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 and that you label any ODOTS builds with either the ODOTS logo or ”ODOTS”. Bonus points of course for finding something to make the ODOTS out of that cannot be written on by a Sharpie pen!

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, sections 2 contains the quantitative and descriptive specification and performance metrics of the ODOTS. For reader wishing to use this document for the second purpose, sections 3, 6 and 5 are a reference for the hardware design and software structure of the ODOTS. For reader wishing to use the ODOTS for themselves sections 8 and **??** are intended to take you through the process of building and using an ODOTS.

Part I

Developer Guide

1.1 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.

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.

Chapter 2

Performance and Specification

This chapter provides information of the perfomance of the ODOTS and some brief technical specifications for use in assessing the ODOTS for a particular calculation.

2.1 Electrical Characteristics

Characteristic	**min**	**typ**	**max**	**unit**
Power Supply	5.5	9	40	V

Table 2.1: Specimin Electrical Characteristics

2.2 Performance Evaluation Metrics Metrics

Characteristic	**min**	**typ**	**max**	**unit**
Time to Power Drain	4		30	yrs
Visit Process Time	6		10	ms
Clock Drift	1		100	sec/Week
Punches Per Card	92		180	Visit Records
PCD Cost	15			£
PICC Cost	0.5			£

Table 2.2: Specimin Electrical Characteristics

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 .cpp also acts as the top layer of the library with glue code to pass non-RFID functionality to

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

Table 3.1: Brief structure of firmware source files

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()

ODOTS.cpp

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

Arduino.h,*EEPROM.h*, MRFC522.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 **??**.
Returns: True if interaction successful, false if error.

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 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 claculates time in lilliseconds by assuming the ihnterrupt was called at Xh:Xm:Xs:0ms
and then counting milliseconds since the interrupt was called.
Returns: 0.

---

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)

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)

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.

Chapter 4

RFID Card memory management

The RFID cards used by the ODOTS, in the current design must be MIFARE classic compatible devices, at testing has showed that MIFARE Classic 1k rather that 4k gives slightly better performance. Early in the design process every effort was made to ensure that a wide variety of cards could be use, 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 oraganisation 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 block). 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 record is 8 bytes long allowing for two per block. The structure of the ODOTS visit record

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

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 chapter **??**. 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 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.
Parameter:

5.1.2 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 de-

tected, 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.

5.1.3 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 1.1. 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<-
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 manager Interprocess interface

This interface allows the asynchronous operation of the User Interface for ODOTS configuration and management and the actual implementation of this operation and management. The interface exists through two queue objects, currently using the multiprocessing.Queue but simply changeable in the future. 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

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.

Part II

Build Guide

Chapter 8

Building Guide

8.1 Acquiring Components

8.2 Building Hardware

8.3 Compiling and embedding firmware

8.4 Running the UI

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 9

User guide introduction

This document was written for the Proof of concept version of the ODOTS hardware and software. Therefore software installation and hardware acquisition 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.

9.1 General Guidance

9.1.1 Installing Software

9.1.2 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.
2. Connect the ODOTS unit to your chosen serial to USB converter.

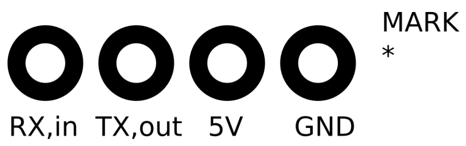

Figure 9.1: Serial Pinout header from hardware version 0.

- 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 9.1.
- 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.

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

4. 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.
5. 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 mving their card around during the download!

9.1.4 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.

9.1.5 Maintenance and Troubleshooting

9.2 Tutorials

9.2.1 Preparing for an Event

What units go where?

9.2.2 Running an Event

References