Master's Thesis

Firmware update in a resource constrained environment

Anton Martinsen
Alexander Nässlander

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, September, 2015.

Firmware update in a resource constrained environment

at u-blox

Anton Martinsen
Alexander Nässlander

2015

Master's Thesis

"Wireless communications"

Faculty of Engineering LTH
Department of Electrical and Information Technology

Supervisors:
Mats Andersson, u-blox
Mats Cedervall, EIT

Abstract

An embedded system's software is often the backbone for a successful product. To make the system stand out from the other competitors on the market, your system must be adaptive and ready for quick updates. However, one may never truly know where your system is going to be used and how difficult it can be to access and update it.

Here we show a solution that securely updates a system based upon a lower layer Bluetooth connection. We found a way to update a system that has constrained memory resources using RSA keys for verifying the validity of the firmware. Furthermore, we found that our solution did not meet the memory restrictions but there are ways to improve the solution so that these restrictions are met.

The findings of this report can be a used as a reference for further development of wireless firmware updates using digital signatures in resource constrained environments.

Contents

1 | Introduction

U-blox develop and sell wireless modules that use Bluetooth, Wi-Fi, cellular technology and positioning technology to communicate in a broad variety of applications. The modules have an applied firmware running the functionality of the communication interface, i.e the Bluetooth stack etc., and an application utilizing this interface. This means that the firmware can be made quite small. A typical firmware for a Bluetooth and Wi-Fi module from u-blox needs about 1.5 MB of storage. A memory used for storing these kinds of firmwares on the module does not need to have big capacity. Using memories with smaller capacities is beneficial in regard to both production costs and power consumption.

Since these modules are very small they can be embedded in larger systems and be hard to physically reach in order to update the firmware on the module if it is needed. At the moment there is no way to update the module's firmware if the module can not be reached physically which is a problem. Hence implementing a way to update the firmware through wireless communication is desired. By adding wireless updating of the firmware on the module using Bluetooth new problems are introduced, someone could possibly connect to the module and send malicious firmware.

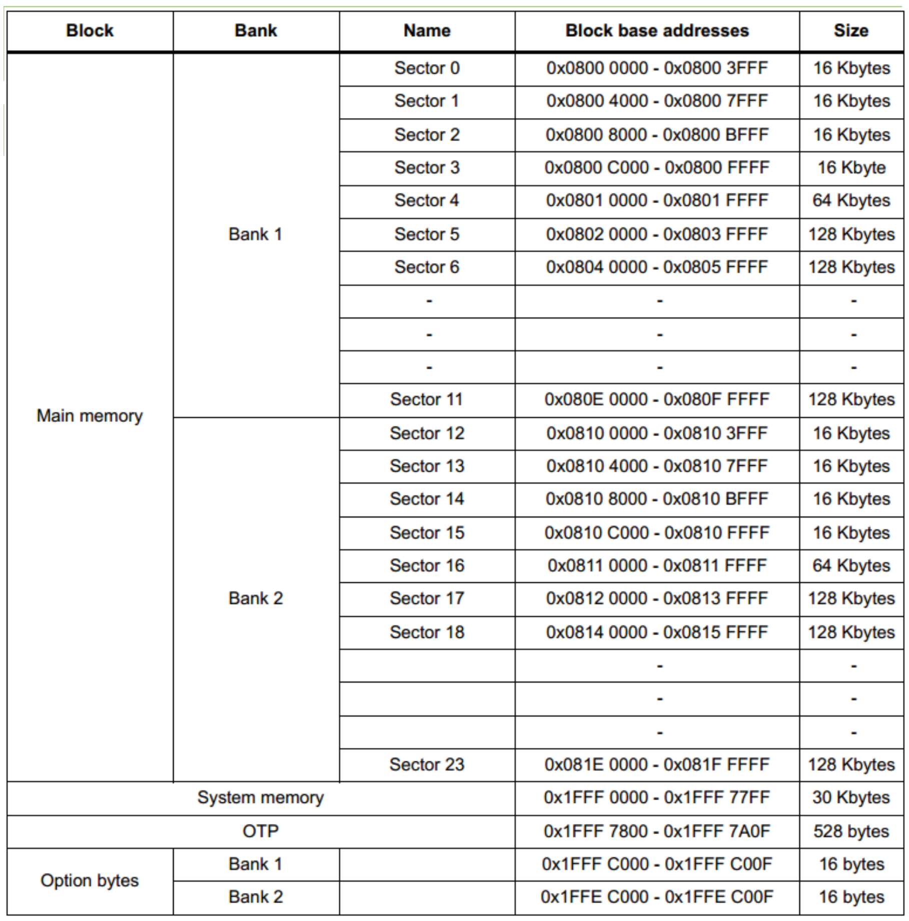
The goal of this project is to create a program on u-blox's wireless module series ODIN-W26x that can upgrade the application firmware over the air. The program needs to be robust and be able to recover from failed updates since the modules can be hard to reach physically. The module also needs to be protected from downloading and executing unintended firmware for the module. Because of memory restrictions the program needs to preferably be less than 128 kB. U-blox want us to research if it is possible to implement a wireless way to update firmware on their module including data security within the size restriction of 128 kB.

