The University of Sydney
Faculty of Engineering
School of Electrical and Information Engineering

Year S2, 2020

A thesis submitted in partial fulfilment of a Bachelor of Engineering Degree

Student Name	**Ivan Hu**
SID	**460383543**
Unit of Study Code and Name	**ELEC4713 Thesis B**
Supervisor	**Dr Zihuai Lin**
Title	Design of an embedded system with on-demand image capturing and transmission for remote agricultural monitoring

© Copyright 2020

The University of Sydney

SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

PROJECT CLEARANCE FORM

Unit of Study Code and Name **ELEC4711 Engineering Project B**

This is to certify that my student

Student Name Ivan Hu

SID 460383543

Has:

- **Returned all books and reference material;**
- **Returned all equipment and keys; and**
- **Tidied their work place.**

SUEIE Academic supervisor:

Signature: Zihuai Lin **Date: 17/11/2020**

Name: Zihuai Lin

External supervisor (if applicable):

Signature: **Date:**

Name:

Design of an embedded system with on-demand image capturing and transmission for remote agricultural monitoring

IVAN HU

B.Eng (Hons)

Supervisor: Dr Zihuai Lin

School of Electrical and Information Engineering
Faculty of Engineering
The University of Sydney
Australia

18 November 2020

Statement of Achievement

- Design and simulation of a latch circuit to facilitate turning the project device on/off.
- Hardware assembly of the project device.
- Hardware assembly of a platform to upload firmware onto the microcontroller.
- Full firmware implementation of the project transmitter device.
  - Using existing code libraries in conjunction with original code to take a picture and store the result in an SD card.
  - Development of a firmware finite state machine to send an AT command sequence to the LTE modem that will facilitate file transmission.
  - Using the MQTT protocol as a novel means of facilitating file transmission.
  - Using existing libraries in conjunction with original code to encrypt and encode the file in AES128 and Base64 respectively prior to file transmission.
- Python development of the file receiver programs.
  - Development of a script which will subscribe to the same MQTT broker and topic as the transmitter in order to download the encrypted and encoded file.
  - Development of a script which will decrypt and decode the file to recover the original image.
  - Development of a HTML-based webpage dashboard on which the user can execute the two aforementioned scripts and view the recovered image.

Abstract

The ability to visually verify some element of a remotely controlled agricultural automation system through a photograph is valuable in many cases, not only in the operational phase of the system, but especially in the design and implementation phases. Owing to the remote location of many of the application sites, cellular techology is one enabling medium through which wireless transmission of the photographs could be realised. As data usage is a concern for systems using cellular technology, MQTT chosen over other protocols due to its lower message-to-header overhead. This paper outlines the hardware and firmware design of the LTE Cat-M1 enabled embedded system and the backend web development of the cloud-based web application to facilitate the receiving of the photograph. A satisfactory degree of implementation success was achieved in this project, with deployment to a production environment possible after further refinements.

Acknowledgements

I thank Dr Zihuai Lin for helping me shape the vision of the project in my head into something that is more fleshed-out and reflective of real-world concerns. I thank Ric Otton, the managing director of the company I work at for allowing me to borrow some crucial electronic equipment over the course of my project. I thank my parents for allowing me to use the dinner table every night for project development and debugging, under the condition that I clean up every night. And I thank my Scarlett for providing the day-to-day emotional support that I need to not only slowly-but-surely progress through this project, but also through 2020.

Contents

Statement of Achievement ii

Abstract iii

Acknowledgements iv

Contents v

List of Figures viii

Chapter 1 Introduction 1

Chapter 2 Literature review 4

2.1 LTE Cat-M1 4

2.1.1 Lower complexity and cost 4

2.1.2 Coverage 5

2.1.3 Low power usage 5

2.1.4 Data rate 6

2.2 MQTT 7

2.2.1 Low messaging overhead 7

2.2.2 Quality of service (QoS) guarantee 7

Chapter 3 Methods 9

3.1 System overview 9

3.1.1 Typical operation 9

3.1.2 Development process 10

3.1.2.1 Digitally-toggled power latch 10

3.1.2.2 Hardware development 11

3.1.2.3 Debugging techniques 12

3.1.2.4 Firmware development of cellular functionalities 16
3.1.2.5 Python development of the file download functionality 20
3.1.2.6 Firmware development of file processing functionalities 22
3.1.2.7 Python development of the file decoding and decryption functionalities 24
3.1.2.8 Development of a HTML-based dashboard 25
3.1.2.9 Firmware incorporation of the camera function 27

Chapter 4 Results and Discussion 28

4.1 Cellular data usage 28
4.2 Image characteristics 28
4.3 End-to-end process duration 29
4.4 Power usage 31
4.4.1 Data security 32
4.4.2 Data privacy 33

Chapter 5 Conclusion 35

5.1 Future outlook 35

Bibliography 38

Appendix 41

Appendix A: cellularFunction.h 41
Appendix B: cellularFunction.cpp 42
Appendix C: mqtt_subscriber_file.py 47
Appendix E: sdCardFunctions.h 47
Appendix F: sdCardFunctions.cpp 48
Appendix G: aesbase64decode.py 50
Appendix H: index.html 50
Appendix I: cameraFunctions.h 50
Appendix J: cameraFunctions.cpp 50
Appendix K: Main setup and loop functions 52
Appendix L: Digitally-toggled power latch simulation 53
Appendix M: Video demonstrations 54

Appendix N: Github repositories 54

List of Figures

1 Telstra network LTE-M coverage within Australia 6

2 Digital pulse input latching power switch circuit 11

3 Timing diagram of the circuit response 11

4 The project transmission device 13

5 The programming jig 13

6 ESP32-CAM schematic 14

7 Development workspace setup with TTL to USB converter 15

8 Monitoring AT commands and responses with RealTerm 15

9 HTML-based webpage dashboard 26

10 Transmitted file. Timezone is UTC (GMT +0) 28

11 Cellular service provider. Timezone is GMT +11. 28

12 The captured image. 29

13 Comparison of the input file (left) and the recovered file (right). 37

CHAPTER 1

Introduction

Technological advancements constitute a crucial productivity multiplier in the agricultural sector, enabling expenditure reductions of time, manpower and material input for the same amount of yield.

Internet of Things (IoT) is a technology that can be leveraged to great advantage in the agricultural sector. This is particularly the case in Australia, where the vast geographical size of the continent coupled with low population density means that it is economically desirable to minimise human operators manually working on-site on farms, thereby avoiding overheads associated with travelling to, sometimes multiple, remote agricultural locations far away from the place of residence of the operator.

Through a network of sensors and actuators connected to a wireless IoT network, the operator has the power to remotely gather data such as precipitation, photosynthetically active radiation (PAR), and soil moisture about his/her farm. Using this information, the farmer could actuate nodes which could perform several functions, such as actuating a solenoid valve for irrigation/fertigation. Alternatively, actuators could be programmed to actuate automatically according to values reported by the sensor.

The degree to which the "smart farm" system is productive depends greatly on the accurate and reliable operation of the sensors and actuators. For example, the valve actuators must accurately deliver a certain volume of water in order to effectively irrigate/fertigate a certain crop, but the only variable that the system has control over to achieve this goal is the duration of actuation. However, the relationship between duration of actuation and volume of water delivered cannot be generalised to all work sites because there are other variables at play which

affects this relationship, such as the water pressure within the water input pipes. The degree to which this variable can vary depends on yet more factors, such as whether a water pressure regulation system is installed onsite [1]. Thus, this relationship must be experimentally obtained in the implementation stage of the "smart system" for each farm.

Sensors must also work accurately and reliably because effective actuation depends on these measurements. For example, for the measure of precipitation, tipping-bucket rain gauges are known to exhibit significant errors if based on time scales of less than 10 to 15 minutes [2]. These issues could be fixed to a satisfactory degree during the implementation stages of the system through the experimental processes of observation and calibration.

It would be greatly beneficial to the initial implementation process to have a method of visually verifying the correct operation of sensors and/or actuators in the "smart farm". Engineers would implement the system according to theoretical specifications, and then remotely monitor the system for its actual behaviour before making the necessary adjustments in the next prototyping iteration, until the system behaves according to expectations.

An embedded device (hereafter known as the project transmission device) incorporating a camera, LTE-M Cat M1 modem and a SD card in conjunction with some software to receive the transmission, would be a solution to satisfy the above requirements.

The project transmission device would be electronically connected to the device device being monitored using a digital I/O. After the monitored device performs its function, it would send a digital pulse signal to the project device, which would be powered on via a latching power switch circuit. The project device would then take a picture of the monitored subject, store it in the project device's local SD card, perform encoding and encryption, upload the file to the cloud, and then power itself off by sending a digital pulse signal to the latching power switch circuit.

The outcome of this project would give engineers a method of remotely verifying the correct operation of the agricultural monitoring devices during the implementation stage without the hassle of going on-site. Not only that, the project device would also provide the ability

to debug problems post-implementation, during the monitoring phase, if the “smart farm” system is observed to or suspected of exhibiting unexpected behaviour.

CHAPTER 2

Literature review

2.1 LTE Cat-M1

LTE-M, of which LTE Cat M1 is a subset, is a low power wide-area network (LPWAN) radio communications technology standard defined by the 3rd Generation Partnership Project (3GPP).

The design of LTE-M is motivated by a need to extend existing LTE technology for better support of machine-type communications (MTC). As such, the specification is designed with some radio access principles in mind.

2.1.1 Lower complexity and cost

LTE-M was designed with the motivation to bring down costs of LTE modem production to a value comparable to GSM and GPRS, as at the time of conception, demands for MTC were adequately satisfied by those types of cheaper modems [3].

3GPP Release 12 (Cat 0) was the first release of LTE-M and it has incorporated some elements of cost and complexity reduction as compared to Cat 1, an LTE technology. It is characterised by a reduced data rate of 1/1 (Mbps DL/Mbps UL) down from 10/5 (Mbps DL/Mbps UL), a single receive antenna down from two, and optional half-duplex frequency-division duplex (HD-FDD) operation [3].

3GPP Release 13 (Cat M1), from the specifications of Cat 0, incorporates further cost and complexity reductions characterised as a reduced bandwidth of 1.4 MHz down from 20 MHz and an optional reduced transmit power of 20 dBm [3].

2.1.2 Coverage

Nodes in IoT applications may be situated in locations where it is difficult to receive adequate coverage. Coupled with device simplification measures intended to drive down the complexity and cost of LTE-M modems, the device would experience significant coverage loss unless it is compensated with some coverage enhancement (CE) techniques.

CE techniques are used to increase the chance of successful transmission in low-coverage environments. In normal 4G operation, each transmission lasts approximately 1 millisecond. As such, a common coverage enhancement technique is the repetition of each transmission in order to increase the likelihood of successful transmission [4].

LTE-M specifies two modes of CE, A and B, which respectively have increasing coverage effects resulting from increasing degrees of repetition. 3GPP Release 13 mandates that Cat M1 devices must have support for CE mode A, with CE mode B support being optional. CE mode B is always a superset of mode A.

In Australia, the Telstra network is able to provide LTE-M coverage to an extent that covers the overwhelming majority of Australia's arable land.

2.1.3 Low power usage

Power Saving Mode (PSM) is specified for LTE-M in which the device enters a state where it is unreachable by mobile terminated (MT) services, and as a result, power consumption is reduced to a bare minimum. The LTE-M modem only leaves PSM when a higher layer in the device triggers a mobile originated (MO) access, for example for a data transmission [3].

