForClear

Simple flag used to switch device behaviour in different functions.

3.2.6 DS1337.cpp, DS1337.h

For an understanding of the DS1337 source code please visit the github repository: `https://github.com/richard-clark/ds1337`, and the chip data sheet from Maxim: `https://datasheets.maximintegrated.com/en/ds/DS1337-DS1337C.pdf`.

3.2.7 MFRC522.cpp, MFRC522.h,deprecated.h, require_cpp11.h

For an understanding of the MRFC522 source code please visit the github repository: `https://github.com/miguelbalboa/rfid` which includes some very useful examples and a more in depth explanation of how the library works. Unfortunately communication with MIFARE (Classic) cards relies on two layers of protocols the ISO14443A RF interface specification and the proprietary MIFARE interface and communication specification. Trying to crack these from first principles and datasheets is a daunting task, and is one of the reasons the ODOTS design has opted for RF hardware that already has drivers written for it.
Three functions have been appended in the final lines of this source file to allow custom dumping of card data to the serial port these all have 'Data' added to the standard MRFC522 function name (e.g. PICC_Dump*Data*ToSerial()). These also have in built error detection changing the return variable from void to a bool.

3.2.8 LowPower.cpp, LowPoer.h

This is a handy library that exposes functionality of the AVR hardware abstraction layer it is used to handle the microcontrollers sleeping between poll cycles. This library has been obtained from `https://github.com/rocketscream/Low-Power`.

Chapter 4

RFID Card memory management

The RFID cards used by the ODOTS, in the current design must be MIFARE classic compatible devices. Testing has showed that MIFARE Classic 1k rather that 4k gives slightly better performance, primarily due to the time required to read and write the entire 4k card. Early in the design process every effort was made to ensure that a wide variety of cards could be used, however single standard compatibility simplifies the program required for the embedded microcontroller and works to ensure that ODOTS systems should have roughly equal performances between builds.

Practically MIFARE cards come preprogrammed with a Unique Identification Code, which should be unique to that card (shady manufacturers aside). This is generally used to identify the card and is the number declared as the ”SI Number” in the download software.

4.1 Card Memory Organisation and Usage

The organisation of MIFARE cards' memory is well documented in NXP's own documentation: `https://www.nxp.com/docs/en/data-sheet/MF1S70YYX_V1.pdf`. MIFARE Classic cards have their memory sorted into blocks, then sectors. Each block consists of 16 bytes and each sectors consists of 4 block (though the final sectors in a 4k card have 16 blocks). In each sector one block (the 'trailer') is used to store access keys and information and so is unusable for ODOTS purposes.

The ODOTS does not use the lowest sector (sector 0) for visit record storage. Block 0 in this sector is used for the Card's UID and other information and is written to by the manufacturer, it should not be modified. Block 1 is used by the ODOTS to store race information, though this consists of the 'next block to punch' value in the third byte of this block and a 'second use of block' flag (0xff) in the fourth byte. The value of 'next block to punch'is iterated by each checkpoint visit record write operation, the 'second use of block' flag is discussed below. Block 2 is not used. This leaves a lot of spare memory that is not currently used, however the intention is to leave these blocks alone for the time being as they will allow additional functionality in later designs.

The remaining non-'trailer' blocks on the card are used to store visit records, information dumped by checkpoints including a timestamp and the code of the checkpoint. Each visit

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Contents:	DeviceID[0]	DeviceID[1]	0x55	Hour	Minute	Second	Millisecond»8	Millisecond%255

Table 4.1: Bytewise structure of ODOTS visit record

record is 8 bytes long allowing for two per block. The structure of the ODOTS visit record is described by table 4.1. The ODOTS has the ability to time to the accuracy of about 5ms, though this functionality is not supported by the current competition management software. Byte 6 and 7 of the visit record contain the milliseconds of the time stamp, unfortunatly the 1000 milliseconds a second do not fit into a byte, therefore the value has been stored over two! The Device ID is similarly split.

As mentioned previously each MIFARE block can store two ODOTS visit records. Writing to the second spot requires an extra read operation (as the whole block must be written to at once, and ideally the first record should be maintained), therefore the ODOTS first fills up all the 'first' spots in the first 8 bytes of each available block. Once these are all filled the value of 'next block to write' is reset to its lowest value and 'second use of block' flag is set, indicating to future checkpoints that they should be using the second slot available in each block.

4.2 .ODOTSRAW Records

ODOTSRAW files contain the downloaded contents of a card, ready for processing and passing off to MEoS or other competition management software. They take the form shown below (in plain text):

Card UID: 81 E6 39 2F
Card SAK: 08
PICC type: MIFARE 1KB
61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
... .

All values are in HEX, except for block numbers which are in decimal form (61 in the example shown). All blocks apart from trailer blocks should be included in the record, though occasionally the block record will be replaced with an error message (this should no longer happen). The second and third lines are not currently used, however the provide useful information about the card and can be used to detect chen a non standard card has been used.

Chapter 5

Host Software Developers Guide

This chapter is developer documentation for the software required to run on a host computer, currently targeted for Windows. This software manages communications with an attached ODOTS device, whether this is for device configuration or card download operations. The User Interface of this software is currently text based (to reduce computer operation overhead). Usage instructions and tutorials for this interface may be found in part III. The host software has been implemented in Python 3 due to its high level of readability and functionality provided by external libraries.

Competition management and results calculation is handled by MEos, developed by Melin Software, which can be found at `http://www.melin.nu/meos/`. The interface to this software is straight forward over a TCP link, provided by 'SendPunch' a Java program modified from an example provided by Melin. MEos has been used as the interface provided by 'Send Punch' can be operated with a single command.

Developed code for the host software is made of four custom objects, each a unique instance of one of four classes. It has two processes, one to run the User Interface and another to handle the serial interface and manage asynchronous card downloads. The scripts for the Host software can be found in four files, ODOTSInterface.py, ODOTSFileParser.py, SerialUtils.py and TimeUtils.py, the final three will generally be stores in "./Modules".

5.1 File by File Guide

5.1.1 ODOTSInterface.py

This script contains the main execution code for the ODOTS interface. It creates two processes, a user interface process that takes input from the command line, and a serial interface thread used to communicate with the serial port and automatically forward downloads from an ODOTS card to competition management software. Interprocess communication is achieved using two FIFO queues, detailed in chapter 6. The serial interface consists mostly of glue code to pass user commands from the user interface off to the serial port. About the only non-trivial action is 'Read Unit Info' which strings several ODOTS unit information requests together before providing the information to the user in a readable format. Full usage of the ODOTS interface is detailed in Part III.

5.1.2 ODOTSFileParser.py

This module contains a class used to take the ODOTSRAW files and send them off to the TCP socket attached to MEoS.
Imports: *[Python Official]* [project specific]

subprocess,*os*.

Declares:*[Variables, Classes]* [Functions]

FileParserTCPWriter: __init__(), ProcessRecord(), ParseFile(), CalculateCardID(), ReadPunches(), ReadTimeLine(), InterpretEntry(), SendToSocket().