2 Background

This chapter will give a brief explanation of different concepts that might be unknown to the reader.

2.1. Sectors in the memory

In the report we will bring up sectors. A sector is a group of memory cells that has a dedicated address range and a size. The sectors supports reading and writing data. The flash memory used in this project (see figure 2.1) can store up to 2 MB of data.



Block	Bank	Name	Block base addresses	Size
Main memory	Bank 1	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
		Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
		Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
		Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
		Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
		Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
		Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
		-	-	-
		-	-	-
		-	-	-
		Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
	Bank 2	Sector 12	0x0810 0000 - 0x0810 3FFF	16 Kbytes
		Sector 13	0x0810 4000 - 0x0810 7FFF	16 Kbytes
		Sector 14	0x0810 8000 - 0x0810 BFFF	16 Kbytes
		Sector 15	0x0810 C000 - 0x0810 FFFF	16 Kbytes
		Sector 16	0x0811 0000 - 0x0811 FFFF	64 Kbytes
		Sector 17	0x0812 0000 - 0x0813 FFFF	128 Kbytes
		Sector 18	0x0814 0000 - 0x0815 FFFF	128 Kbytes
			-	-
			-	-
			-	-
		Sector 23	0x081E 0000 - 0x081F FFFF	128 Kbytes
System memory			0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP			0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes	Bank 1		0x1FFF C000 - 0x1FFF C00F	16 bytes
	Bank 2		0x1FFE C000 - 0x1FFE C00F	16 bytes

Figure 2.1.: Embedded Flash memory interface.

2.2. The Bluetooth stack

The Bluetooth stack (see figure 2.2) is the counterpart to the OSI model for Bluetooth. It varies from implementation to implementation since a lot of the higher layers of the Bluetooth stack are optional. The reason for this is that different manufactures want different functionality for their Bluetooth devices. Let us say that we have a headset that could be connected to your smartphone. It would be necessary for the headset to handle audio streams for playing the sound to the speakers and for sending the recorded audio from the microphone. But it would be unnecessary for the headset to have support for BPP *(Basic Printing Profile)*, which is used for sending items to printing devices.

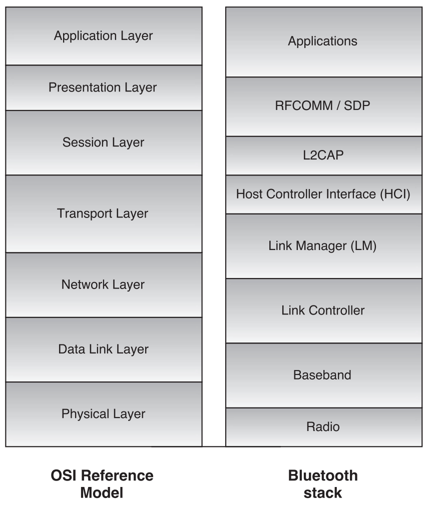


Figure 2.2.: OSI reference model and Bluetooth stack.