FIGURE 1: Telstra network LTE-M coverage within Australia

A further power optimisation feature of PSM is that in this mode, the device stays registered to its network and preserves its connection configurations. This means overheads associated with initial network attachment and connection setup are avoided, as would be otherwise needed if the device were to experience a complete power cycle [3]. Not only does this measure optimises the device for power consumption, but also lessens the time between initiation and completion of a MO access.

After the data transmission associated with the MO access is complete, the device enters a period of MT reachability as specified by its Active Timer. After the Active Timer expires, the device once again enters the PSM state, remaining unreachable by MT services until the next MO event, thus continuing the cycle [3].

2.1.4 Data rate

According to [3] the estimated maximum uplink data rate for a Cat M1 device supporting (half-duplex frequency division duplex) HD-FDD is 375kbps. Data rate depends on the

coupling loss experienced the device, with the maximum coupling loss (MCL) data rate at 164 dB being 167 bps.

2.2 MQTT

Message Queuing Telemetry Transport (MQTT) is a protocol introduced in 1999, having been developed by Andy Stanford-Clark and Arlen Nipper from IBM and Arcom Control Systems Ltd respectively. The messaging protocol revolves around a many-to-many relationship between clients performing publishing roles, subscribing roles, or both, via a message broker. A publishing client publishes messages to a broker, to addresses known as topics, from which subscribing clients can subscribe to and receive messages from every topic [5].

2.2.1 Low messaging overhead

MQTT, being designed for lightweight machine-to-machine (M2M) communications in constrained networks [6], has a lower message to message overhead ratio compared to other protocols such as AMQP and HTTP [5].

Each MQTT control packet has a two-byte fixed header, with the 0th byte detailing the type of control packet and flags specific to the control packet type, and the 1st byte dedicated to specifying the remaining length of the packet. For every two-byte fixed headers sent, a payload of up to 256 MB can be supported.

2.2.2 Quality of service (QoS) guarantee

MQTT has three levels of QOS built into the protocol that offer three different guarantees of message delivery.

QOS 0 guarantees that a message is received “at most once”. It is more intuitive to think of this QOS level as an absence of guarantee rather than a guarantee, because this is simply a

fire-and-forget transmission. The recipient does not acknowledge the reception of the message, nor does the sender store the message so that it can be re-transmitted [7].

QOS 1 guarantees that a message is received “at least once”. As the name suggests, there is a potential for the recipient to receive duplicate messages, and this isbecause the sender stores the message and continues to retransmit the message until the sender receives an acknowledgement packet from the recipient [7]. This two-part handshake constitutes the middle ground out of the three QOS levels in terms of overhead.

QOS 2 guarantees that a message is received “exactly once”. Utilising a more complex and resource-expensive four-way handshake system than QOS 1, it eliminates the issue of potential duplicate messages and guarantees that each message is delivered exactly once. The sender first publishes the message to the recipient and will continue to do so until a PUBREC message is received from the recipient. PUBLISH packets subsequent to the initial one will have its duplicate (DUP) flag set. The sender will eventually receive and store the PUBREC, at which point the process is repeated with the final two handshakes PUBREL and PUBCOMP. After this process, the receiver is left with one copy of the original PUBLISH packet and the sender is assured that a copy of the original PUBLISH packet has been sent [7].

CHAPTER 3

Methods

3.1 System overview

3.1.1 Typical operation

The project transmission device lies in a powered-off state until it received a digital input pulse from the monitored device. The project transmission device will then power on and take a photograph. The photograph is stored in an SD card attached to the project transmission device as an JPEG file. The JPEG file is padded to the nearest file size that is divisible by 16 bytes. The JPEG file is encrypted with AES-128 using a pre-defined key. The encrypted file is encoded using Base64. The microcontroller sends a sequence of AT commands to the LTE-M modem to establish signal quality and establish a connection with an MQTT broker. The project transmission device will command the LTE modem to do a sequence of MQTT publishes which will publish the entire encrypted and encoded JPEG image in as many 1500-byte segments as required.

On the receiver end, an active Python script (the "Downloader") running on the Linux backend of an AWS instance would have already established connection to the same MQTT broker and have subscribed to the same topic that the project transmission device publishes to. The Downloader downloads the encrypted and encoded image file onto the storage of the Linux server. Another python script (the "Decoder/Decrypter") is executed, which after the correct password is inputted, will decode the file from Base64 to an encrypted bitstream and then decrypt the file using the same pre-defined key. The Decoder/Decrypter writes original image onto the storage of the Linux server. The user interfaces with the receiver system

via a webpage that is hosted on an Apache webserver that is running on the same Linux server. The webpage embeds two iframes containing respectively the Downloader and the Decoder/Decrypter, and an img element that displays the last image that has been downloaded, decoded and decrypted. The webpage serves as a rudimentary HTML-based dashboard.

3.1.2 Development process

This section gives a logically and roughly chronologically ordered outline of the development process. Each subsection will outline the justification and development process of a different technology used in this project.

3.1.2.1 Digitally-toggled power latch

The use-case of the project transmission device is query-based, such that the system will come out of a dormant state based only on an external stimulus, as opposed to an internal wake-up source such as a timer. This means that the project transmission device need not have any computational capacity whatsoever during its dormant state. By exploiting this fact in conjunction with the nature of low duty-cycle embedded systems that most of their energy expenditure originates from dormant-state quiescent current [8], significant power savings could be realised if a duty-cycling mechanism is chosen such that power is completely cut off from the project transmission device during its dormant state.

The solution to this problem is the use of a latching power switch circuit that takes two digital signals as inputs to switch power on and off respectively. The circuit and its timing diagram is as follows.

The use of this circuit also has the additional benefit of saving development time, because there is no longer a need to configure the microcontroller or any peripheral components to go into their respective sleep modes via firmware.

This circuit is adapted from [9] with the adaption being the use of digital inputs instead of physical momentary switches as in the original circuit.

FIGURE 2: Digital pulse input latching power switch circuit

FIGURE 3: Timing diagram of the circuit response

3.1.2.2 Hardware development

The AI-Thinker ESP32-CAM module is chosen as the main microcontroller module because it has an integrated camera, SD-card reader and an on-board 5V to 3.3V regulator. Furthermore, this module is officially supported by the Arduino libraries written for the ESP32 microcontroller, enabling ease of interfacing with the SD card and camera peripherals.

The Dragino NB-IoT Bee QG96 in conjunction with its NB-IoT Shield designed for the Arduino Uno is chosen as the cellular transmitter for the project transmission device. The Shield features on board 5V to 1.8V regulator which is useful because it can share the same 5V power rail with the ESP32-CAM module. Despite its name, the NB-IoT Bee features the Quectel BG96 modem which is primarily an LTE Cat-M1 modem.

As the NB-IoT Shield is designed to interface with an Arduino Uno and not an ESP32-CAM, the two components had to be soldered onto a prototyping board with appropriate connections made using wires soldered to the underside of the board. The ESP32-CAM's UART port is connected to that of the Shield. Other than the 5V input power and ground rails, the only other notable connection is the 3.3V power rail between the two components as there is no on-board 3.3V voltage regulator on the Shield because it is designed to use the 3.3V power rail from its controlling board.

The Shield and the ESP32-CAM are not directly soldered onto the board but are rather inserted into female header strips which are directly soldered onto the board. This modularity enhances the convenience of reprogramming the ESP32-CAM as the module must be inserted into a separate programming jig to do be reprogrammed. The programming jig features a generic FT232RL USB-to-TTL module and jumper wires to make the correct connections to the ESP32-CAM module.

3.1.2.3 Debugging techniques

JTAG-based debugging techniques commonly used in conventional embedded firmware development were not possible with the hardware used in the project transmission device. Although the ESP32 microcontroller possesses JTAG capabilities, the associated pins are already in use in the ESP32-CAM module to interface with the SD card and the camera [10].

The relatively small number of pins (32) on the ESP32 family of microcontrollers means that the majority of pins in the ESP32-CAM modules are used to satisfy its I/O-hungry applications, namely to interface with the camera and the SD card. This means that there is only one UART port available to the user for development applications. For this application,

FIGURE 4: The project transmission device

FIGURE 5: The programming jig

that single UART port is used to interface with the LTE Cat-M1 modem for AT commands. This essentially eliminates the ability to debug through printing out variables. However, the

ability to merely read AT commands between the microcontroller and the modem proved to have considerable debugging utility. This was achieved through the use of a TTL to USB converter by connecting the Rx of the TTL input to the Tx of the modem. Due to the fact that the modem echoes back any AT commands the microcontroller sends to it prior to sending back its response, merely monitoring this single connection is enough to obtain the information for the entire serial exchange between microcontroller and modem. The serial exchange can be monitored on a PC through any terminal software, which in this case was RealTerm.

FIGURE 6: ESP32-CAM schematic

FIGURE 7: Development workspace setup with TTL to USB converter

FIGURE 8: Monitoring AT commands and responses with RealTerm

3.1.2.4 Firmware development of cellular functionalities

The source code is designed to be highly modular, with each module defined according to function. The three modules each govern the camera, cellular transmission and storage functions respectively. Each module has its own `.cpp` and `.h` file and its functions are called from `ESP_mqtt_code.ino` in which the main loop of the program lies. The modular nature of the code aids in debugging during the development process because problems are isolated to that particular module.

```
void setup() {
  Serial.begin(FAST_CATM1_BAUDRATE);
  sdInit();
  configInitCamera();
  takeSavePhoto("/input.jpg", SD_MMC);
  aesEncode(SD_MMC, "/input.jpg", "/inputjpg_encrypted");
  base64Encode(SD_MMC, "/inputjpg_encrypted", "/inputjpg_encrypted_encoded");

  getFileSizeSegment(SD_MMC, "/inputjpg_encrypted_encoded");
}

void loop() {
  cellularHandler();
}
```

LISTING 1: Main program setup and loop (ESP_mqtt_code.ino)

The firmware development process began with the development of the cellular functions module. The cellular module, like every other module in the code, has all its variables and buffers stored in an extern struct. This design feature is to emphasise the modular nature of the code. Being explicit during the process of reference reduces the potential for confusion. For example, if another module were to access the flags register of the cellular module, it would have to reference this variable using `cellularObj.flags_reg`.

```
typedef struct {
  char tx_buffer[TX_BUFFER_LEN];
  char rx_buffer[RX_BUFFER_LEN];
  unsigned long tx_wait_duration_ms;
  unsigned long tx_start_time_ms;
  uint8_t flags_reg = 0;
} cellularObj_t;

extern cellularObj_t cellularObj;
```

LISTING 2: Cellular module struct definition and declaration (cellularFunctions.h)

As the cellular module is essentially responsible for determining and sending the correct AT command based on the result of the previous AT command, the flow control of the cellular module is implemented as a finite state machine (FSM) with its state stored in an `enum`.

```
typedef enum {
  cellular_init,
  cellular_wait_RDY,
  cellular_do_RDY,
  cellular_send_CSQ,
  cellular_wait_CSQ,
  cellular_do_CSQ,
  cellular_send_QMTOPEN,
  cellular_wait_QMTOPEN,
  cellular_do_QMTOPEN,
  cellular_send_QMTCONN,
  cellular_wait_QMTCONN,
  cellular_send_QMTCONNquery,
  cellular_wait_QMTCONNquery,
  cellular_do_QMTCONNquery,
  cellular_send_QMTPUBCmd,
  cellular_wait_QMTPUBCmd,
  cellular_do_QMTPUBCmd,
  cellular_send_QMTPUBhelloworldMsg,
  cellular_wait_QMTPUBhelloworldMsg,
  cellular_do_QMTPUBhelloworldMsg,
  cellular_send_QMTPUBfileCmd,
  cellular_wait_QMTPUBfileCmd,
  cellular_send_QMTPUBfile,
  cellular_wait_QMTPUBfile,
  cellular_do_QMTPUBfile,
  cellular_done
} cellular_handler_state;
```

LISTING 3: Cellular module finite state machine (cellularFunctions.cpp)

Execution of code during each state is done via a handler function which can repeatedly be called from the main loop. This has the additional benefit of the code being non-blocking. For example, if the handler function determines that the modem has not yet detected a signal, it can simply return from the function and try again the next time that the main loop calls the handler function.