Classes:
FileParserTCPWriter

Base class for the module, allows for persistence of settings without relying on global variables.

Functions:
FileParserTCPWriter.__init__()

Initialises File Parser object, and takes configuration arguments, changing TCP socket settings has no effect at the moment.

FileParserTCPWriter.ProcessRecord(filename, DeleteFileAfterUse = None)

Parses ODOTSRAW file into time stamps and checkpoint codes, and sends the information off to the TCP socket. The all in one function.
Parameter: Filename, name of file containing card data.
Parameter: DeleteFileAfterUse, boolean, no current functionality.

FileParserTCPWriter.ParseFile(self,filename,DeleteFileAfterUse = None)

Parses ODOTSRAW file into string that can be sent to TCP socket.
Parameter: Filename, name of file containing card data.
Parameter: DeleteFileAfterUse, boolean, no current functionality.
Returns: String to be sent to TCP socket.

FileParserTCPWriter.CalculateCardID(self, ReadString)

Calculates Card ID number from uid (which is in hex) this is taken directly from SerialUtils.py script.
Parameter: ReadString, UID line of ODOTSRAW file.
Returns: Integer value of UID.

FileParserTCPWriter.ReadPunches(self,file = None)

Reads ODOTS RAW file and extracts all punch records, dumps empty records and orders all punches in order they were punched.
Parameter: file, file object of ODOTSRAW file to be read.
Returns: List of strings, each string being a visit recording.

FileParserTCPWriter.ReadTimeLine(self,LineFromFile)

Reads line from ODOTSRAW file and returns visit records contained within.
Parameter: lineFromFile, string representing line read from ODOTSRAW file
Returns: Two element array containing the two visit record ([NONE,NONE] for record spots where there is no visit record detected.)

FileParserTCPWriter.InterpretEntry(self, VisitRecordEntry)

Reads record from ODOTSRAW file and converts into program friendly [ID, Time] arrays.
Parameter: VisitRecordEntry, string representing Byte Literals of visit record.
Returns: Two element array containing the two visit record ([NONE,NONE] for record spots where there is no visit record detected.)

FileParserTCPWriter.SendToSocket(self, String=None)

Calls Java *SendPunch.java* used to interact with MEoS TCP socket, and sends string provided (ideally formatted by self.ParseFile) *Parameter:* String, string to be sent to the TCP socket.

5.1.3 SerialUtils.py

This Module provides serial interface functionality, managing communications with the ODOTS device while providing an abstracted interface to the rest of the software.
Imports: *[Python Official]* [project specific]

Time, *Serial* (PySerial),*serial.tools.list_ports*,*os*,TimeUtils.

Declares:*[Variables, Classes]* [Functions]

ODOTS_Serial_Interface: __init__(), ListAvailablePorts(), SelectPort(), OpenPort(), ClosePort(), ReadTime(), ReadSystemTime(), WriteTime(), ReadDeviceID(), WriteDeviceID(), ReadDeviceMode(), SetDeviceMode(), ReadFirmwareVersion), ReadHardwareVersion(), InterceptDownloadMessage(), InterpredDownloadandWritetoFile(), CheckifHex(), InitialiseCommand(), NeatenPortList(), PollInputPOrtforDownload.
Main

Classes

ODOTS_Serial_Interface

Base class for the module, allows for persistence of serial ports without using global scope variables.

Functions:

ODOTS_Serial_Interface.__init__(FileWritePath="",CurrentWorkingDirectory=os.getcwd(),ConfigObject=None,Verbose = False)

Initialisation function for ODOTS Serial Interface Class, takes a number of configuration inputs. ConfigObject is an unimplemented entry point for a configuration object allowing for straightforward interface changes. Initialises Serial Object.

ODOTS_Serial_Interface.ListAvailablePorts()

Uses SerialFind tool to list comports, then returns a list of available serial ports with additional details to allow for the ODOTS device induced serial port to be identified.
Returns: List of available serial port details.

ODOTS_Serial_Interface.SelectPort(Port)

Selects port from port list to be used for serial communications, it does not open it. Selected port is retained in class scope variable for use later.
Parameter: Port, index in port list of the port to be selected.
Returns: Name of ActivePort.

ODOTS_Serial_Interface.OpenPort(ConfigObject=None)

Opens port for serial communications.
Parameter: ConfigObject, provides port parameters, these revert to default values if not provided, baudrate = 9600, timeout = 1 second.

ODOTS_Serial_Interface.ClosePort()

Closes currently open serial port.

ODOTS_Serial_Interface.ReadTime()

Reads time from attached ODOTS device.
Returns: TimeString [Year,Month,Day,Hour,Minute,Second]

ODOTS_Serial_Interface.ReadSystemTime()

Calls TimeUtils.GetCurrentTime and returns output.
Returns: TimeString [Year,Month,Day,Hour,Minute,Second]

ODOTS_Serial_Interface.WriteTime()

Writes current system time (on second boundary) to attached ODOTS device.
Returns: True if successful, False if timeout detected.

ODOTS_Serial_Interface.ReadDeviceID()

Reads the DeviceID of attached ODOTS device.
Returns: DeviceID of attached device (bytes object)

ODOTS_Serial_Interface.WriteDeviceID(NewID=None)

Writes new ID to attached ODOTS device (changing it)
Parameter: NewID, two byte object representing desired device ID (MSB fist), None if no change desired.
Returns: Old Device ID of attached ODOTS device.

ODOTS_Serial_Interface.ReadDeviceMode()

Reads the Operation flags from the attached ODOTS device.
Returns: [Bool,Bool], [DownloadEnabled,ClearEnabled]

ODOTS_Serial_Interface.SetDeviceMode(DownloadEnabled,ClearEnabled)

Writes the operation flags on the attached ODOTS device.
Parameter: DownloadEnabled, boolean true if the Download functionality is to be expressed.
Parameter: ClearEnabled, boolean true if the Clear functionality is to be expressed.
Returns: [Bool,Bool], [DownloadEnabled,ClearEnabled], [None,None] if write failed.

ODOTS_Serial_Interface.ReadFirmwareVersion()

Reads the firmware version of the attached ODOTS device.
Returns: bytes representaiton of the firmware version, false if serial timeout occured.

ODOTS_Serial_Interface.ReadHardwareVersion()

Reads the hardware version of the attached ODOTS device.
Returns: bytes representaiton of the hardware version, false if serial timeout occured.

ODOTS_Serial_Interface.InterceptDownloadMessage(intercept)

Identifies when the ODOTS device has attempted to start a card data download, calls *ODOTS_Serial_Interface.InterpretDownloadandWritetoFile()* if download detected, and returns result, generally the file name that the downloaded data has been stored in.
Parameter: byte read from the serial port.
Returns: False if no download detected, otherwise result of *ODOTS_Serial_Interface.InterpretDownloadandWritetoFile()*.