2.2.1. Host Controller Interface, HCI

The lower layers of the Bluetooth stack are basically the same for any Bluetooth device. This part of the stack is needed in order to establish a connection with another Bluetooth device. Since different applications in the higher part of the Bluetooth stack can communicate at the same time, HCI and Logical Link Control and Adaptation Protocol (L2CAP) handles all the different application connections that sends or receives data.

HCI sends HCI commands to the Link Manager to make the Bluetooth device behave desirable. For example; sending data, making the device visible for other devices and so on. There is a quite big set of HCI commands and we will just bring up a few of them. The Link Manager may reply if something is happening; received data, a connection request and so forth. The messages from the Link Manager is called HCI events.

To send and receive data, HCI uses ACL (Asynchronous connection-less) data packets, as can be seen in figure 2.3. The packet has a *Connection Handle*, that is used for keeping track of what connection the packet is associated with. There is a field containing information about how much data there is in a packet, called *Data Total Length*. The flags are not used in this project.

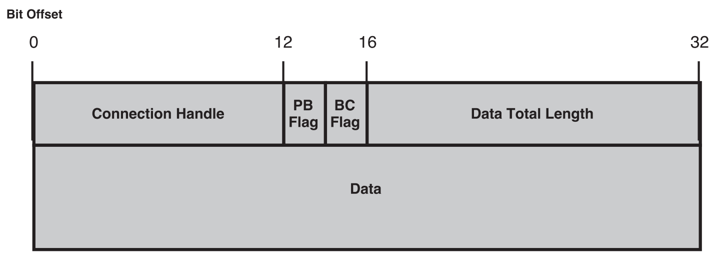


Figure 2.3.: Image of the ACL data packet.

2.3. Callbacks

A callback is a function that can be used as an input parameter to other functions. These callbacks are used to execute the right HCI event handlers when a given HCI event has been detected.

2.4. RSA

RSA is used for secure data transmission and is the most widely used public-key cryptosystem according to *HANDBOOK of APPLIED CRYPTOGRAPHY*[1], chapter *8*, section *2*. It is based upon the practical difficulty of factoring the product of two large prime numbers, known as the factoring problem. The larger primes you use, the harder it will be to crack. An RSA key pair is made of a private key and a public key.

2.4.1. Encryption and Decryption

The public key can be known by anyone and is used for encryption of the data. The data can be decrypted by the owner of the private key. If someone wants to send secret private message to you, the sender encrypts the message and sends it to you. The message can be decrypted with the private key, that is only known by you. It is very important that you keep your private key secret.

2.4.2. Signing and Verifying

You can also use RSA keys to verify the transmitter of the message, called digital signature. If you want to send a message to someone and you want the receiver to verify that the message was from you, your private key can be used. You sign the message with your private key and the receiver uses your public key to see the message. Since everyone has access to your public key they can see the message that you signed. However, the message could only be signed by your private key. In that way everyone knows that you sent the message.

2.4.3. In combination

If you combine a receivers public key and your private key, the message can be encrypted and signed. The receiver can only see the message with the receiver's private key, and verify that the message was sent from you with your public key.

In order to encrypt, decrypt, sign and verify with RSA keys, some mathematical operations must be supported for big integers. Namely modulo operations and exponential functions.

3 Methodology

Here we explain the methods and techniques used to get the module to support wireless firmware updates and the choices we made during the development of our project. The results and limitations of our work is presented in chapter 4, page 39.