Another design feature that is consistent with the non-blocking nature of this module is the use of non-blocking waits. Each AT command sent to the modem has a specific maximum response time that is specified by the Quectel BG96 application notes. For example, the `AT+CSQ` command has a maximum response time of 300ms [11]. Therefore, the code will send the command in the `cellular_send_CSQ` state and then store the number of milliseconds since beginning of code execution. In the next `cellular_wait_CSQ` state, the code will check whether 300 milliseconds have elapsed since that time. Because the code is non-blocking, the main loop will keep calling `cellularHandler()` for 300 milliseconds, after which point the FSM will progress into the `cellular_do_CSQ` state where it will conduct flow control according to the response to the AT command, which the

microcontroller should have received by now. Processing of all other AT commands are done in a similar manner.

```
void cellularHandler() {
  static cellular_handler_state mode;

  switch (mode) {
    case cellular_init:
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 3000;  // Specify how long we will wait in the next state
      mode = cellular_wait_RDY;
      break;

    case cellular_wait_RDY:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_do_RDY; // proceed
      }
      break;

    case cellular_do_RDY:
      rxBufferZero();
      rxBufferProcess();
      if ( isRxBufferContains("RDY") ) {
        mode = cellular_send_CSQ; // proceed
      } else {
        mode = cellular_init; // try again
      }
      break;
```

LISTING 4: Cellular module handler function snippet (cellularFunctions.cpp)

A brief summary of the logical progression of the cellular module FSM is as follows. Start in a waiting state and wait for the modem to send a `RDY` string. After that, send the `AT+CSQ` check signal quality command to the modem and wait until the response is anything other than `CSQ: 99,99` which should indicate that an acceptable signal has been acquired by the modem. The reason why the modem can directly look for a signal after powering on is because a one-time sequence of configuration AT commands has already been sent to the modem and the modem has stored the configuration settings into its non-volatile memory. [11]

```
AT+QCFG="nwscanseq",02,1     // use the LTE Cat-M1 radio access technology (RAT), take effect immediately
AT+QCFG="nwscanmode",3,1     // only allow the use of LTE RAT, take effect immediately
AT+QCFG="iotopmode",0,1      // With the LTE RAT, only allow the use of Cat M1, take effect immediately
AT+CFUN=1,0        // Allow full functionality to the mobile equipment, take effect without reset
AT+CGDCONT=1,"IP","telstra.m2m",, // packet data protocol type IP, set telstra.m2m as the access point
AT+COPS=1,2,"50501",8      // Connect to the telstra network operator specified by 50501, use Cat M1
```

LISTING 5: Cellular modem pre-configuration AT command sequence [11]

Then send `AT+QMTOPEN=5,"broker.hivemq.com",1883` to connect to a public MQTT broker. A response other than `QMTOPEN: 5,3` should indicate successful connection. Then send `AT+QMTCONN=5,"clientExample"` to register a client connection to the same MQTT broker. This AT command will only be performed once, since the MQTT broker seems to react unfavourably to repeated `AT+QMTCONN` connection requests. The firmware will instead query the broker for the connection status of the first connection request by using the `AT+QMTCONN?` command, and a response of `QMTCONN: 5,3` would indicate successful connection. [12]

To perform a single MQTT publish on the topic `IvanHu`, the `AT+QMTPUB=5,0,0,0,"IvanHu"` command is sent. The modem will prompt the microcontroller to start sending the data payload by printing a > character. After the microcontroller has sent its data payload, the microcontroller will signal end of data by sending a `0x1A` (substitute character in the ASCII encoding) followed by a `\r` (carriage return). [12]

The next sequence of AT commands deal with transmitting the file over a sequence of MQTT publishes. The microcontroller first publishes three comma separated values as a header. The header is necessary because the receiver must know this information in order to reconstitute the file properly. The first two values are the number of segments and the segment size respectively. The meaning of segment is the block of data that is to be published for the transmission of the actual file, because the entire file cannot be transmitted over a single MQTT publish. The modem can only support a maximum payload publish size of 1548 bytes [12]. In this firmware, the segment size is defined to be 1500. The number of segments is obtained by dividing the file size by 1500 and rounding up to the nearest integer. The third comma separated value is the size of the final segment, which can range from 1 to 1500.

```
    case cellular_send_QMTPUBhelloworldMsg:
      Serial.printf("%u,", sdFunctionObj.numSegments);
      Serial.printf("%u,", SEGMENT_SIZE);
      Serial.printf("%u", sdFunctionObj.lastSegmentSize);
      Serial.write(char(26));
      Serial.write("\r");
```

LISTING 6: Sending of the header (cellularFunctions.cpp)

After sending the header, the file is sent over a series of MQTT publishes each containing 1500 byte segment of the file with the exception of the last segment, which can contain 1 to 1500

bytes. After the file has been sent, the FSM will go into the `cellular_done` state where it will send the `AT+QLTS=2` command modem to obtain the date and time calculated from the latest time that the modem has obtained from the network [11]. This last step is not strictly necessary but it was a development aid as it serves as an indicator that the cellular FSM has successfully entered the `cellular_done` state. For further reference, Appendices A and B contains the full code for `cellularFunctions.h` and `cellularFunctions.cpp`.

3.1.2.5 Python development of the file download functionality

Chronologically speaking, toward the end of the cellular file transmission firmware development process, there was a need to develop the functionality for the receiver system in order to assure the functionality of the transmission system. For programming the receiver system, Python is chosen as a suitable language to expedite the development process, because it is widely and quantitatively regarded as a user-friendly language for basic procedural programs [13] and because it has an abundance of libraries that is suitable for this application.

```
import paho.mqtt.client as mqtt
MQTT_SERVER = "broker.hivemq.com"
MQTT_TOPIC = "IvanHu"

def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe(MQTT_TOPIC)

def on_message(client, userdata, msg):
    # Create a copy of userdata for use within this callback
    global client_userdata
    print("Initial: " + str(client_userdata))

    # handle the header
    if client_userdata["isHeaderReceived"] == 0:
        print("Header received, ", end = '')
        client_userdata["headerString"] = str(msg.payload).strip("b'")
        client_userdata["isHeaderReceived"] = 1
        print("isHeaderReceived set to " + str(client_userdata["isHeaderReceived"]))
    # handle payloads
    else:
        # if first segment
        if client_userdata["currentSegment"] == 1:
            f = open('inputjpg_encrypted_encoded', "wb")
            f.write(msg.payload)
            f.close()
            client_userdata["currentSegment"] += 1
        # if last segment
        elif client_userdata["currentSegment"] == int(client_userdata["headerString"].split
    (",")[0]):
            f = open('inputjpg_encrypted_encoded', "ab+")
            f.write(msg.payload)
            f.close()
            client_userdata["isHeaderReceived"] = 0
            client_userdata["currentSegment"] = 1
            client_userdata["headerString"] = ""
        else:
            f = open('inputjpg_encrypted_encoded', "ab+")
```

```
            f.write(msg.payload)
            f.close()
            client_userdata["currentSegment"] += 1
        print("Segment " + str(client_userdata["currentSegment"]) + " received.")

    print("Final: " + str(client_userdata))
    print("")

client_userdata = {
    "isHeaderReceived":     0,
    "currentSegment":       1,
    "headerString":      ""
}

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(MQTT_SERVER, 1883, 60)

client.loop_forever()
```

LISTING 7: Python MQTT downloader program (mqtt_subscriber_file.py)

`paho-mqtt` [14] is a Python library that provides functionality for a program to act as an MQTT client. The code structure consists of a run-once section prior to `client.loop_forever()` which serves as a infinite loop to keep the program open.

`on_connect()` and `on_message()` are functions which practically serve as Python equivalents to C interrupt routines. `on_connect()` is called when the program has successfully connected to an MQTT broker. In this case, a message is printed out signifying the success/failure of the connection and the client promptly subscribes to the same topic that the transmitter would be publishing the file on.

The main bulk of the development involves writing the `on_message()` function which handles parsing the series of MQTT publishes that make up the header and the segmented payload of the transmission. There was a need for `on_message()` to access variables persistent to numerous different function calls during this process. The dictionary `client_userdata` serves as a single memory location where different variables could be accessed. `client_userdata` is called with the `global` keyword within `on_message()` so that the function could modify the variable which is located outside of the function scope.

The first part of the function handles parsing of the header. The payload of the MQTT publish containing the header is stripped of trailing characters so it becomes a true comma-separated

string of 3 integers containing the number of segments, segment size and last segment size respectively. This comma-separated string is stored within `client_userdata`.

The next part of the function handles the case where the incoming MQTT publish is the first segment of the data payload. The function will open a file in binary writing mode and write the first 1500 bytes of the file. `currentSegment` within `client_userdata` is incremented to keep track of how many payload segments the program has received thus far. The next case handles an MQTT data payload publish that is neither the first nor last segment. In this case, the program will open the same file in binary append mode and append the next 1500 bytes to the file. The last case handles an MQTT data payload publish that is the last segment. In this case, the program will open the same file in binary append mode and append the last of the bytes into the file, and then resets `client_userdata` to its initial state so that the program will be ready for a subsequent file download sequence.

3.1.2.6 Firmware development of file processing functionalities

To address data security concerns, files should always be encrypted prior to transmission over the internet.

The function `void aesEncode(fs::FS &fs, char* srcDir, char* destDir)` was written to take a file with directory specified with `char* srcDir` and create an encrypted version of the file whose at `char* destDir`.