ODOTS_Serial_Interface.InterpretDownloadandWritetoFile()

Interprets download from ODOTS and writes result to a file for later use, dumps the data and deletes the file if a card error is detected.
Returns: False if error detected, otherwise file name of file used to store card data.

ODOTS_Serial_Interface.CheckIfHex(TestChar)

Checks to see if the byte of 'TestChar' could be used to represent a hexadecimal digit.
Parameter: TestChar, byte to be tested (generally the one just read from the serial port)
Returns: True if the character passed could be a hex character, False otherwise.

ODOTS_Serial_Interface.InitialiseCommand(CommandChar)

Handles initial exchange of command characters over serial interface with attached ODOTS device, provides timeout handling.
Parameter: CommandChar, single character representing initial transmission in serial interface interaction.
Returns: True if exchange successful, False otherwise.

ODOTS_Serial_Interface.NeatenPortList(PortList)

Replaces certain PID and VID serial port values with more intelegible ones where they match known devices, at the moment Arduino Leonardo and Uno devices.
Parameter: PortList, list of available ports.
Returns: PortList with certain entries replaced.

ODOTS_Serial_Interface.PollInputForDownload()

Reads a byte from the serial port to try and detect card data downloads, allows for this to happen when there is not command ongoing.
Returns: File Name of successful download, or False if no download was detected.

Main

Simple script that runs if Serial Utils is used as the application entry point, selects first available serial port, reads device time, then enters loop where it can intercept card data downloads.

5.1.4 TimeUtils.py

This module provides access to the system time to the serial interface.
Imports: *[Python Official]* [project specific]

Time

Declares:*[Variables, Classes]* [Functions]

ReturnCurrentTimeOnSecondEnd(), GetCurrentTime()

Functions:

ReturnCurrentTimeOnSecondEnd()

Does what is says, returns the current time (in GMT) on a second end, the timing relative to a second is useful for maintaing sub-second precision with the ODOTS.
Returns: TimeStruct representing current time (according to system, so as accurate as the last internet time update) in GMT.

GetCurrentTime()

Glue code around previous function, picks out elements of Time struct returned by GetCurrentTimeOnSecondEnd() and interprets them so that they may be used by serial interface. *Returns:* List representing current time (according to system, so as accurate as the last internet time update) in GMT.

Chapter 6

Interface Specification

This chapter is intended to provide a central reference for the various interface that exist between asynchronous entities in the ODOTS from the embedded microcontroller to the socket handler which manages communication with the competition management software. There are of course many more interfaces in the ODOTS design, these are generally between software being used by the same entity and have far better inherent clear and coherent expression in the source code. The interfaces here are not necessarily implemented between programs utilising the same programming language or running on the same hardware.

6.1 Serial Communication Interface

The Serial communication interface describes communication between the embedded software running on the microcontroller in an ODOTS unit with the software running on a host computer. It should be provided by a UART over USB connection with a baudrate of **9600** symbols a second, though this can be confirmed by checking *Config.h*. Serial communications are handled by both ends of the current implementation on a byte by byte basis with most significant byte first and ASCII encoding of characters though not of numeric values. Final specifications of the exact characteristics of the USB manager PID,VID and name will be provided in the hardware description.

Communication is generally host initiated, with a single capitalised character sent over the serial link. The ODOTS unit will then generally reply with a lower case version of this command character after which communication may proceed as dictated by the command issued. The exception to this is card download data transfers which may be initiated by the ODOTS unit when there is no ongoing interaction and which must be intercepted by the host software.

To prevent the microcontroller from locking up if serial communications are interrupted all reception of serial characters is completed with a timeout, currently set to 1.2 seconds. If the timeout is triggered the microcontroller drops the interaction and continues as normal with its other functions.

Table 6.1 specifies each command and the data to be transferred. Interpretation of version numbers is explained in section 1.2. A full description of the card data dump format can be found in chapter 4.

Command	Device ID, Request and Update
Host Sends:	**ODOTS Unit Sends:**
->'A' - - DeviceID, new or old	- 'a'<- DeviceID (2 bytes)<- -
Command	Operation Mode Configure
Host Sends:	**ODOTS Unit Sends:**
->'B' - - Operation Flags, new or old	- 'b'<- Operation Flags<- Bitwise:RFU |RFU |RFU |RFU |RFU |RFU |ClearIfSet |DownloadIfSet<- -
Command	Firmware Version Read
Host Sends:	**ODOTS Unit Sends:**
->'C' - -	- 'c'<- VersionNumber (Literal)<-
Command	Hardware Version Read
Host Sends:	**ODOTS Unit Sends:**
->'D' - -	- 'd'<- VersionNumber (Literal)<-
Command	Clock Update, Set
Host Sends:	**ODOTS Unit Sends:**
->'E' - ->Year ->Month ->Day (of month) ->Hour (24hr) ->Minute ->Second	- 'e'<- - - - - - -
Command	Clock Read
Host Sends:	**ODOTS Unit Sends:**
->'G' - - - - - - -	- 'g'<- Year<- Month<- Day (of month)<- Hour (24hr)<- Minute<- Second<-
Command	Device Battery Voltage Read
Host Sends:	**ODOTS Unit Sends:**

->'H' - -	- 'h'<- Battery Voltage (integer byte 0-255)<-
Interaction	Dump Card Memory
Host Sends:	**ODOTS Unit Sends:**
- - - - - -	'Y'<- Card UID line<- Card SAK line<- Card Type line<- Card Memory Block (multiple)<- 'Z'<-

Table 6.1: Serial Interface Command Interactions

6.2 UI to Serial Manager Interprocess Interface

This interface allows the asynchronous operation of the User Interface and implementation of the ODOTS configuration and management. The interface exists through two queue objects, currently using the *multiprocessing.Queue*. The interactions over this interface consist of passed messages, stored in list objects (often with only a single element) in plain text that are interpreted at either end. All interactions are initiated by the user interface process which loads a message into the UI to Manager queue, this should invoke some form of action in the manager process which may result in it loading messages into the Manager to UI queue. The invokation messages and the actions they invoke have been listed below.

- **"ChangeStateToStandard"**, manager requests serial interface to update device operation flags to result in standard ODOTS device operation (used for checkpoint units).
- **"ChangeStateToClear"**, manager requests serial interface to update device operation flags to result in clear ODOTS device operation, where card interactions result in the card being reset.
- **"ChangeStateToDownload"**, manager requests serial interface to update device operation flags to result in download ODOTS device operation, where card interactions result in the cards memory being copied and sent to a host device.
- **"ChangeDeviceID"**, manager triggers ODOTS device ID rewrite, with the new ID value being provided in the next message provided.
- **"SyncDeviceTime"**, manager requests serial interface to update device time (writing to the RTC too) to match system time, currently configured to write GMT.
- **"DumpDeviceInfo"**, manager requests serial interface to read device information from attached ODOTS device. The UI expects a list returned in the Manager to UI queue containing the relevant information.