3.1. HCI commands in HCITester

Before starting to program the HCI communication in `C` some experience with the HCI layer had to be gained in order to understand which HCI commands and parameters were needed to perform the task of updating the firmware on the module. In order to perform the task there were four main functionalities that had to be implemented and used in the HCI layer:

1. Connect to the module
2. Receive data
3. Send data
4. Disconnect from the module

To decide what commands were needed to complete these four sub tasks, research was made by reading the *Bluetooth 1.1 Connect Without Cables* [2], chapter 8, where the chain of commands to set up a connection is described. The script program HCITester from *Texas Instrument* [5] was used to write four different scripts called ”Master”, ”Slave”, ”Extended Inquiry Master” and ”Extended Inquiry Slave”. The first two scripts were made from a sample script that could set up three different Bluetooth Low Energy connections, send data to them and disconnect. These scrips were carried out by two instances of HCITester connected to two different USB ports with two different USB dongles connected.

3.1.1. Master Script

The Master script sends a connection request to the specified Bluetooth address then waits for a connection complete event response from the Link Manager. When the Master has received a connection complete event it sends an ACL data packet to the Slave and awaits an ACL data packet back. When the Master receives an ACL data packet it disconnects and waits for a disconnection event.

3.1.2. Slave Script

The Slave script waits for a connection to be made then it waits for an ACL data packet and responds by sending an identical ACL data packet back to the Master. After this it waits for a disconnection event.

3.1.3. Extended Inquiry Master Script

Sets up an extended Inquiry scan of the surrounding. Also includes a remote name request to identify the name of the module with a specified Bluetooth address. This one was used a lot throughout the project to identify the Bluetooth address of the module but mostly to identify if the module was up and running and check if the firmware had been switched by setting the radios extended inquiry response message to different text messages.

3.1.4. Extended Inquiry Slave Script

Sets up the radios local name and its extended inquiry response. Then makes the module visible by enabling scan, setting scan type and setting up the scan activity. The commands in this script should be used by the module to make it visible.

3.2. C Implemention

U-blox already had a project where they could tunnel HCI commands and events via a UART interface. The project included the complete Bluetooth stack and the RTSL (*Real Time System Library*) environment. This means that it was possible to connect and send data to a target device using the Bluetooth stack. Since the implementation for the communication with the Link Manager was already done in that project, it was a good basis for our server implementation.

To make our solution smaller we had to remove some unwanted functionality from the u-blox project, i.e the higher layers of the Bluetooth stack and the UART interface. We decided to remove the L2CAP part of the Bluetooth stack and the higher-layered protocols. We made that decision based on the knowledge that the module only should have one connection with another device, namely the client that sends the new Firmware to the module. Another benefit of doing this is that we got less higher-layered headers to take care of in the frames. This gave us more space for the firmware to be sent in our sending frames. We removed the UART interface because we thought it would be unnecessary for the module to be able to communicate with the entity that the module is physically connected to, during the update of the firmware. When we had removed the UART interface and the higher-layered parts of the Bluetooth stack, the project could still be compiled and was executable. However, the module would not know what to do with the incoming packets from the Link Manager since the pipe was now broken. If we executed the modified project it would generate errors where the pipe was previously used. We used this to our advantage by debugging the modified project and observing the call stack to see where the errors were produced.

3.2.1. Setting up a connection on the module using HCITester

3.2.1.1. Making the module visible

We started off by trying to find the HCI commands in the HCI layer of the `C`-code project that corresponded to the same commands as the ones in the HCITester script called ”Extended Inquiry Slave” that made the dongle visible and able to respond to an extended inquiry request. The following five functions were found:

- `HCI_CmdWrScanEnable(0x03)`
- `HCI_CmdWrName(remoteName)`
- `HCI_CmdWrExtInqRsp(FALSE, text)`
- `HCI_CmdWrPScanActivity(uiPageScanInterval,uiPageScanWindow)`
- `HCI_CmdWrInqScanType(0x01)`

These functions seemed to match the commands that were used in the ”Extended Inquiry Slave” script. We tried executing these functions, however the device did not show up when we tried to find it using our ”Extended Inquiry Master” script in HCITester. We decided to insert breakpoints in

Visual Studio in the two functions called `cbOS_error` and `cbOS_error2`. These functions catch any errors produced in the OS used by the module. One of the breakpoints managed to catch an error and we decided to follow the callstack and noticed that a callback was missing, we had to register the appropriate callback in order for the program to have the necessary functions to perform the task. A more detailed description of the procedure of identifying the callback can be seen in 3.2.1.2.