The `mbedtls/aes.h` [15] library uses AES128 to encrypt 16 byte buffers using a 16 byte key and outputs the encrypted result into another 16 byte buffer. The `aesEncode()` function will read the `srcDir` file in 16 bytes "chunks" at a time, use the `mbedtls/aes.h` to encrypt it, and then write those 16 encrypted bytes into `destDir`. This process is repeated until end of file. If the last "chunk" of the input file is less than 16 bytes in size, the `aesEncode()` function will pad it to size with the required number of `0xFF` bytes.

```
53 void aesEncode(fs::FS &fs, char* srcDir, char* destDir) {
54   //Serial.print("Commencing aesEncode\r\n");
55   sdFunctionObj.file = fs.open(srcDir, "rb");
56   sdFunctionObj.fileEncoded = fs.open(destDir, FILE_WRITE);
57
58   if(!sdFunctionObj.fileEncoded){
59       //Serial.println("Failed to open file for writing");
60       return;
```

```
  }

  char * key = AES_KEY;
  uint8_t paddingCount = 0;

  while (sdFunctionObj.file.available()) {
    uint8_t buf[AES_CHUNK];
    unsigned char encryptedBuf[AES_CHUNK];
    mbedtls_aes_context aes;

    // Write AES_CHUNK bytes into buf
    for (uint8_t i = 0; i < AES_CHUNK; i++) {
      if (sdFunctionObj.file.available()) {
        buf[i] = sdFunctionObj.file.read();
      } else {
        // Padding needed
        buf[i] = '0xFF';
        paddingCount++;
      }
    }

    // AES128 encrypt buf into encryptedBuf
    mbedtls_aes_init( &aes );
    mbedtls_aes_setkey_enc( &aes, (const unsigned char*) key, strlen(key) * 8 );
    mbedtls_aes_crypt_ecb(&aes, MBEDTLS_AES_ENCRYPT, (const unsigned char*)buf,
    encryptedBuf);
    mbedtls_aes_free( &aes );

    for (uint8_t i = 0; i < AES_CHUNK; i++) {
      sdFunctionObj.fileEncoded.write(encryptedBuf[i]);
      //Serial.print(encodedBuf.charAt(i));
    }
  }

  //Serial.print("File encrypted\r\n");
  sdFunctionObj.fileEncoded.close();
  sdFunctionObj.file.close();
}
```

LISTING 8: aesEncode() function (sdCardFunctions.cpp)

It was discovered that raw binary data, encrypted or not, cannot be fed directly into the LTE modem and published using MQTT. Doing so results in unsuccessful publishes. The reason for this is not definitively proven but it is likely due to the fact that some binary sequences can be interpreted by some intermediary network protocol to be special characters, and as a result breaking the protocol.

The `crypto/base64.h` [16] library encrypts buffers containing binary data of any size into another buffer of greater size than the original due to the nature of the encoding. It will even return the size of the output buffer to aid the user's development process. The `void base64Encode(fs::FS &fs, char* srcDir, char* destDir)` function was written to encode a binary file into a file that is wholly represented by ASCII characters. The operation of the function is similar to `aesEncode()`, except that the user-padding requirement is not a concern here because this library handles padding. The RFC 4648

standard of Base64 encoding specifies that padding will be added if at the end of the data being encoded, fewer than 24 bits are available [17]. Appendices E and F contain the full code for this module.

3.1.2.7 Python development of the file decoding and decryption functionalities

A Python program was subsequently developed to provide a means of recovering the original file from its encrypted and encoded analogue.

```
import base64
from Crypto.Cipher import AES
from pathlib import Path

while True:
    pw = input("Please enter the password:\n")
    if pw == "samplepassword":
        break

# Start of decoding process of a base64 encoded file into a binary AES128 encrypted file
imageEncryptedEncoded = open('inputjpg_encrypted_encoded', 'rb')
imageEncryptedEncoded_str = imageEncryptedEncoded.read()
imageEncrypted_str = base64.decodebytes(imageEncryptedEncoded_str)
# create a writable file and write the decoding result
imageEncrypted = open('image_encrypted', 'wb')
imageEncrypted.write(imageEncrypted_str)
imageEncryptedEncoded.close()
imageEncrypted.close()

# Start of decrypting process of a binary AES128 encrypted file into a JPEG image
# Calculate the nearest multiple of 16 that is lesser than the current file size
print("File size of encrypted, padded image: ")
padded_file_size = Path('image_encrypted').stat().st_size
print(str(padded_file_size))
print("nearest round-down 16-divisible byte size")
nearest_16_byte_size = padded_file_size - (padded_file_size % 16)
print(str(nearest_16_byte_size))

# Decrypt binary AES128 encrypted file into jpg
key = 'ivanhuIVANHUivhu'
# Encode the key string into UTF8 as is required by the Crypto.Cipher library
keyutf8 = str.encode(key)
input_file = open('image_encrypted', 'rb')
output_file = open('image_decrypted.jpg' , 'wb')
input_file_str = input_file.read()
# Trim encrypted, padded image to nearest 16 bytes
input_file_str = input_file_str[0:nearest_16_byte_size]
decipher = AES.new(keyutf8, AES.MODE_ECB)
output_file_str = decipher.decrypt(input_file_str)
output_file.write(output_file_str)
```

LISTING 9: Python Base64 decoder and AES128 decrypter program (aesbase64decode.py)

As a proof of concept of data privacy, any users of this program must first input the correct password. The severe security limitations of this implementation of password authentication will be addressed in the next chapter.

The `base64` package within the Python Standard Library provides functionality for Base64 decoding. As Python is a high-level language that is being run on a processor with significantly higher processing capabilities than an ESP32, the entire `input jpg_encrypted_encoded` file can be opened and its entire content can be stored in a single variable. The package then takes this variable and outputs the the decoded binary content into another variable. The program then writes this binary stream into a file called `image_encrypted`.

The `PyCryptodome` package provides various cryptographical tools for the Python language. The subpackage `Crypto.Cipher` is used for AES128 decryption. As is mentioned in the Section 3.1.2.6, the Base64 encoding process could potentially insert padding that will cause the file size to no longer be divisible by 16. Due to this, the program calculates the next lower number from the current file size that is divisible by 16. The program will then truncate the file to this size, then input it into the `Crypto.Cipher` library. The result is the recovered JPEG image which to human eyes, has no discernible difference to the original image that is taken by the embedded system.

3.1.2.8 Development of a HTML-based dashboard

To enhance the device compatibility of the system e.g. if the user wanted to download, decode and decrypt transmission from a device which cannot natively run Python, these functions must be integrated into a web-based platform.

Due to time constraints, it was determined that the most time-effective solution to this requirement was to somehow run the Python code already developed from a web page. An AWS Linux instance was set up to host the website through Apache. On the webpage, two HTML iframes embedding two SSH terminals to the Linux instance are implemented using `shellinabox` [18]. This is achievable because `shellinabox` is a webserver that implements and emulates an SSH terminal connection on port 4200. In this manner, the user can, from a single webpage, run the two aforementioned Python scripts for downloading and decoding/decrypting respectively.

FIGURE 9: HTML-based webpage dashboard

Additionally, from the same webpage, the user can view the recovered image. A "Refresh Image" button allows the user to refresh the image for subsequent downloads without refreshing the whole page so as not to compromise the login status of the two `shellinabox` iframes. The "Refresh Image" button calls a simple Javascript cache breaker function that retrieves the newly downloaded image and updates the `<img>` element. Without this cache breaker function, the user's web browser would simply display the same image that is stored in the web browser's cache.

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  document.getElementById("yourimage").src = "image_decrypted.jpg?random="+new Date().
    getTime();
}
</script>
</head>
<body>

<h2>Downloader</h2>
<iframe src="http://3.106.165.175:4200" width="800"></iframe>

<h2>Decrypter</h2>
<iframe src="http://3.106.165.175:4200" width="800"></iframe>

<h2>This is your image</h2>

<img id="yourimage" src="image_decrypted.jpg">

<button type="button" onclick="myFunction()">Refresh Image</button>

</body>
</html>
```

LISTING 10: HTML code for the webpage (index.html)

3.1.2.9 Firmware incorporation of the camera function

The most trivial part of the development is left until last. There already exist numerous tutorials and code examples on the internet regarding how to take a picture on an ESP32-CAM and store the result in the SD card. The code in this project is adapted from [19]. The `esp_camera.h` library [20] forms the main basis of this code.

With regards to the code, it is self-explanatory and there is not much that requires elaboration. See Appendices I and J for the full code.

CHAPTER 4

Results and Discussion

4.1 Cellular data usage

Using the file transmission scheme developed in this project, a 18093 byte file is able to be transmitted over the internet with the cellular service provider claiming the total volume of data transmitted to be 20686 bytes.

FIGURE 10: Transmitted file. Timezone is UTC (GMT +0)