- **”ListRequest”**, manager requests serial interface to list available serial ports. The UI expects a list to be returned in the Manager to UI queue containing details of all available ports.
- **”ConnectToPort”**, manager requests serial interface to connect to a serial port indicated by a subsequent recieved message containing the desired ports position in an earlier ’listrequest’ response. Manager responds with a simple success/failure message placed in the Manager to UI queue.
- **”ClosePort”**, manager requests serial interface to close current serial port.

6.3 ”Send Punch.java”

This interface has been included despite all interactions over it being completed in a simple function call. As mentioned elsewhere SendPunch.java is a modified version of an example provided by Melin Software for use with their MEoS competition management software. The program has been modified to take its input as input arguments, rather than requiring interaction.

SendPunch.java USAGE: java SendPunch A B C D E.

- **A**- switch C for entire card download, P for single punch download
- **B**- number of punches included (integer) (best to set to ’1’ for single punch registers)(INCLUDES FINISH)
- **C**- Card ID (integer)
- **D**- PunchID String (string of integers, split with ”,”) (codes of controls punched)
- **E**- Punch Time string (string of time stamps HH:MM:SS split by ”,”) NOTE: Finish time stamp added on last, does not need to have an accompanying punch id. It is best if the start does have a punch ID

Chapter 7

Hardware

The hardware for the ODOTS proof of concept is far less complex than its software. In this case the hardware refers to the physical ODOTS device that would be placed at a checkpoint, the start, the finish or at a competitor information download station. For the proof of concept all components used are through hole to make soldering straight forward, even for users with little soldering experience. This means that the design is quite large for the number of functions it performs.

A critical design choice for the proof of concept has been to use a premade development board to fulfil RFID functionality. There were two reasons for this, first the RFID package chosen does not have a simple-to-solder package option and many of the antenna matching components must also be very small surface mount components. Second the design effort required for a standard-compliant 13.34MHz antenna and transceiver is non-trivial. Using a pre-made board sidesteps these difficulties and also has the benefit of also being unlikely to breach the band plan at the RFID frequencies, so we can play nicely with other spectrum users. Note, even with this assumed conformity the ODOTS will need to be tested for EMI at some point in the future if some version of it is to be brought to market.

The hardware design can be easily viewed and edited using the KiCad design files available in the ODOTS-Release repository using KiCad, which is free to use.

7.1 Element by Element

7.1.1 The Mounting Board

The mounting board provides an attachment point for the electronic components. It is recommended to use the provided PCB design, but the ODOTS hardware can be implemented on a stipboard.

7.1.2 The RFID module

The RFID module supplies the RFID functionality to the system. It has a pinout containing pins to establish a serial interface with the central microcontroller and inputs for the power and ground rails. The board uses an MFRC522 chip and has integrated oscillator, matching and antenna components. The antenna stands out on the module PCB as it is the 'empty' square at the opposite end from the pinout.

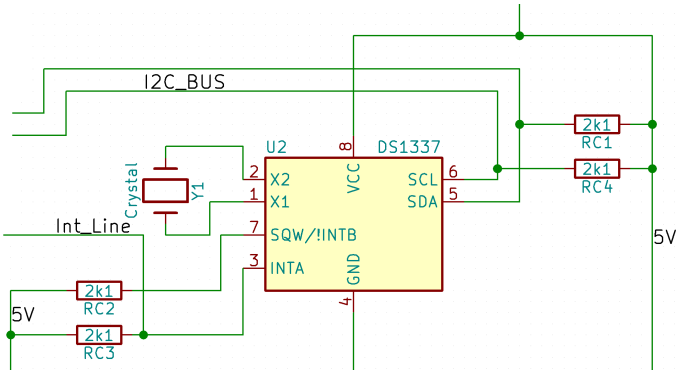

Figure 7.1: Clock schematic with busses and rails labelled.

7.1.3 The Battery

In the proof of concept the battery has been implemented with a cheap phone power bank. It consists of two components, an 18650 Lithium Ion power cell and a power interface board. The power cell provides over two amp hours of capacity, the interface board allows the battery to be charged from any standard USB charging port. The board also provides a regulated 5V output that is used to provide power to the rest of the ODOTS unit.

7.1.4 The Microcontroller

The Microcontroller performs coordination and calculation tasks in the ODOTS. In the proof of concept it is an AtMega 328p chip in a PDIP package, mounted on the main mounting board via a socket. The purpose of the socket is to allow chips to be quickly changed in case of chip failure and to protect them from the potential damage while being soldered. The Microcontroller requires a single external device to operate, a 16MHz crystal used to generate the clock signal for its digital systems. The two pins of the crystal are connected to ground through a single 22pf ceramic capacitor each.

The microcontroller is powered from the 5V rail.

7.1.5 The Clock

The clock is an external real time clock chip designed to keep time with a greater accuracy than the oscillator of the microcontroller. It is a DS1337 chip mounted via a socket for the same reasons as the microcontroller. Figure 7.1 shows an extract from the design schematic containing the clock. The clock requires a single crystal to provide a clock signal to the internal circuitry and pull up resistors are required on the interrupt and data bus pins. Interrupt Pin A is used to send alarm signals to the microcontroller. The data bus uses the I2C bus standard. The clock is powered from the 5V rail, future designs will include a backup power supply for the clock to allow it to keep time even when the main battery is unplugged.

7.1.6 The Buzzer

The buzzer is used to send audio prompts to the user notifying them of certain functionality fulfilments. The buzzer is a piezoelectric device using quartz crystals to move a diaphragm

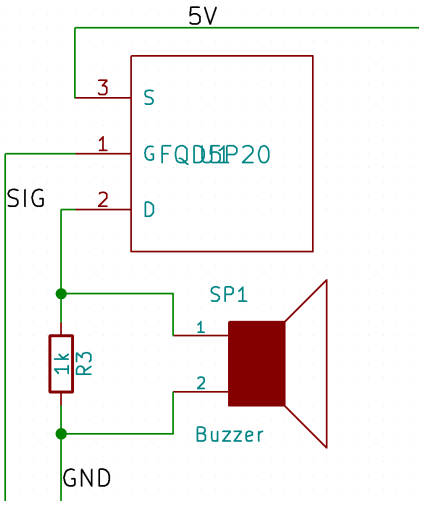

Figure 7.2: Buzzer schematic with busses and rails labelled.

and create noise. The distortion of the crystals, and therefore the displacement of the diaphragm, is directly proportional to the electric field through them. A mosfet is used to maximise the difference in field strengths that can be exerted across the buzzer (the power rail to shorted to ground). A P-channel mosfet has been used as it gets used elsewhere in the design. The 1k Ohm resistor is used to provide inductive current a safe discharge path during mosfet switching. Figure 7.2 shows an extract from the design schematic containing the buzzer and associated components.

7.1.7 Batter Voltage Reader

The battery voltage reader is designed to allow the Microcontroller to read the raw output voltage from the battery cell. The cell is expected to be a lithium ion 18650, with a maximum output voltage of just over 4.3V. Figure 7.3 shows an extract from the design schematic containing the battery voltage reader. The mosfet is used to prevent current flowing when the voltage is not being read to save power. The voltage is read from the centre of a voltage divider that is used to turn the maximum expected voltage into something that the internal ADC reference of the microcontroller can be compared with (1V).