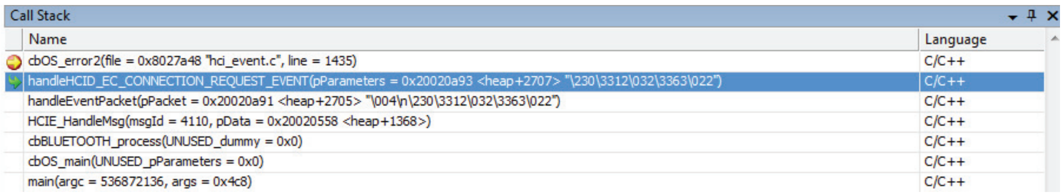
The callback was of the type `cbHCI_CallBack` and the function to register it was `cbHCI_registerCallBack(cbHCI_CallBack* hciCallback)`. Using this function to register the callback in our initiation we managed to execute the code without errors.

The module was now showing up when an extended inquiry request was performed in HCITester.

3.2.1.2. Connecting to the module

The module was now visible and able to respond to an extended inquiry request, meaning we could now identify the Bluetooth address of the module in HCITester from the extended inquiry response. By inserting the modules Bluetooth address into the Master script in HCITester we could now force a connection event on the module, once again we researched the call stack in Visual Studio to find the callback needed.

In figure 3.1 on page 13 you can see the call stack and the function that created the error was called `handleHCID_EC_CONNECTION_REQUEST_EVENT`.


Call Stack

Name	Language
cbOS_error2(file = 0x8027a48 "hci_event.c", line = 1435)	C/C++
handleHCID_EC_CONNECTION_REQUEST_EVENT(pParameters = 0x20020a93 <heap+2707> "\230\3312\032\3363\022")	C/C++
handleEventPacket(pPacket = 0x20020a91 <heap+2705> "\004\n\230\3312\032\3363\022")	C/C++
HCIE_HandleMsg(msgId = 4110, pData = 0x20020558 <heap+1368>)	C/C++
cbBLUETOOTH_process(UNUSED_dummy = 0x0)	C/C++
cbOS_main(UNUSED_pParameters = 0x0)	C/C++
main(argc = 536872136, args = 0x4c8)	C/C++

Figure 3.1.: Picture of the call stack from a forced connection event on the module.

By looking closer at this function, in listing 3.1. We could see that the error was asserted at line 6. This meant that the callback called `pCmCallback` was equal to `NULL`, which means that the callback had not been registered.

```
 1  static void handleHCID_EC_CONNECTION_REQUEST_EVENT(uint8 *pParameters)
 2  {
 3    TBdAddr       BdAddress;
 4    uint8         linkType;
 5
 6    cb_ASSERT(pCmCallBack != NULL);
 7
 8    if(pCmCallBack != NULL &&
 9       pCmCallBack->pfConnectInd != NULL)
10    {
11        HCI_ReverseBdCopy(BdAddress.BdAddress,pParameters);
12        BdAddress.AddrType = BT_PUBLIC_ADDRESS;
13        linkType = pParameters[9];
14
15        pCmCallBack->pfConnectInd(BdAddress,linkType);
16    }
17  }
```

Listing 3.1: Error asserted at line 6 if `pCmCallBack` is `NULL`.