```
-rw-r--r-- 1 root root 18093 Oct 21 00:53 inputjpg_encrypted_encoded
```

FIGURE 11: Cellular service provider. Timezone is GMT +11.

Session Start	Data Volume (KB)	Duration (Minutes)	Country	Carrier
Oct 21, 2020 11:53 am	20.686	60:01	AU	Telstra Mobile...

The transmission scheme designed in this project is able to achieve a 87.46% payload to total data usage ratio. From inspection, this figure looks quite optimised. To fully appreciate the efficiency gains of using MQTT as opposed to HTTP, it would be ideal to replicate this project using HTTP, and compare the results. However, this is outside the scope of this project.

4.2 Image characteristics

The camera is configured to capture a picture 640 by 480 pixel in dimension. The image quality in terms of lossiness is set to 10 from a scale of 1 to 63, with a lower number indicating

higher quality [20]. The image quality is good enough to discern black, size 72 Arial font printed on white paper from a distance of 1.5 meters.

FIGURE 12: The captured image.

Image resolution and quality is a compromise between visual information and cellular data usage. Depending on the specific application for the project transmission device, the `esp_camera.h` library allows for easy configuration of these parameters.

4.3 End-to-end process duration

To transmit a 18093 byte file, the project transmission device takes approximately 40 seconds from the time it is switched on to when the entire file is received on a Sydney-based AWS server [21]. This duration is subject to variation depending on the size of the image captured,

the wireless channel between the project transmission device and its base station, and the state of the IP route that the datagrams will traverse over the internet.

The following is an analysis on the factors contributing to the end-to-end process duration.

There are 26 seconds between when the project transmission device is turned on and when it starts uploading. For the most part, these 26 seconds cannot be further optimised, because of, firstly, factors out of my control and secondly, factors that have been optimised to the fullest extent without sacrificing system reliability.

Of these 26 seconds, 5 seconds each are allocated to waiting for the responses of the `AT+QMTOPEN` and `AT+QMTCONN` commands respectively. These allocated times are chosen through repeated experimentation to find the best compromise between waiting too long or not waiting long enough for the response to be intercepted. Such experimentation is necessary for these commands because the LTE modem's user manual states that the Maximum Response Time (MRT) for these commands are 75 seconds [12], and it would certainly be infeasible to wait the entire duration of the MRT for these commands. Another factor that has already been optimised is the 3 seconds waiting time for the `RDY` string from the modem.

However, for some other AT commands, for instance `AT+CSQ` have a MRT of 300ms. In this case, it would be feasible and best practice to wait the entire duration. The same applies for all AT commands with a specified MRT in the order of less than a few seconds.

The final factor in this 26 second duration that is entire out of my control is waiting for `AT+CSQ` to return a signal strength signifying connection to the base station. This is entirely dependant on the wireless channel quality between the project transmission device and the base station.

For the remainder approximately 14 seconds, the project transmission device repeatedly publishes 1500 byte segments of the file. This time can be divided into actual transmission time which is dependant on the baud rate of the serial port between the microncontroller and the LTE modem, and the waiting time for the response of the moodem to the `AT+QMTPUB` command. Since the baud rate is 115200 with 8 data bits and one stop bit, the true data rate

is 102400 bits per second. Therefore, it takes approximately 1414 milliseconds to transmit 18093 bytes of data plus a 10 byte header. And since the header plus payload is divided into 14 publishes in total with a 500ms wait time after each publish, 7000 milliseconds is spent during this waiting period. It is assumed that the rest of the unaccounted-for time within this 14 second duration are due to other latency factors out of my control.

Therefore, it can be concluded that the main factor (7 seconds out of 14) contributing to the time spent during the transmission phase is the AT command response waiting time. As the user guide specifies an unrealistically long MRT of 15 seconds for `AT+QMTPUB` [12], it has been experimentally found that the lowest safe waiting time is 500ms and it would be unwise to compromise system reliability by reducing it further. Naturally, as the size of the file being transmitted increases, the number of separate MQTT publishes to transmit the entire file increases, leading to the proportion of time taken by the AT command response waiting time also increasing. This means that optimisation of serial transmission time through increasing baud rate becomes less impactful as the file size increases. Assuming that the image resolution and quality (and therefore size) should not be further reduced in order to preserve visual information, it can be concluded that little can be done to optimise transmission time.

4.4 Power usage

When the project transmission device is turned on, its average current consumption from the power source is approximately $190\,\mathrm{mA}$.

Due to time constraints, the power latch circuit has not been successfully implemented. However, simulation of the power latch circuit shows that it consumes a quiescent current of $8.885\,\mu\mathrm{A}$ (see Appendix L).

Based on these figures, if the project transmission device is to perform its functions three times every hour, with the remainder of the time being turned off, the project transmission device would have an average current draw of:

$$\frac{3 \times 0.19 \times 40 + 8.885 \times 10^{-6} \times 3480}{3600} \approx 6.3mA$$

Accounting for the 5V power source, the project transmission device draws an average of $0.0315\,\mathrm{W}$.

To calculate the battery life if a project transmission device were to use a typical 18650 lithium-ion battery and a boost converter to 5V, let us consider the following parameters. Suppose the battery has a current capacity of $2600\,\mathrm{mA\,h}$ at 3.7V, and the MCP16251 boost converter with a typical efficiency of 96% [22] is used. The battery capacity would have a power capacity of 9.62Wh. This means it can sustain the project transmission device for over 290 hours. If we consider solar energy as a means of energy harvesting for this application, it would take over 12 days of zero solar input to deplete the battery.

Therefore, the energy consumption of this device is low enough such that it is feasible to be deployed at a remote location with solar energy harvesting to enable continuous operation from an energy consumption standpoint.

4.4.1 Data security

Data security in the form of end-to-end encryption is crucial in this project because the project transmission device, by default, is configured in firmware to use a public MQTT test broker `broker.hivemq.com` on a public test topic `IvanHu` with no MQTT-layer authentication. This means that anyone connected to this broker can subscribe to the same topic and be able to have access to any data that the project transmission device publishes. It is therefore crucial from a data security standpoint that the project transmission device encrypts any published data so that even if the encrypted data is intercepted, no information can be discerned from it. It is not sufficient to rely on Base64 alone to obfuscate the data because this encoding scheme is ubiquitously used and can be recognised easily by inspection.

The fact that the MQTT broker acts as an intermediary between the project transmission device and the receiver enhances the data security of the system as a whole. In a traditional

source-to-destination data transmission protocol, for instance HTTP using a POST request, data interception would result in information pertaining to the address of the destination to be divulged to the attacker. Once this is known, the attacker can then attempt to find and exploit vulnerabilities at the destination. However, to attack this project's system, the most likely manner that malicious attackers would attempt to intercept data is by monitoring inbound traffic at the broker. But after this, it becomes very difficult for the attacker to work out the address of the destination, because the the data is directed to the rightful receiver of the data by way of a client-oriented MQTT subscription. The only way that the attacker could conceivably work out the destination address is by sampling all outbound traffic from the MQTT broker and analyse it all, which would be a very impractical task in the case of a public MQTT broker.

Changing topics from MQTT to the web-based dashboard at the receiver, the implementation of the dashboard presents a very serious security vulnerability. In order for a user to be able to use the downloader and Decoder/Decrypter functions, the user must be given root access to the Linux backend of the receiver system. Should the password-protected root access be compromised, the attacker can directly access the Python scripts which contain the AES128 key as an unobfuscated string. To address this issue, the Python scripts providing the downloader and Decoder/Decrypter functionalities should be ported to a proper web framework and any notion of being able to SSH into the backend of the server should absolutely be removed from the front end i.e. the webpage.

4.4.2 Data privacy

The concept of data privacy is addressed by this project on a proof-of-concept level through the separation of the download versus decoding/decryption functions at the receiver. Having a separate password before access to the decoding/decryption functions serves as a representation that there should be two levels of security clearance for the personnel manning the receiver system; one to download and one to actually recover the originally-sent data.

The implementation used in this project is quite vulnerable. The password is simply stored as an unobfuscated string within the Python script for the Decoder/Decrypter which can easily be obtained by opening the script with a text editor. A more robust password authentication system is required for proper implementation of data privacy measures.

CHAPTER 5

Conclusion

This project has successfully implemented an embedded system that is able to, upon powering on, take a photograph, encrypt and encode the photograph, publish the photograph as a series of MQTT publishes to an MQTT broker. This project has also successfully implemented a web application that is able to subscribe to and "listen for" the file at the same MQTT broker, download the file, and then perform decoding and decryption in order to recover the original photograph.

The use of MQTT as a protocol in the context of file transmission is unorthodox, but theoretically provides data usage efficiency gains compared to other protocols such as HTTP. It also provides a slight edge in data security over HTTP as incoming MQTT backets to the broker does not store any information about the address of the intended recipient.

The use of a digital pulse-trigger latch circuit as a means of switching power on and off to the project device was not realised, but simulations show that it is a functional solution.

With no fundamental conceptual shortcomings which could not be fixed with further refinements, the development undertaken within the course of this project can be eventually used in a real-life production environment.

5.1 Future outlook

In addition to all the problems of the current state of the project, there are still some feature additions that could be made to this project which would enhance various aspects of the project in a production environment.

One manner in which user-friendliness could be increased is the use of a configuration text file stored in the SD card. Currently, all configurations (e.g. MQTT broker address, camera settings etc.) are hardcoded into the firmware. Storing configurable variables into a text file on the SD card would enable configuration without having to reprogram the embedded system with new firmware. Even though it is obvious that this feature is a must-have if the project were to progress into a marketable product, some reasons are outlined as follows. Firstly, the source code is regarded as proprietary intellectual property which must not be exposed to the public in order to preserve the exclusive right of the developer to profit from the intellectual property, and in the project's current state, configuration would require access to the source code. Secondly, the public would not be able to reprogram the project device without specialty electronics such as a USB to UART converter, whereas files could easily be put onto an SD card using consumer electronics.

To further optimise quiescent power consumption, a latching relay instead of the current circuit involving MOSFETs and transistors could be used to toggle input power to the project device. With the use of a latching relay, an energising pulse is provided to the relay to toggle its contact position, after which point the project device is physically isolated from the power supply, thereby drawing zero current.

Cellular data usage is a critical factor to consider in a production environment. Currently, the use of Base64 as a method to encode data comes at a cost of a 133% increase in size of the data being encoded. Base64 was chosen as an encoding scheme for this project due to its ubiquity and consequently the increase in development speed stemming from the use of this technology. However, more novel encoding schemes such as Base-122, boasting an additional approximately 14% encoding efficiency over Base64 [23] can be explored in future efforts to further optimise cellular data usage.

Currently, some of the information sent in the header MQTT publish prior to the main file upload is not used by the receiver file processing processes. Only the number of segments is used by the downloader to determine whether the latest MQTT publish successfully downloaded is the final segment of the file. The size of each segment and the size of the last

segment remains unused. Further development can either make use of this information to optimise the project process, or cut it out to optimise cellular data usage.

Currently, true data integrity is not preserved in the project process. As a result, the recovered file at the receiver is not identical to the one that is originally taken by the embedded system. The files are identical until the close to the end of file.

FIGURE 13: Comparison of the input file (left) and the recovered file (right).

```
00003240 00 14 50 01 45 00 14 50 07 10 69 2b cf 3c
00003250 50 31 28 a0 04 a2 90 05 25 00 14 94 ca 0a
00003260 4a 28 00 a2 90 09 45 31 89 45 20 0a 33 40
00003270 29 0c 4a 29 80 94 50 01 9a 4a 06 14 e3 40
00003280 90 0e a2 a4 05 3d 2a 24 eb 4c 09 bb 50 b5
00003290 2a bc 7f f1 f0 f4 08 b1 52 47 d6 91 27 4f
000032a0 00 c7 ab ff 00 bf 5a 15 e8 52 f8 51 d7 4f
000032b0 ce d6 ff 00 e3 c3 e8 e2 b1 aa 29 98 06 b5
000032c0 7e 7b 85 fa 1a e5 39 cd ca 2b 3b 6a 5f 27
000032d0 a2 ba 22 74 05 15 e8 c7 62 82 8a a0 0a 28
000032e0 80 0a 28 00 a2 80 0a 28 00 a2 80 0a 28 00
000032f0 0a 28 00 a2 80 38 7a 6d 79 e7 9a 14 94 80
00003300 31 85 25 20 12 8a 43 0a 4c d3 00 cd 14 58
00003310 48 2c 25 14 00 51 4a e3 12 8a 60 21 35 11
00003320 19 a7 83 40 0b 46 68 01 29 33 40 c3 34 94
00003330 e3 40 c4 a7 52 01 69 6a 40 53 d2 a2 5a 68
00003340 f2 d0 a6 93 01 eb 50 2f fc 7d 35 22 4b 14
00003350 48 93 a8 d1 7f e3 d5 bf de ad 1a f4 28 3b
00003360 a7 f0 85 50 d6 41 6d 35 f0 3b 8f e7 58 4f
00003370 a1 ce bf 4a d2 f0 ff 00 fc 7d 4d fe e0 ae
00003380 7e 8a 46 c1 45 68 8d 02 8a f4 a9 fc 25 85
00003390 14 50 01 45 00 14 50 01 45 00 14 50 01 45
000033a0 50 01 45 00 14 50 01 45 00 7f ff d9 0a 09
000033b0 09 09 09 09 09 09 09 09 09 09 09 09 09 09
000033c0 20 20 20
```