7.1.8 Other Peripherals

The other peripheral (to the microcontroller) components used are a pair of LEDs and a Reset button. Figure 7.4 shows an extract from the design schematic containing the peripheral devices.

The LEDs are driven directly from the microcontroller digital outputs. In the design they are red LEDs and have appropriately sized current limiting resistors.

The Reset button provides reset functionality for the microcontroller and RFID module. This signal is active high so a pull up resistor is used to maintain 5v for this signal when the button is not pressed.

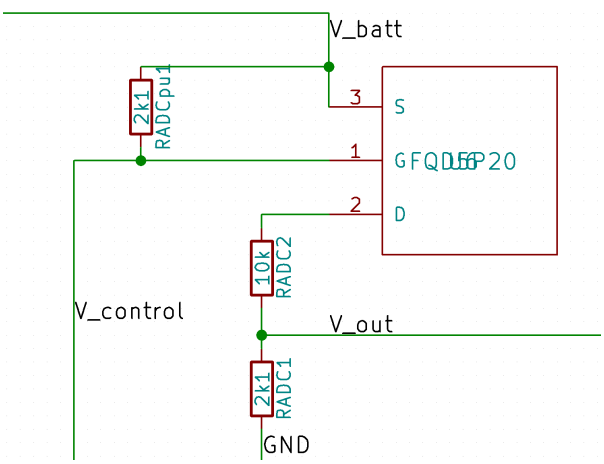

Figure 7.3: Battery Voltage Reader schematic with busses and rails labelled.

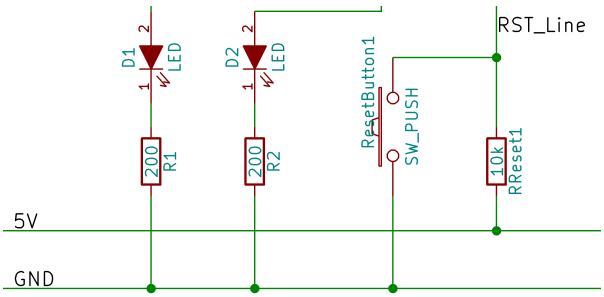

Figure 7.4: Peripheral schematic with busses and rails labelled.

7.1.9 Power Regulators

With many components requiring stable power supplies, while also having bursts of high power usage the power regulator in the design has been under loaded, and reservoir/stabilisation capacitors have been provided for both power rails in the design (5V and 3.3V). The 3.3V rail has its power provided from a L78L33ABZ 3.3V linear voltage regulator. The 5V rail does not require a voltage regulator (this is provided by the battery and battery controller). Tests of system functionality with a with a 5V voltage regulator anyway, for power rail stability and protection from over voltage, showed that it drew too much power and dropped the system power efficiency significantly. The capacitors are intended to provide small reservoirs of charge for transient spikes in power demand while keeping the power supply voltage stable and providing a short to ground for RF leakage from the RFID module.

7.2 USB Connection

The microcontroller used cannot natively interface with a USB port and requires a bridge of some sort to establish a serial communication link with a PC. While it is plugged into the UNO development board the board provides this functionality. To link a Veroboard or PCB unit to a computer there is a dedicated set of header pins on the ODOTS unit board which can be wired to a USB bridge. In the case of an ODOTS a spare UNO board with its microcontroller dismounted (likely the one used to flash the microcontrollers) can be used as a bridge with the TX, RX 5V and ground pins marked on the board.

Chapter 8

Known Bugs

This chapter lists the known bugs, useful to watch out for.

- There appears to be an issue with the way the device calculates the month from the RTC, adding a random 10 or so to the correct value. Given this bug has no immediate impact on the system behaviour it has been left for the time being.
- The breadboard prototype exhibits a number of issues after sustained operation, for example the Alarm signal no longer triggering an interrupt. These issues have not manifested in the soldered designs.

Chapter 9

Outstanding Development Tasks

This chapter lists outstanding tasks, vital for the next step in the development of the ODOTS.

9.1 Software

- Rewrite firmware for interrupt driven behaviour, allowing better power efficiency.
- Find other competition management program interfaces, break dependence on MEoS.

9.2 Hardware

- Move PMOS circuits back to NMOS, while PMOS has worked for the proof of concept it is far more expensive than NMOS and is actually based on a concept that was not used. (It also improves the battery voltage reader performance.)
- Move to SMD components for reduced cost and footprint.
  - Investigate component changes, ideally like for like swaps can be used.
  - Design new PCB.
  - Update documentation and firmware to power with new design.
- Change Microcontroller to something more power efficient.

Part II

Build Guide

Chapter 10

Build Guide Introduction

This part is intended to guide the manufacture of an ODOTS. All designs can be found in the ODOTS-release GitHub repository (`https://github.com/ljones278/ODOTS-Release`) sub folder *Hardware* contains the hardware designs, the sub folder *Software* contains both the embedded firmware and ODOTS interface software the sub folder *Admin* contains other items such as the bill of materials. The following chapters of this part will cover the methods of construction and installation of an ODOTS.

- Chapter 11 outlines the various components required, and some possible sources of components.
- Chapter 12 outlines some of the construction methods for building the hardware of an ODOTS unit.

Chapter 11

Acquiring Components

During the development of the proof of concept hardware was obtained from reputable sources who could deliver quickly, or from sources that could provide replacements quickly. Being slightly more adventurous with component aquisition will significantly cut costs but may incur increased lead times and potential reliability issues. These acquisition decisions are left to the discretion of the developer. Table 11.1 lists the components required to build a single ODOTS unit, along with the price and source of the component during development. The raw total cost per unit (during development) was **£21.18**, though this does not include the wasted costs associated with minimum purchase quantities.

Component	**Description**	**Quantity per unit**	**Development Source**	**Stock ID**	**Cost**
Microcontroller	AtMega328p	1	RS	131027	£1.54
Clock	DS1337	1	RS	7860749	£2.12
MOSFET (P-Channel)	FQD5P20	2	RS	8628782	£0.18
3.3v regulator	L78L33ABZ	1	RS	6869552	£0.25
Microcontroller Socket	28 DIP socket	1	RS	7022726	£1.27
Clock Socket	8 DIP socket	1	RS	7020654	£0.39
Reset Button	Basic Push Button	1	RS	1359467	£0.10
RTC Crystal	Crystal 32.768KHz 3x8mm	1	RS	5476985	£0.28
AtMega Crystal	16MHz Quartz Crystal	1	RS	1441039	£0.21
Microcontroller Oscillator Capacitor	22pf Cap	2	RS	6994872	£0.13
PowerRail Capacitors	10uF Electrolytic Capacitor	4	RS	7111425	£0.03
LED	Red 5mm LED	2	RS	1278393	£0.05
Buzzer	Buzzer Piezo,3V,60dB,Radial	1	RS	1347307	£0.29