Then we searched the Bluetooth stack of the project for `pCmCallBack` and found that `pCmCallBack` was of the type `HCIE_TCmCallBack`. However we could not simply just register this callback because the original functions in the callback sent information to higher layers in the Bluetooth stack. In order to break the communication with higher layers we had to statically define the callback in our handler and override the functions. The statically defined connection callback can be seen in listing 3.2 on page 15. The handle functions that has been set as `NULL` will be disregarded by our handler, this was done because these functions were not necessary for the solution of the project. Also we noticed that this callback included the function handling disconnection events, naturally we decided to include the function handling disconnection events and by overriding it we could make sure it did not pass information to higher layers in the stack.

```
static HCIE_TCmCallBack hciConnectionCallback=
{
        HandleHciConnectEvt,
        HandleHciConnectCnfNeg,
        HandleHciConnectInd,
        HandleHciDisconnectEvt,

        NULL, //HandleHciDisconnectCnfNeg,
        NULL, //HandleHciConnectCancelCnf,
        NULL, //HandleHciAuthEvt,
        NULL, //HandleHciChangeEncryptEvt,
        NULL, //HandleHciRequestKeyInd,
        NULL, //HandleHciRequestPinInd,
        NULL, //HandleHciNewKeyEvt,
        NULL, //HandleHciIoCapabilityInd,
        NULL, //HandleHciIoCapabilityEvt,
        NULL, //HandleHciUserPasskeyInd,
        NULL, //HandleHciUserPasskeyEvt,
        NULL, //HandleHciUserConfirmationInd,
        NULL, //HandleHciSimplePairingComplEvt,
        NULL, //HandleHciRemoteExtFeatureCnf,
        NULL, //HandleHciRemoteExtFeatureCnfNeg,
        NULL, //HandleHciReadEncryptionKeySizeCnf,
};
```

Listing 3.2: The static definition of the callback handling connections.

An example of one of the overridden functions can be seen in listing 3.3 on 15. The function simply responds that it accepts a connection if it has no current connection, if it however already has a connection then it will decline the connection attempt. As you can see, the function only uses HCI commands to respond and does not include any higher layers.

```
static void HandleHciConnectInd(TBdAddr tAddress, TPacketType ↩
    tPacketType){

    if(connHandle == INVALID_CONN_HANDLE){
                HCI_RspConnect((TBdAddr*)&tAddress,↩
                    BT_MASTER_SLAVE_POLICY_OTHER_SIDE_DECIDE);
        }
        else
        {
                HCI_RspNegConnect((TBdAddr*)&tAddress, ↩
                    HCI_ERR_REMOTE_USER_TERMINATED_CONNECTION);
        }
}
```

Listing 3.3: The overridden `HandleHciConnectInd` function.

After that the registration of the callback was done with one simple line of code: `HCIE_RegisterConnectionCallBack(&hciConnectionCallback)`

The connection event function became rather simplistic, only saving the connectors Bluetooth address in a variable.

3.2.1.3. Receiving data on the module

Once the code for connection had been implemented we could verify it working by noticing in HCITester that the Master script had got a connection event from the module and had now begun to try and send an ACL packet instead of waiting for a connection event. This generated another `cbOS_error` and we simply followed the call stack once again to find the callback needed for the task of handling data. By registering the appropriate callback for handling data and overriding the `handleDataEvent` function we could verify that the information we had sent from HCITester was available as a parameter in the function `handleDataEvent`.

3.2.1.4. Sending data from the module

The Master script was now waiting for a response from the module in the form of an ACL data packet. In order to make the script move on to the disconnection part of the script we needed to send an ACL data packet from the module in response to the received data from HCITester.

By searching the different HCI functions available we managed to find the function "`HCI_WritePacket`" which needed the saved connection handle and the length of the data, including the ACL header size, to be sent. As one can guess it also needed the actual data to be sent. Before calling the `HCI_WritePacket` function an ACL data packet must be assembled by the above parameters. The function takes two arguments, the ACL data packet to be sent and a status telling that it is a ACL data packet. An example of the usage of the `HCI_WritePacket` function can be seen in listing 3.4, row 15, page 17.