```
00003270 29 0c 4a 29 80 94 50 01 9a 4a 06 14 e3 40
00003280 90 0e a2 a4 05 3d 2a 24 eb 4c 09 bb 50 b5
00003290 2a bc 7f f1 f0 f4 08 b1 52 47 d6 91 27 4f
000032a0 00 c7 ab ff 00 bf 5a 15 e8 52 f8 51 d7 4f
000032b0 ce d6 ff 00 e3 c3 e8 e2 b1 aa 29 98 06 b5
000032c0 7e 7b 85 fa 1a e5 39 cd ca 2b 3b 6a 5f 27
000032d0 a2 ba 22 74 05 15 e8 c7 62 82 8a a0 0a 28
000032e0 80 0a 28 00 a2 80 0a 28 00 a2 80 0a 28 00
000032f0 0a 28 00 a2 80 38 7a 6d 79 e7 9a 14 94 80
00003300 31 85 25 20 12 8a 43 0a 4c d3 00 cd 14 58
00003310 48 2c 25 14 00 51 4a e3 12 8a 60 21 35 11
00003320 19 a7 83 40 0b 46 68 01 29 33 40 c3 34 94
00003330 e3 40 c4 a7 52 01 69 6a 40 53 d2 a2 5a 68
00003340 f2 d0 a6 93 01 eb 50 2f fc 7d 35 22 4b 14
00003350 48 93 a8 d1 7f e3 d5 bf de ad 1a f4 28 3b
00003360 a7 f0 85 50 d6 41 6d 35 f0 3b 8f e7 58 4f
00003370 a1 ce bf 4a d2 f0 ff 00 fc 7d 4d fe e0 ae
00003380 7e 8a 46 c1 45 68 8d 02 8a f4 a9 fc 25 85
00003390 14 50 01 45 00 14 50 01 45 00 14 50 01 45
000033a0 50 01 45 00 14 50 01 45 00 7f ff d9 46 46
000033b0 2e c3 18 b4 1d 7c 0a c8 5c 0c 7c 15 74 97
000033c0 dc 3a f2 a2 f9 71 08 a6 6b 6b 32 bb f0 33
000033d0 56 48 15 35 70 da f4 ad 6b e4 8d aa 08 af
000033e0 8f d6 00 1c d2 dd 52 1a a9 05 d4 0c 25 26
000033f0 62 a0 a9 e6 02 17 76 db 62 4b ab 81 0d 48
```

However, this is of no practical consequence because the recovered image can still be opened and represented as an image by any image-viewing software, and there is no human-perceptible difference between the original and the recovered images. This issue was discovered during the development process, but does to its practically inconsequential nature, it was never properly investigated. However, should the application for this project be extended to accuracy-critical data types, this issue should be investigated and fixed.

Bibliography

[1] Gene Ross and Mark Fletcher. (Dec. 2019). 'Html-embedded mqtt file receiver demo,' [Online]. Available: https://www.senninger.com/sites/senninger.hunterindustries.com/files/understanding-pressure-regulation-white-paper.pdf.

[2] E. Habib, W. Krajewski and A. Kruger, 'Sampling errors of tipping-bucket rain gauge measurements,' *Journal of Hydrologic Engineering*, vol. 6, Apr. 2001. DOI: 10.1061/(ASCE)1084-0699(2001)6:2(159).

[3] O. Liberg, M. Sundberg, E. Wang, J. Bergman and J. Sachs, *Cellular Internet of Things: Technologies, Standards, and Performance*. Elsevier Science, 2017, ISBN: 9780128124598. [Online]. Available: https://books.google.com.au/books?id=n9wkDwAAQBAJ.

[4] J. Bergman, *Enhanced 4g lte coverage for machine-type communications and internet of things*, Apr. 2017. [Online]. Available: https://www.ericsson.com/en/blog/2017/4/enhanced-4g-lte-coverage-for-machine-type-communications-and-internet-of-things.

[5] N. Naik, 'Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http,' in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7. DOI: 10.1109/SysEng.2017.8088251.

[6] S. Bandyopadhyay and A. Bhattacharyya, 'Lightweight internet protocols for web enablement of sensors using constrained gateway devices,' in *2013 International Conference on Computing, Networking and Communications (ICNC)*, 2013, pp. 334–340. DOI: 10.1109/ICCNC.2013.6504105.

[7] HiveMQ. (Feb. 2015). 'Quality of service 0,1 & 2 - mqtt essentials: Part 6,' [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

[8] R. Jejurikar, C. Pereira and R. Gupta, 'Leakage aware dynamic voltage scaling for real-time embedded systems.,' Jan. 2004, pp. 275–280. DOI: 10.1109/DAC.2004.240321.

[9] D. Jones. (Mar. 2012). 'Eevblog #262 - world's simplest soft latching power switch circuit,' [Online]. Available: https://www.youtube.com/watch?v=Foc9R0dC2iI%5C&t=243s.

[10] Espressif. (Oct. 2020). 'Configure other jtag interface,' [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/jtag-debugging/configure-other-jtag.html.

[11] Quectel, *Bg96 at commands manual*, 2.1, Quectel, Building 5, Shanghai Business Park Phase III (Area B), No.1016 Tianlin Road, Minhang District, Shanghai 200233, China, May 2018. [Online]. Available: https://www.quectel.com/UploadImage/Download/Quectel_BG96_AT_Commands_Manual_V2.1.pdf.

[12] ——, *Bg96 mqtt application note*, 1.0, Quectel, Building 5, Shanghai Business Park Phase III (Area B), No.1016 Tianlin Road, Minhang District, Shanghai 200233, China, Jan. 2018. [Online]. Available: https://sixfab.com/wp-content/uploads/2018/09/Quectel_BG96_MQTT_Application_Note_V1.0.pdf.

[13] A. Jayal, S. Lauria, A. Tucker and S. Swift, 'Python for teaching introductory programming: A quantitative evaluation,' *Innovation in Teaching and Learning in Information and Computer Sciences*, vol. 10, no. 1, pp. 86–90, 2011. DOI: 10.11120/ital.2011.10010086. eprint: https://doi.org/10.11120/ital.2011.10010086. [Online]. Available: https://doi.org/10.11120/ital.2011.10010086.

[14] Roger Light, *Paho-mqtt*, version 1.5.1, 23rd Sep. 2020. [Online]. Available: https://pypi.org/project/paho-mqtt/.

[15] github.com/me-no-dev, *Mbedtls-aes.h*, version commit 1977370, 21st Jan. 2020. [Online]. Available: https://github.com/espressif/arduino-esp32/blob/master/tools/sdk/include/mbedtls/mbedtls/aes.h.

[16] ——, *Crypto-base64.h*, version commit 7e0811e, 14th Nov. 2016. [Online]. Available: https://github.com/espressif/arduino-esp32/blob/master/tools/sdk/include/wpa_supplicant/crypto/base64.h.

[17] S. Josefsson. (Oct. 2006). 'The base16, base32, and base64 data encodings,' [Online]. Available: https://tools.ietf.org/html/rfc4648#section-4.

[18] Marc Singer, *Shellinabox*, version 2.20, 10th Nov. 2016. [Online]. Available: https://github.com/shellinabox/shellinabox.

[19] R. Santos. (Oct. 2020). 'Change esp32-cam ov2640 camera settings: Brightness, resolution, quality, contrast, and more,' [Online]. Available: https://randomnerdtutorials.com/esp32-cam-ov2640-camera-settings/.

[20] Espressif Systems, *Esp_camera.h*, version commit 4af111a, 16th Mar. 2020. [Online]. Available: https://github.com/espressif/esp32-camera/blob/master/driver/include/esp_camera.h.

[21] I. Hu. (Oct. 2020). 'Html-embedded mqtt file receiver demo,' [Online]. Available: https://www.youtube.com/watch?v=SwMuQBwRPd8.

[22] Microchip, *Mcp16251/2 datasheet*, Revision B, Microchip, 2355 W Chandler Blvd, Chandler, AZ 85224, USA, Sep. 2016. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/20005173B.pdf.

[23] Kevin Albertson, *Base122*, version commit 060c560, 29th Aug. 2020. [Online]. Available: https://github.com/kevinAlbs/Base122.

Appendix A: cellularFunction.h

```
#ifndef __CELLULAR_FUNCTIONS_H__
#define __CELLULAR_FUNCTIONS_H__

#include "Arduino.h"
#include <stdint.h>
#include <string.h>

#define TX_BUFFER_LEN 129
#define RX_BUFFER_LEN 129
#define FAST_CATM1_BAUDRATE 115200

// flag_reg defines
#define CELLULAR_DONE_FLAG 0x01

typedef struct {
  char tx_buffer[TX_BUFFER_LEN];
  char rx_buffer[RX_BUFFER_LEN];
  unsigned long tx_wait_duration_ms;
  unsigned long tx_start_time_ms;
  uint8_t flags_reg = 0;
} cellularObj_t;

extern cellularObj_t cellularObj;

void txBufferSend(uint8_t bytes_to_send);
void rxBufferZero(void);
void rxBufferProcess(void);
void cellularHandler(void);
uint8_t isRxBufferContains(char* substr);
uint8_t isRxBufferAllZero(void);
void printRxBuffer(void);

#endif
```

LISTING 11: cellularFunctions.h

Appendix B: cellularFunction.cpp

```
// Non blocking state machine for processing AT commands

#include "cellularFunctions.h"
#include "sdCardFunctions.h"

cellularObj_t cellularObj;

typedef enum {
  cellular_init,
  cellular_wait_RDY,
  cellular_do_RDY,
  cellular_send_CSQ,
  cellular_wait_CSQ,
  cellular_do_CSQ,
  cellular_send_QMTOPEN,
  cellular_wait_QMTOPEN,
  cellular_do_QMTOPEN,
  cellular_send_QMTCONN,
  cellular_wait_QMTCONN,
  cellular_send_QMTCONNquery,
  cellular_wait_QMTCONNquery,
  cellular_do_QMTCONNquery,
  cellular_send_QMTPUBCmd,
  cellular_wait_QMTPUBCmd,
  cellular_do_QMTPUBCmd,
  cellular_send_QMTPUBhelloworldMsg,
  cellular_wait_QMTPUBhelloworldMsg,
  cellular_do_QMTPUBhelloworldMsg,
  cellular_send_QMTPUBfileCmd,
  cellular_wait_QMTPUBfileCmd,
  cellular_send_QMTPUBfile,
  cellular_wait_QMTPUBfile,
  cellular_do_QMTPUBfile,
  cellular_done
} cellular_handler_state;

// VERY IMPORTANT TO KNOW LENGTH
// Write the contents of the txBuffer to the serial only up to the specified length
// And then set the rest of the buffer to zero
void txBufferSend(uint8_t bytes_to_send) {
  if (bytes_to_send > TX_BUFFER_LEN) {
    Serial.write("ERROR: bytes_to_send exceeded TX_BUFFER_LEN");
    return;
  }

  for (uint8_t i = 0; i <= bytes_to_send - 1; i++) {
    Serial.write(cellularObj.tx_buffer[i]);
    cellularObj.tx_buffer[i] = 0;
  }

  for (uint8_t i = bytes_to_send; i <= TX_BUFFER_LEN-1; i++) {
    cellularObj.tx_buffer[i] = 0;
  }
}

void rxBufferZero(void) {
  for (uint8_t i = 0; i < RX_BUFFER_LEN; i++) {
    cellularObj.rx_buffer[i] = '\0';
  }
}

// We ASSUME that there's content in the Serial buffer, read the first RX_BUFFER_LEN bytes
    into the rxBuffer
// Zero the rest of the buffer if the message is not long enough to fill the buffer
void rxBufferProcess(void) {
  // null terminate
  cellularObj.rx_buffer[RX_BUFFER_LEN - 1] = '\0';

  // Loop up to and including the penultimate byte (ignore the null terminator)
  for (uint8_t i = 0; i < RX_BUFFER_LEN - 1; i++) {
```

```
    if (Serial.available()) {
      cellularObj.rx_buffer[i] = Serial.read();
    } else {
      cellularObj.rx_buffer[i] = '\0';
    }
  }
}

uint8_t isRxBufferContains(char* substr) {
  if (strstr(cellularObj.rx_buffer, substr) == NULL) {
    //Serial.write("does not contain");
    return 0;
  }
  //Serial.write("does contain");
  return 1;
}