Component	Description	Quantity per unit	Development Source	Stock ID	Cost (single)
LED Resistor	200 Ohm Resistor	2	RS	1650814	£0.03
2k1 Resistor	2k1 Ohm Resistor	6	RS	6833473	£0.01
10K resistor	10k Ohm Resistor	2	RS	1372750	£0.01
Buzzer Resistor	1k Ohm Resistor	1	RS	1650224	£0.05
Battery	Phone power bank	1	Poundland	n/a	£2
Enclosure	TBA	1	?	?	£4.00
PCB	Gerbers Provided online	1	JLCPCB	(or equivalent)	<£1
RFID Board	MRFC522 dev board	1	Amazon	`https://www.amazon.co.uk/dp/B074S9FZC5/ref=pe_3187911_189395841_TE_dp_1`	£3.70
Solder	various uses	?	?	?	£1
Wire	various uses	?	?	?	£1

Additionally you may find it helpful to get an Arduino Uno Development board (with PDIP chip, the big one) as this can be used to burn in bootloaders, quickly upload programs, act as a serial bridge as well as providing an AtMega chip with a pre-burned bootloader.

11.1 Sourcing the PCB

One of the most noticeable elements of any electronics project is the mounting. Generally for more reliable projects this will consist of a board (or boards) and generally this will be a Printed Circuit Board or PCB. The ODOTS proof of concept design has two options, either a PCB or a design made for Strip (or Vero) Board. Of the two the PCB will offer easier construction less prone to errors, will be smaller and look better. Strip board required more soldering (and more chance to create an accidental short and is prone to mistakes (wires going in wrong holes, misplacing components e.c.t.). Having said this PCBs, if sourced from China for maximum cost saving, can have large lead times which may not be ideal.

PCBs can be readily sourced from many Chinese manufacturing plants, during the proof of concept production JLCPCB was used. There are manufacturers in Europe too, however you often end up paying for an increased quality of workmanship that is not necessary for the ODOTS. The PCB CAM files (called Gerbers) will be available on the ODOTS-Release GitHub repository these are required when you put in a PCB order. The PCBs are 2 layer and ideally

you should select the RoHS compliant options for solder and solder masks.

11.2 Sourcing the RFID Board

The MFRC522 has been blessed with the attention of several mass producers so development boards with integrated antennas and matching components. This is brilliant as these can be tricky to make properly, and then entire board can be found for less that £5 from many sellers. The seller used in the Proof of Concept production was simply the first found on Amazon for a reasonable price and with a distribution centre in Europe. Unfortunatly there have been reports of fraudulent sellers who sell either sub-standard chips or dummy packages (with the box but no silicon inside), so some discretion when bargain hunting is advised.

Chapter 12

Building Hardware

12.1 Soldering Components

All components in the proof of concept design are *through hole*, their pins generally consist of long wires that will pass the whole way through the mounting board. Generally you will need to solder the component to the board on the opposite side to the component. Some points on soldering to those who may be inexperienced, you will be able to find far better tutorials on the web if needed:
Step by step overview of soldering to a PCB:

1. Turn the soldering iron on, and wait for it to get to temperature, you can test this by trying to melt solder against the tip. Once the solder start melting give the iron another bit then begin soldering.
2. Place component on PCB, ideally bending protruding legs on reverse side to hold the component on the board.
3. Place the PCB in a spot where it won't move, reverse side up.
4. Press the soldering iron tip onto join between component leg and the PCB pad.
5. Feed in solder directly on to join. It may take a while for the joint to get up to temperature. If it is taking too long wetting the iron with solder can help heat conduction.
6. Once the solder melts onto the joint remove the iron and the solder wire and let the joint cool.
7. You may find it necessary to remove solder from the iron, a wet sponge works well as a wiping implement, but generally held static on a work surface for the iron to wipe against, to avoid accidental burning.
8. You may find it necessary to remove solder from the join. A solder sucker can be used for this, or you can use the iron itself to draw solder away, this works well if it has just been cleaned as the surface tension of the liquid solder will draw it up onto the iron.
9. Trip excess leg protruding from the join using a pair of snippers.

Some other tips on soldering:

- Soldering is about connecting two components by melting solder so that it joins them then letting the solder refreeze.
- Getting both surfaces to be joined hot is key to this, solder has a hard time flowing onto cold surfaces and will tend to ball up instead. Flux can help flow, but should be unnecessary for the large, stable, joints in this project.
- Having said that, getting things too hot can damage them, the PCB contacts will come off if you melt the Adhesive holding them to the board, this will make soldering the connection supposed to be made to them impossible.
- Hot things are hot after heating, especially the soldering iron. Burns are a hazard, be aware that the soldering iron tip stays how for quite a while after it has been turned off.
- Wall plug irons can be problematic with little regulation and poor power draw they can take a while to heat up, and will change temperature quickly during use. Basic soldering stations with inbuilt transformers and iron holders can be found for less than £20, use this project as an excuse to get one if you do not already.

Component positions will be marked on the PCB, including values for resistors and chip numbers for the integrated circuits. If you are using strip board these will instead be indicated by the design, you may find it helpful to mark up the positions of components on the strip board yourself. The suggested soldering order of components is:

1. RFID board connecting wires (and intra board wires, if using stripboard).
2. The two DIP sockets (ensuring that their notches are facing the right way!).
3. Resistors. Pliers can be used to pull their legs all the way through the board to ensure the resistor is held tightly.
4. The Serial Header Pins.
5. Ceramic capacitors and oscillator crystals. The RTC crystal should be left with a bit of leg above the board so that it can be bent over and lain against the board surface.
6. P-Mosfets, ensure that the low threshold FET is used for the battery indicator. If using the first PCB design you will now need to solder in an extra wire connecting the source pin of the buzzer MOSFET to a 5V trace. The most accessible one is the 5v pin of the serial header nearby, soldering to the reverse end of this pin prevents this fix from interfering with normal operation.
7. The buzzer and switch. Check to ensure that you solder the shorted sides long ways - the pins closes to the microcontroller should not have a connection to the pins further away unless the button is pressed. (You can check with a continuity meter)
8. The 3.3V Voltage regulator, this is actually marked incorrectly on the board, the flat side should face away from the microcontroller.
9. The polarised capacitors, these should be soldered with their negative pin (marked on the capacitor package) in the white side of their footprint.

10. The LEDs, soldered so that they are on the reverse side of the board. These are also polarised components, when both placed through the holes their longer legs should be in the middle. Think AFL goal posts rather than Black Gate of Mordor.

After completing soldering you can inset the two DIP chips, ensuring that their top (marked) ends point towards the notch in the DIP sockets.

12.1.1 Using Stripboard

If you are using strip board component positions will be indicated by the design, you may find it helpful to mark up the positions of components on the strip board yourself. The design can be found on the ODOTS-Release github repository Using strip board also requires you to break the traces on the reverse side at certain points, this can be done by hand with a 4mm drill bit. In general you will need to break the trace at the end of each connection.