However this function did not behave as we expected, the ACL data packet was sent to the Link Manager but was never transmitted from the radio.

We decided to put a breakpoint at the function and follow the operations it made. When doing so we noticed that the `HCI_WritePacket` function, which took a pointer to the ACL data packet to be sent as input parameters, made the pointer go back one byte in the memory to verify that the byte before the pointer was a constant called `HCI_TRANSPORT_INDICATOR_ACL_DATA_PACKET`, which was represented by `0x02`. Basically it was double checking that the ACL data packet was indeed an ACL data packet.

We made a function called `sendData` which took the data to be sent and the length of the data to be sent as an input parameters. Then it created a pointer to a byte which had the value `0x02` (`HCI_TRANSPORT_INDICATOR_ACL_DATA_PACKET`) followed by a ACL data packet assembled with the connection handle and the length of the data to be sent and the actual data. The pointer was then changed so that it was pointing to the first byte of the ACL data packet. This pointer was then sent into the HCI function for sending ACL data packets. The function can be seen in listing 3.4.

The connection handler and the length are both represented with two bytes, in Little-Endian format. The Link Manager interprets the ACL data packet in Big-Endian, so we had to convert them as can be seen in listing 3.4, row 6 to 9.

```
 1  extern void sendData(uint8 *pucData, uint16 uiLength){
 2          uint8[uiLength + 5] sendingAclPacket;
 3          sendingAclPacket[0] = 0x02;
 4
 5          sendingAclPacket[1] = (uint8)(0xFF & connHandle);
 6          sendingAclPacket[2] = (uint8)(connHandle >> 8);
 7          sendingAclPacket[3] = (uint8)(0x00FF & uiLength);
 8          sendingAclPacket[4] = (uint8)(uiLength >> 8);
 9
10          for(uint16 i = 0; i < uiLength; ++i){
11                  sendingAclPacket[i + 5] = pucData[i];
12          }
13
14          HCI_WritePacket(HCI_ACL_DATA_PACKET, &sendingAclPacket[1]);
15  }
```

Listing 3.4: Code snippet for sending data via ACL data packet.

3.2.1.5. Disconnecting from the module

When the HCITester Master script received an ACL data packet in return from the module it forced a disconnection event on the module. The callback needed to handle this had already been registered and the disconnection handle function had already been overridden at this point.

However we encountered a rather strange error when the event was forced on the module. Somehow when the HCI function wanted to decide if the connection was a classic Bluetooth connection or Bluetooth Low Energy through a function called `getLinkType()`, it decided that the connection was of the type Bluetooth Low Energy, which produced a `cbOS_error`.

This was of course wrong, the connection was not a Bluetooth Low Energy connection. We tried to, instead of using `getLinkType()`, just set the link type to a static `0x01`, which represented a classic Bluetooth connection. This simple attempt was successful. And the disconnection was successful.

The handle disconnection event function became very simplistic as well, just clearing the current connection variable.

3.2.2. Client connection

When the module could handle the HCI events needed for setting up a connection, disconnecting, sending and receiving data, a client solution which could connect to the module and send the data had to be created.

We were provided with a project from an employee at u-blox. That project had some similarities with the original project that we started with. The project had the functionality to connect to a specified Bluetooth address with HCI, i.e establish a data link with the lower part of the Bluetooth stack. This project was using a COM port to communicate with the radio unit via a USB interface.

This mean that the software for this project can run on a computer and send HCI commands to the radio and receive HCI events from the radio as well. Since the software would run on our computer it would be convenient.

We copied the `sendData` function from the server implementation and registered the same callbacks that we needed to send data, receive data and disconnect. We also hard coded the modules Bluetooth address so it would be easier for us to test our solution.