// this function is useful for determining whether the buffer is all zero
// and therefore, received no response
uint8_t isRxBufferAllZero(void) {
  for (uint8_t i = 0; i < RX_BUFFER_LEN; i++) {
    if (cellularObj.rx_buffer[i] != '\0') {
      return 0; // not all Zeroes
    }
  }
  return 1; // All zeroes
}

void printRxBuffer(void) {
  for (uint8_t i = 0; i < RX_BUFFER_LEN; i++) {
    Serial.write(cellularObj.rx_buffer[i]);
  }
}

void cellularHandler() {
  static cellular_handler_state mode;

  switch (mode) {
    case cellular_init:
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 3000;  // Specify how long we will wait in the next state
      mode = cellular_wait_RDY;
      break;

    case cellular_wait_RDY:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_do_RDY; // proceed
      }
      break;

    case cellular_do_RDY:
      rxBufferZero();
      rxBufferProcess();
      if ( isRxBufferContains("RDY") ) {
        mode = cellular_send_CSQ; // proceed
      } else {
        mode = cellular_init; // try again
      }
      break;

    case cellular_send_CSQ:
      Serial.write("AT+CSQ\r\n");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 300;  // Specify how long we will wait in the next state
      mode = cellular_wait_CSQ;
      break;

    case cellular_wait_CSQ:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
```

```
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_do_CSQ; // proceed
      }
      break;

    case cellular_do_CSQ:
      rxBufferZero();
      rxBufferProcess();
      //printRxBuffer();
      // If contains +CSQ: 99,99 or NO_RESPONSE, try again
      if (isRxBufferContains("CSQ: 99,99") || isRxBufferAllZero()) {
        mode = cellular_send_CSQ; // try again
      } else {
        mode = cellular_send_QMTOPEN; // proceed
      }
      break;

    case cellular_send_QMTOPEN:
      Serial.write("AT+QMTOPEN=5,\"broker.hivemq.com\",1883\r\n");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 5000;  // Specify how long we will wait in the next state
      mode = cellular_wait_QMTOPEN;
      break;

    case cellular_wait_QMTOPEN:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_do_QMTOPEN; // proceed
      }
      break;

    case cellular_do_QMTOPEN:
      rxBufferZero();
      rxBufferProcess();
      if ( isRxBufferContains("QMTOPEN: 5,3") || isRxBufferAllZero() ) {
        mode = cellular_send_QMTOPEN; // try again
      } else {
        mode = cellular_send_QMTCONN; // proceed
      }
      break;

    case cellular_send_QMTCONN:
      Serial.write("AT+QMTCONN=5,\"clientExample\"\r\n");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 5000;  // Specify how long we will wait in the next state
      mode = cellular_wait_QMTCONN;
      break;

    case cellular_wait_QMTCONN:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_send_QMTCONNquery; // proceed
      }
      break;

    case cellular_send_QMTCONNquery:
      Serial.write("AT+QMTCONN?\r\n");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 500;  // Specify how long we will wait in the next state
      mode = cellular_wait_QMTCONNquery;
      break;

    case cellular_wait_QMTCONNquery:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
```

```
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_do_QMTCONNquery; // proceed
      }
      break;

    case cellular_do_QMTCONNquery:
      rxBufferZero();
      rxBufferProcess();
      if ( isRxBufferContains("QMTCONN: 5,3") ) {
        mode = cellular_send_QMTPUBCmd; // proceed
      } else {
        mode = cellular_send_QMTCONN; // try again
      }
      break;

    case cellular_send_QMTPUBCmd:
      Serial.write("AT+QMTPUB=5,0,0,0,\"IvanHu\"\r\n");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 500;  // Specify how long we will wait in the next state
      mode = cellular_wait_QMTPUBCmd;
      break;

    case cellular_wait_QMTPUBCmd:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_send_QMTPUBhelloworldMsg; // proceed
      }
      break;

    case cellular_send_QMTPUBhelloworldMsg:
      Serial.printf("%u,", sdFunctionObj.numSegments);
      Serial.printf("%u,", SEGMENT_SIZE);
      Serial.printf("%u", sdFunctionObj.lastSegmentSize);
      Serial.write(char(26));
      Serial.write("\r");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 500;  // Specify how long we will wait in the next state
      mode = cellular_wait_QMTPUBhelloworldMsg;
      break;

    case cellular_wait_QMTPUBhelloworldMsg:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_do_QMTPUBhelloworldMsg; // proceed
      }
      break;

    case cellular_do_QMTPUBhelloworldMsg:
      rxBufferZero();
      rxBufferProcess();
      //printRxBuffer();
      if ( isRxBufferContains("QMTPUB: 5,0,0") ) {
        sdFunctionObj.file = SD_MMC.open("/inputjpg_encrypted_encoded"); // Open file
        mode = cellular_send_QMTPUBfileCmd; // proceed
      } else {
        cellularObj.tx_start_time_ms = millis(); // Save the current time
        cellularObj.tx_wait_duration_ms = 500;  // Specify how long we will wait in the next state
        mode = cellular_wait_QMTPUBhelloworldMsg; // wait a little longer
      }
      break;

    case cellular_send_QMTPUBfileCmd:
      Serial.write("AT+QMTPUB=5,0,0,0,\"IvanHu\"\r\n");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 500;  // Specify how long we will wait in the next state
```

```
      mode = cellular_wait_QMTPUBfileCmd;
      break;

    case cellular_wait_QMTPUBfileCmd:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_send_QMTPUBfile; // proceed
      }
      break;

    case cellular_send_QMTPUBfile:
      for (int i = 0; i < SEGMENT_SIZE; i++) {
        if (sdFunctionObj.fileCounter < sdFunctionObj.fileSize) {
          Serial.write(sdFunctionObj.file.read());
          sdFunctionObj.fileCounter++;
        }
      }
      Serial.write(char(26));
      Serial.write("\r");
      cellularObj.tx_start_time_ms = millis(); // Save the current time
      cellularObj.tx_wait_duration_ms = 500;  // Specify how long we will wait in the next state
      mode = cellular_wait_QMTPUBfile;
      break;

    case cellular_wait_QMTPUBfile:
      if (millis() - cellularObj.tx_start_time_ms > cellularObj.tx_wait_duration_ms) {
        // wait duration has elapsed
        cellularObj.tx_start_time_ms = 0;
        cellularObj.tx_wait_duration_ms = 0;
        mode = cellular_do_QMTPUBfile; // proceed
      }
      break;

    case cellular_do_QMTPUBfile:
        sdFunctionObj.numSegments--; // decrement numSegments because we will be using this to decide whether to send another pub
        if (sdFunctionObj.numSegments) {
          // if there's still segments to send
          mode = cellular_send_QMTPUBfileCmd;
        } else {
          mode = cellular_done; // done
        }
      break;

    case cellular_done:
      if (! (cellularObj.flags_reg & CELLULAR_DONE_FLAG)) {
        sdFunctionObj.file.close();
        Serial.write("AT+QLTS=2\r\n");
        cellularObj.flags_reg |= CELLULAR_DONE_FLAG;
      }
      break;
  }
}
```

LISTING 12: cellularFunctions.cpp

Appendix C: mqtt_subscriber_file.py

See Section 3.1.2.5 for the full code.

Appendix E: sdCardFunctions.h

```
#ifndef __SD_CARD_FUNCTIONS_H__
#define __SD_CARD_FUNCTIONS_H__

#define SEGMENT_SIZE 1500
// Bytes to encode into Base64 at a time
#define BASE_64_CHUNK 120
#define AES_CHUNK 16
#define AES_KEY "ivanhuIVANHUivhu"

#include "Arduino.h"
#include "FS.h"                // SD Card ESP32
#include "SD_MMC.h"            // SD Card ESP32
#include "mbedtls/aes.h"
extern "C" {
#include "crypto/base64.h"
}

typedef struct {
  uint8_t flags_reg;
  uint32_t fileSize;
  uint16_t numSegments;
  uint32_t lastSegmentSize;
  uint32_t fileCounter; // variable for keeping track of how many bytes have been read
  File file;
  File fileEncoded;
} sdFunctionObj_t;

extern sdFunctionObj_t sdFunctionObj;

uint8_t sdInit(void);
void getFileSizeSegment(fs::FS &fs, char* dir);
void sdTest(fs::FS &fs);
void aesEncode(fs::FS &fs, char* srcDir, char* destDir);
void base64Encode(fs::FS &fs, char* srcDir, char* destDir);

#endif
```

LISTING 13: sdCardFunctions.h

Appendix F: sdCardFunctions.cpp

```
#include "sdCardFunctions.h"

sdFunctionObj_t sdFunctionObj;

uint8_t sdInit(void) {
  sdFunctionObj.fileCounter = 0;

  if(!SD_MMC.begin()){
    return 1;
  }
}

void getFileSizeSegment(fs::FS &fs, char* dir) {
  sdFunctionObj.file = fs.open(dir);
  uint32_t counter = 0;
  while (sdFunctionObj.file.available()) {
    sdFunctionObj.file.read();
    counter++;
  }
  sdFunctionObj.fileSize = counter;
  sdFunctionObj.numSegments = (sdFunctionObj.fileSize / SEGMENT_SIZE) + 1;
  sdFunctionObj.lastSegmentSize = sdFunctionObj.fileSize % SEGMENT_SIZE;
  sdFunctionObj.file.close();
}

void sdTest(fs::FS &fs) {
  sdFunctionObj.file = fs.open("/inputjpg_encrypted_encoded");
  uint32_t fileSize = 0;
  while (sdFunctionObj.file.available()) {
    sdFunctionObj.file.read();
    fileSize++;
  }
  Serial.printf("\r\nfileSize = %u\r\n", fileSize);
  sdFunctionObj.file.close();

  sdFunctionObj.file = fs.open("/inputjpg_encrypted_encoded");
  uint8_t numSegments = (fileSize / SEGMENT_SIZE) + 1;
  Serial.printf("\r\nSend %u segments\r\n", numSegments);

  int fileCounter = 0;
  for (int j = 0; j < numSegments; j++) {
    Serial.printf("\r\nSegment %u:\r\n", j);
    for (int i = 0; i < SEGMENT_SIZE; i++) {
      if (fileCounter < fileSize) {
        Serial.write(sdFunctionObj.file.read());
        fileCounter++;
      }
    }
  }
  sdFunctionObj.file.close();
}

void aesEncode(fs::FS &fs, char* srcDir, char* destDir) {
  //Serial.print("Commencing aesEncode\r\n");
  sdFunctionObj.file = fs.open(srcDir, "rb");
  sdFunctionObj.fileEncoded = fs.open(destDir, FILE_WRITE);

  if(!sdFunctionObj.fileEncoded){
      //Serial.println("Failed to open file for writing");
      return;
  }

  char * key = AES_KEY;
  uint8_t paddingCount = 0;

  while (sdFunctionObj.file.available()) {
    uint8_t buf[AES_CHUNK];
    unsigned char encryptedBuf[AES_CHUNK];
    mbedtls_aes_context aes;

```

```
    // Write AES_CHUNK bytes into buf
    for (uint8_t i = 0; i < AES_CHUNK; i++) {
      if (sdFunctionObj.file.available()) {
        buf[i] = sdFunctionObj.file.read();
      } else {
        // Padding needed
        buf[i] = '0xFF';
        paddingCount++;
      }
    }

    // AES128 encrypt buf into encryptedBuf
    mbedtls_aes_init( &aes );
    mbedtls_aes_setkey_enc( &aes, (const unsigned char*) key, strlen(key) * 8 );
    mbedtls_aes_crypt_ecb(&aes, MBEDTLS_AES_ENCRYPT, (const unsigned char*)buf,
    encryptedBuf);
    mbedtls_aes_free( &aes );

    for (uint8_t i = 0; i < AES_CHUNK; i++) {
      sdFunctionObj.fileEncoded.write(encryptedBuf[i]);
      //Serial.print(encodedBuf.charAt(i));
    }
  }