Due to the marginal cost benefit and large manufacturing complexity penalty it is not recommended to use strip board, it is definatly worth the delay of shipping from China!.

12.2 Wiring it together

With the main board complete connections to the rest of the ODOTS hardware is now required. It is recommended that you embed the firmware before completing this step.

The rfid module wires can be soldered into their marching mounting holes (the wire from 'gnd' should be soldered to 'gnd'). IN this case 'VCC' is equivalend to '3.3V' and on some boards the slave select pin ('SS') is marked 'SDA'.

12.2.1 The Battery

If you are using a cheap power bank battery then the USB cable proided can be used to provide a simple interface to connect the battery to the board.

1. Cut the USB cable about halfway along and discard the microUSB end.
2. Expose lengths of each of the internal wires (you may need to strip the wires back a little), there should be two wires. Determine which of the wires is the ground wire, a good trick is to open up one of the ower banks, plug the USB in and check for continuity between the metal socket sheath and the output wires.
3. Solder the exposed wire lengths to the board, trying to ensure tht there is little to no exposed wire showing above the board.

The USB can now be used to quickly plug and unplug the battery.

12.3 Using an Enclosure

An enclosure can be used to protect the electronics during use, and improve the look of the ODOTS. The 1591XXSSFLBK enclosure from Hammond engineering was used for the proof of

concept, but has not been specified in the component list as the choice of enclosure is likely to be updated in the near future. For the 1591XXSSFLBK all electronics should fit (all in one plane) on the PCB mountings in the lid. You will need to drill holed to poke the LEDs through.

Attach components to the inside of the enclosure with hotglue and mark the outside to indicate where the RFID antenna is (for competitors) and the box function and ID number (for yourself and competitors). You may want to use some kind of sealant to close the box, though in the proof of concept we simply ensured the box was shut securely and added some hot glue to the LEDs protruding from the box to add some kind of water resistance.

Chapter 13

Compiling and embedding firmware

The proof of concept design has been built with this particular build step in mind. The embedded firmware cn be compiled from the available source code using the Arduino compiler, accessible through the Arduino IDE (available at `www.Arduino.com`). After downloading the .zip file from the ODOTS-release Github repository (`https://github.com/ljones278/ODOTS-Release`) the ODOTS library may be imported into the Arduino IDE by using the ' add .ZIP library' tool (accessible from the top menu 'edit' then manage libraries).

The current ODOTS firmware program (or 'Sketch' in Arduino terms) is stored as an example 'ODOTS-Generic'. THis can be quickly accessed through the IDE by using the 'file' drop down menu, 'examples' scrolling to the bottom of the menu that appears, 'ODOTS', 'ODOTS-Generic'. You may need to reboot arduino after importing the library for the IDE to show the example sketch. To verify that the code is still valid, use the verify tool, (ctrl-v or the 'tick' button at the top of the program). If this throws errors you can either enter the rabbit hole of debugging (it goes a looong way) or take the error message and start an error report on the Github repository.

You now have a working set of code sitting on your computer, but Ideally we want it running on the Microcontroller. If you are using the proof of concept design and have used the suggested purchase instructions above you should have an Arduino UNO board hanging about.

1. Connect the spare Arduino UNO development board (with chip still mounted)to a computer and upload the Arduino ISP example sketch to it.

2. Connect the spare Arduino UNO development board's SPI port to the SPI port of the ODOTS unit microcontroller (accessible as the wires used to talk to the RFID board.). The reset pin on the Microcontroller should be connected to pin 10 of the development board. The slave select pin does not need to be used.

3. Use the Burn bootloader tool (in the tools drop menu of the Arduino IDE), ensuring that the correct board is selected (arduino Uno/Genuino) and the correct ISP (Arduino as ISP). This will now burn in the bootloader to the Unit Microcontroller.

4. Dismount the Arduino from the development board socket (it may be easiest to burn all units with their bootloaders before doing this, repeated socket removals can damage the chip.).

5. Disconnect the SPI wires from the ODOTS unit and instead attach the UART serial wires. The reset pin is still required, but can now be connected to the standard Reset port of the development board.

6. Use the upload tool to upload the code onto the microcontroller (ctrl-U or with the 'upload' button at the top of the IDE).

The microcontroller should now be programmed, final configuration can be managed with the serial interface, including its ID (the number written to competitor cards to identify the unit), function and time. You can check that the program has been successfully embedded by try to talk to the unit over the serial port, sending a 'B' character should return "b[UNPRINTABLE]»timeout", though the timeout takes about two seconds to trigger.

Chapter 14

Installing the UI

14.1 As A User

A frozen version of the UI application exists in the ODOTS-Release repository (`https://github.com/ljones278/ODOTS-Release/tree/master/Software/ODOTSInterface/ODOTSInterface`). Download the ODOTSInterface Folder into your favourite application folder (or your desktop, the program is not picky). The primary application is ODOTSInterface.exe, SendPunch.jar is used to interact with MEoS while the 'Download Records Folder' is used to store Card download information. Ideally it should be emptied every now and then once the downloads are no longer relevant.

On start up the Program performs a rather crude check to see if a JRE is installed on the host system, and will prompt the user if this is not passed. The JRE is only required to run the interface with MEoS, ODOTS unit configuration can be achieved without it.The program will crash as soon as a card download is attempted if the JRE is not installed, this will be fixed in a future update.

14.2 As a Developer

The source scripts for the ODOTS UI and serial manager are stored (in their current release form) on the ODOTS-Interface Github repository (`https://github.com/ljones278/ODOTS-Interface`).

Most of the UI has been written for Python 3.7, which can be obtained from the Python foundation (). This site has installers that should manage the entire installation and setup for you.

The serial interface uses the PySerial module, this can be quickly installed using the pip utility. Simply launch an instance of the Command line (or a bash shell) and type: *pip install pyserial*. If you are running a Mac (or have already dabbled in the dark arts) you may have a Python 2 version lurking somewhere in your system. In this case using *python3 -m pip install pyserial* ensures that the PySerial module gets installed for the correct Python version.

The final installation required is a Java environment for the program that sends information from the ODOTS interface to MEoS. This requires a trip to Oracle *(ideally this will be removed*

as a required step in the future, their site if infuriating - apologies.) either the JRE or JDK will do.

All dependencies should now be installed. Running the UI should be as simple as double clicking 'ODOTSInterface.py' (if you do not have a Python 2 installation) or navigating to it through your command line/shell and invoking it with *'python3 ODOTSInterface.py'*. Instructions for using the UI can be found in the User guide.

Other Tips

Suggested Phrases to sound experienced while building:

- The Bus Interface has breached thermal limits! *(The USB interface chip has caught fire because I touched it with the soldering iron)*
- Compilation of Embedded Routines performing nominally *(The code for the arduino compiled)*
- Time Synchronised Reference enabled *(The RTC chip turned on)*
- Environmental Contamination has led to a catastrophic increase in undesired conductivity! *(Someone spilt tea over the electronics and shorted out some of the components)*
- An Unexpected Reversal of Power Supply polarity has triggered an exothermic event! *(I plugged in the battery the wrong way and my chips are now emitting blue smoke)*
- Accidental Power Omission has resulted in null operation! *(I forgot to plug in the battery so it did not work.)*

Part III

User Guide

Chapter 15

User guide introduction

This document was written for the Proof of concept version of the ODOTS hardware and software. Therefore the user experience may be slightly less straightforward than expected. If you run into difficulties part II may prove useful. Having said that once the ODOTS is up and running it has been designed to be as painless to use as possible, bar a few quirks that are, again, a product of the fact that the current iteration of the design is just a proof of concept. This section shall assume that the user is in possession of a fully constructed and programmed (firmware flashed) ODOTS set of Units and Competitor cards.

Chapter 16

General Guidance

16.1 Installing Software

A frozen version of the UI application exists in the ODOTS-Release repository (`https://github.com/ljones278/ODOTS-Release/tree/master/Software/ODOTSInterface/ODOTSInterface`). Download the ODOTSInterface Folder into your favourite application folder (or your desktop, the program is not picky). The primary application is ODOTSInterface.exe, SendPunch.jar is used to interact with MEoS while the 'Download Records Folder' is used to store Card download information. Ideally it should be emptied every now and then once the downloads are no longer relevant.

On start up the Program performs a rather crude check to see if a JRE is installed on the host system, and will prompt the user if this is not passed. The JRE is only required to run the interface with MEoS, ODOTS unit configuration can be achieved without it.

16.2 Configuring ODOTS Units

ODOTS unit configuration is managed by the *ODOTSInterface* utility. This program allows a user to manually set certain ODOTS unit parameters while also acting as a download bridge between the ODOTS unit and the competition management software. Unit parameters that can be set from the *ODOTSInterface* utility include:

- Device Time (sync to system time, the time according to the computer, in GMT).
- Device Number (control code in common orienteering terminology).
- Device mode, clear (to reset cards on punch), download (to dump card information to serial on punch), or standard (your normal checkpoint unit).

All of these parameters can also be read from the device as well as the device firmware version and hardware version numbers (useful if you are getting weird errors). Things that cannot currently be read from the device include the battery voltage. The utility includes a built in help page that will have a list of implemented commands, their function and invocation.

To configure an ODOTS Unit:

1. Launch the ODOTS Configuration utility.

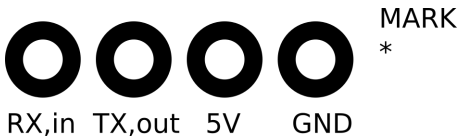

Figure 16.1: Serial Pinout header from hardware version 0.

2. Connect the ODOTS unit to your chosen serial to USB converter.
    - This requires plugging in the RX and TX wires as well as the GND and 5V wires (to ensure that the voltage levels line up) from the ODOTS serial header into the converter. If you are using an empty Arduino UNO board as the bridge these pins will be labelled on the board. The pinout of the ODOTS serial header has been included as figure 16.1.
    - DO NOT unplug the battery, in this case it is buffered from the serial line by an intervening converter, unplugging the battery will reset the clock when you unplug the serial line and the device loses power!
    - Later version of the ODOTS will have built in USB converters, watch for updates!
3. Plug the USB end of the converted into your chosen host computer.
4. In the configuration Utility list available ports, (”l” or ”LISTPORTS”)
5. Find the ODOTS device, which should come up as ”Arduino UNO”.
6. Connect to the device by commanding connect and then inputting the list index of the ODOTS device when asked.
7. The device should now be connected to the software, if it is being used as a download unit its card download dumps will be forwarded on to MEoS automatically.
8. The device can now be configured by using the command listed by the utility help page. It is suggested to read the card info after making the required changes to ensure that they worked.
9. Before disconnecting the device it is good form to close the serial port, achieved by using the close port command. This is not essential but will allow you to attach a new device and connect to it without rebooting the program. Not disconnecting the serial port will not damage the ODOTS device.

16.3 Downloading Card Data

The ODOTS interface software has been made so that the ODOTS downloaded data looks like the data from other timing systems to the competition management software. This makes downloading card data particularly straightforward.

1. Start the ODOTS configuration software and connect to a device as detailed above. The ODOTS side is now set up!
2. Start MEoS and select your competition if you have not done so already (MEoS instructions can be found at `http://www.melin.nu/meos/en/show.php`)

3. In MEoS, in the 'Sport Ident' tab change the connection from "COM Port k" to "TCP" (there is a drop down menu).

4. Press the 'Activate button', this will bring up extra boxes that will allow you tyo change the TCP port. Don't, the ODOTS software looks for port 10000.

5. Press 'Start' (one of the new boxes). This will start MEoS listening for incoming connections on that port, which is how the ODOTS software talks to MEoS.

6. Card downloads will now be automatically passed to MEoS for it to handle. It is worth double checking that the download was successful before letting competitors run away, this can take a while, especially if they are moving their card around during the download!

16.4 What to expect when running with an ODOTS card

Running with an ODOTS card will be very similar to running with any other orienteering electronic timing system. Competitors should clear their card before starting. Starts can be done either as a 'punching start' with a dedicated start unit, or as a 'timed start' with competitors expected to start at a set time. At each checkpoint, competitors can register their presence by placing their card in the RF field of the ODOTS unit. Successful punches elicit a two tone beep and flashing lights, unsuccessful punches result in a single tone double beep and no flashing lights.

After completing their course competitors should download their card data at a download unit. The download procedure may take a while, particularly for MIFARE 4k cards. Be prepared for a couple of error buzzes before the download is successful. Always double check with the competition software that the download was indeed successful.

16.5 Maintenance and Troubleshooting

This section will be updated as the system gets used to include common issues and maintenance tasks.

Chapter 17

Guides for Using the ODOTS

17.1 Preparing for an Event

1. Charge all unit batteries to full capacity (this may take a while if you have a lot of units).
2. Plug the batteries in to the Units, the units should last for about two days after the battery has been added before running out of power.
3. Configure all units with desired functionality and ID numbers, the device time should also by synced with the computer, the Configuration software uses GMT to send to the Unit.
4. Seal all units in their enclosures (apart from download units).
5. Store download units with their accompanying serial bridges which will be used to allow them to talk to a computer running the competition management software.

What units go where?

Generally any course will require (in order of use): A reset Unit, a start Unit (a standard control), any number of standard controls (one per checkpoint), a finish unit (a standard control) and a data download unit.

17.2 Running an Event

During an event (after all controls have been placed), the only bit to worry about (as the ODOTS manager) is the data download units. These will need to be plugged into a computer running the ODOTS Interface UI and competition management software using a serial bridge. The serial bridge can be the spare Arduino Uno board. Connect to the unit in the UI (you can attempt to read unit data with the unit to ensure that the connection is working). If the connection appears to be broken you will need to reboot the UI and attempt to connect again.