  //Serial.print("File encrypted\r\n");
  sdFunctionObj.fileEncoded.close();
  sdFunctionObj.file.close();
}

void base64Encode(fs::FS &fs, char* srcDir, char* destDir) {
  //Serial.print("Commencing base64Encode\r\n");
  sdFunctionObj.file = fs.open(srcDir, "rb");
  sdFunctionObj.fileEncoded = fs.open(destDir, FILE_WRITE);

  if(!sdFunctionObj.fileEncoded){
      //Serial.println("Failed to open file for writing");
      return;
  }

  uint8_t buf[BASE_64_CHUNK];
  size_t outputLength;
  unsigned char * encoded;

  while (sdFunctionObj.file.available()) {
    // Write BASE_64_CHUNK bytes into buf
    for (uint8_t i = 0; i < BASE_64_CHUNK; i++) {
      if (sdFunctionObj.file.available()) {
        buf[i] = sdFunctionObj.file.read();
      }
    }

    // Encode buf into encoded
    encoded = base64_encode((const unsigned char *)buf, BASE_64_CHUNK, &outputLength);

    for (uint8_t i = 0; i < outputLength; i++) {
      sdFunctionObj.fileEncoded.write(encoded[i]);
      //Serial.print(encodedBuf.charAt(i));
    }

    free(encoded);
  }

  sdFunctionObj.fileEncoded.close();
  sdFunctionObj.file.close();
}
```

LISTING 14: sdCardFunctions.cpp

Appendix G: aesbase64decode.py

See Section 3.1.2.7 for the full code.

Appendix H: index.html

See Section 3.1.2.8 for the full code.

Appendix I: cameraFunctions.h

```
#ifndef __CAMERA_FUNCTIONS_H__
#define __CAMERA_FUNCTIONS_H__

#define PWDN_GPIO_NUM     32
#define RESET_GPIO_NUM    -1
#define XCLK_GPIO_NUM      0
#define SIOD_GPIO_NUM     26
#define SIOC_GPIO_NUM     27
#define Y9_GPIO_NUM       35
#define Y8_GPIO_NUM       34
#define Y7_GPIO_NUM       39
#define Y6_GPIO_NUM       36
#define Y5_GPIO_NUM       21
#define Y4_GPIO_NUM       19
#define Y3_GPIO_NUM       18
#define Y2_GPIO_NUM        5
#define VSYNC_GPIO_NUM    25
#define HREF_GPIO_NUM     23
#define PCLK_GPIO_NUM     22

extern camera_config_t config;

void configInitCamera();
void takeSavePhoto(String path, fs::FS &fs);

#endif
```

LISTING 15: cameraFunctions.h

Appendix J: cameraFunctions.cpp

```
#include "esp_camera.h"
#include "Arduino.h"
#include "FS.h"                // SD Card ESP32
#include "SD_MMC.h"            // SD Card ESP32
#include "soc/soc.h"           // Disable brownout problems
#include "soc/rtc_cntl_reg.h"  // Disable brownout problems
#include "driver/rtc_io.h"
#include <EEPROM.h>            // read and write from flash memory
#include "cellularFunctions.h"
#include "sdCardFunctions.h"
#include "cameraFunctions.h"

camera_config_t config;

```

```
void configInitCamera(){
  config.ledc_channel = LEDC_CHANNEL_0;
  config.ledc_timer = LEDC_TIMER_0;
  config.pin_d0 = Y2_GPIO_NUM;
  config.pin_d1 = Y3_GPIO_NUM;
  config.pin_d2 = Y4_GPIO_NUM;
  config.pin_d3 = Y5_GPIO_NUM;
  config.pin_d4 = Y6_GPIO_NUM;
  config.pin_d5 = Y7_GPIO_NUM;
  config.pin_d6 = Y8_GPIO_NUM;
  config.pin_d7 = Y9_GPIO_NUM;
  config.pin_xclk = XCLK_GPIO_NUM;
  config.pin_pclk = PCLK_GPIO_NUM;
  config.pin_vsync = VSYNC_GPIO_NUM;
  config.pin_href = HREF_GPIO_NUM;
  config.pin_sscb_sda = SIOD_GPIO_NUM;
  config.pin_sscb_scl = SIOC_GPIO_NUM;
  config.pin_pwdn = PWDN_GPIO_NUM;
  config.pin_reset = RESET_GPIO_NUM;
  config.xclk_freq_hz = 20000000;
  config.pixel_format = PIXFORMAT_JPEG; //YUV422,GRAYSCALE,RGB565,JPEG

  config.frame_size = FRAMESIZE_VGA; // FRAMESIZE_ + QVGA|CIF|VGA|SVGA|XGA|SXGA|UXGA
  config.jpeg_quality = 10; //10-63 lower number means higher quality
  config.fb_count = 1;

  // Initialize the Camera
  esp_err_t err = esp_camera_init(&config);
  if (err != ESP_OK) {
    //Serial.printf("Camera init failed with error 0x%x", err);
    return;
  }

  sensor_t * s = esp_camera_sensor_get();
  s->set_brightness(s, 0);     // -2 to 2
  s->set_contrast(s, 0);       // -2 to 2
  s->set_saturation(s, 0);     // -2 to 2
  s->set_special_effect(s, 0); // 0 to 6 (0 - No Effect, 1 - Negative, 2 - Grayscale, 3 -
    Red Tint, 4 - Green Tint, 5 - Blue Tint, 6 - Sepia)
  s->set_whitebal(s, 1);       // 0 = disable , 1 = enable
  s->set_awb_gain(s, 1);       // 0 = disable , 1 = enable
  s->set_wb_mode(s, 0);        // 0 to 4 - if awb_gain enabled (0 - Auto, 1 - Sunny, 2 -
    Cloudy, 3 - Office, 4 - Home)
  s->set_exposure_ctrl(s, 1);  // 0 = disable , 1 = enable
  s->set_aec2(s, 0);           // 0 = disable , 1 = enable
  s->set_ae_level(s, 0);       // -2 to 2
  s->set_aec_value(s, 300);    // 0 to 1200
  s->set_gain_ctrl(s, 1);      // 0 = disable , 1 = enable
  s->set_agc_gain(s, 0);       // 0 to 30
  s->set_gainceiling(s, (gainceiling_t)0);  // 0 to 6
  s->set_bpc(s, 0);            // 0 = disable , 1 = enable
  s->set_wpc(s, 1);            // 0 = disable , 1 = enable
  s->set_raw_gma(s, 1);        // 0 = disable , 1 = enable
  s->set_lenc(s, 1);           // 0 = disable , 1 = enable
  s->set_hmirror(s, 0);        // 0 = disable , 1 = enable
  s->set_vflip(s, 0);          // 0 = disable , 1 = enable
  s->set_dcw(s, 1);            // 0 = disable , 1 = enable
  s->set_colorbar(s, 0);       // 0 = disable , 1 = enable
}

void takeSavePhoto(String path, fs::FS &fs){
  // Take Picture with Camera
  camera_fb_t  * fb = esp_camera_fb_get();

  if(!fb) {
    Serial.println("Camera capture failed");
    return;
  }

  // Save picture to microSD card
  File file = fs.open(path.c_str(), FILE_WRITE);
  if(!file){
  }
  else {
```

```
    file.write(fb->buf, fb->len); // payload (image), payload length
  }
  file.close();

  esp_camera_fb_return(fb);
}
```

LISTING 16: cameraFunctions.cpp

Appendix K: Main setup and loop functions

```
#include "esp_camera.h"
#include "Arduino.h"
#include "FS.h"                // SD Card ESP32
#include "SD_MMC.h"            // SD Card ESP32
#include "soc/soc.h"           // Disable brownout problems
#include "soc/rtc_cntl_reg.h"  // Disable brownout problems
#include "driver/rtc_io.h"
#include <EEPROM.h>            // read and write from flash memory
#include "cellularFunctions.h"
#include "sdCardFunctions.h"
#include "cameraFunctions.h"

void setup() {
  Serial.begin(FAST_CATM1_BAUDRATE);
  sdInit();
  configInitCamera();
  takeSavePhoto("/input.jpg", SD_MMC);
  aesEncode(SD_MMC, "/input.jpg", "/inputjpg_encrypted");
  base64Encode(SD_MMC, "/inputjpg_encrypted", "/inputjpg_encrypted_encoded");

  getFileSizeSegment(SD_MMC, "/inputjpg_encrypted_encoded");
}

void loop() {
  cellularHandler();
}
```

LISTING 17: ESP_mqtt_code.ino

Appendix L: Digitally-toggled power latch simulation

```
$ 1 0.000005 10.20027730826997 50 5 50
t 240 160 240 112 1 -1 4.5270374772027955 -0.4727544134752044 40
r 240 160 240 224 0 500
r 352 112 352 288 0 10000
t 352 288 288 288 0 1 -4.526935805334874 0.0002014277624005743 125
g 176 352 176 384 0
g 288 304 288 384 0
g 352 352 352 384 0
w 176 352 176 336 0
w 288 272 240 272 0
w 176 288 176 272 0
w 176 272 240 272 0
w 240 272 240 224 0
w 256 112 352 112 0
w 464 112 352 112 0
w 128 112 224 112 0
R 96 112 80 112 0 0 40 5 0 0 0.5
w 128 112 96 112 0
g 480 144 480 176 0
f 416 320 352 320 32 1.5 0.09
f 112 304 176 304 32 1.5 0.09
w 176 336 176 320 0
w 352 352 352 336 0
w 352 288 352 304 0
s 416 320 464 320 0 1 true
s 112 304 112 272 0 1 true
R 112 224 112 192 0 0 40 3.3 0 0 0.5
R 464 320 464 256 0 0 40 3.3 0 0 0.5
w 112 272 112 224 0
368 480 80 400 80 0 0
368 416 320 416 352 0 0
368 112 304 112 336 0 0
174 480 144 480 96 1 50 0.48020000000000007 Resistance
207 480 80 512 80 4 Load
o 30 64 0 4099 0.0000762939453125 0.00009765625 0 2 30 3 On\sSwitch
o 29 64 0 4099 0.0000762939453125 0.00009765625 1 2 29 3 Off\sSwitch
o 28 64 0 4115 5 0.00078125 2 2 28 3 Load\sVoltage
```

LISTING 18: Digitally-toggled power latch simulation

Please visit https://github.com/JBlakd/latch-simulation-falstad/blob/main/workinglatch.txt for a more copy/paste friendly source of the code.

Please visit https://www.falstad.com/circuit/ and select `File`, followed by `Import from Text` and paste in the code.

Appendix M: Video demonstrations

Please visit https://www.youtube.com/watch?v=-5lCeJ1IYms for a demonstration of the system at an earlier stage of development but gives a more in-depth look at the transmission process from the LTE modem's perspective and the download, decryption and decoding process from the recipient's perspective.

Please visit https://www.youtube.com/watch?v=SwMuQBwRPd8 for a complete demonstration of the finished system.

Appendix N: Github repositories

Please visit https://github.com/JBlakd/ESP_mqtt_code for the source code of the project transmission device firmware.

Please visit https://github.com/JBlakd/mqtt-file-download-decode-decrypt for the source code of the Python downloader and decoder/decrypter scripts.