3.2.3. Watchdogs

If the module got stuck somewhere while it was running, we wanted it to recover from that. We found implementations of watchdogs timers that was made by u-blox. If the module had a connection with the client and no data was sent during 1 second the module would restart itself. Every time the module received data we reset the watchdog timer.

3.3. FOTA, Firmware update Over The Air

Now that we had a module and a client that could establish a connection and send and receive data we had to add some additional logic. We started of by creating two `C` files, `FOTA_client.c` and `FOTA_server.c`. Our idea was that the sending client would use `FOTA_client.c` to send data to the module, then the module would then use `FOTA_server.c` to handle the data. We figured that these two files should behave like big multiplexers for the opposites sides. We decided that the multiplexers would be implemented using the foundations of the TCP state machine: *SYN*, *ACK*, *SEQ* and *FIN*.

3.3.1. Communication protocol, FOTA-header

The TCP state machine and TCP header has a lot of good features[1] but a lot of them was unnecessary for our project. Since the module should only have one connection with the sending client via a data link, not through a network[2], we could remove some features. We did not have to deal with *Ports*[3], *Urgent pointer* and *Options*. So we simply did not included those features in our protocol. *Data offset* was no longer needed because of the removal of the *Options* field made our header fixed sized. This was made to save us space so that the module could receive bigger packets, having in mind that the the maximum amount of bytes in one frame is 1017 bytes as described later in subsection 3.3.2 on page 20. The protocol that was created can be seen in figure 3.2.

[1]Data fields in the TCP header
[2]TCP works on the Transport layer of the OSI model
[3]Source port and Destination port

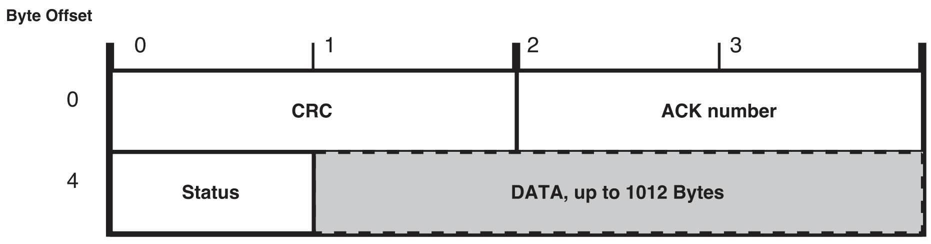


Figure 3.2.: Image of the FOTA header

We decided that we might have use for some error-detection functionality. That is why we dedicated a 16 bit field in the FOTA for CRC (*Cyclic Redundancy Check*). The idea of the ACK number field was that it should be used both as ACK number and SEQ number. The sending client sends the FOTA header and puts the sequence number in that field and the module responds with a FOTA header with an acknowledge number in that field. We realized that the module would never send large amount of data to the sending client, therefore we decided to merge the fields of *Sequence number* and *Acknowledge number* with each other.

The reason we put a Status field, as shown in figure 3.2, is that we wanted to use it as the TCP flags. One byte would give us plenty of different combinations of flag to send ($2^8 = 256$).

3.3.2. Client and Server communication

When we were able to connect, receive data, send data and disconnect to the module and the client, we where now ready for the module and client to establish a data link. We had to take care of the incoming ACL data packets that was dedicated for the module and the client. If either the client or the module got an ACL data packet, it would trigger the `handleDataEvt` function, see listing 3.5, page 21.

In order to make sure that the ACL data packet is dedicated for the module we analysed the Connection handler from the ACL header an compered it with the previously stored Connection handler, see listing 3.3, page 15. If the ACL data packet was not dedicated for the module, the module would simply ignore that packet. If it was the correct Connection handler, a pointer to the data field of the ACL header and the length of the data field (in bytes) would be the parameters for the `FOTA_handleData` function. That function is the entry for the state machine in `FOTA_server.c`, see listing 3.6. The ACL data packets that is sent to the client is processed in the a similar manner.

```c
static void handleDataEvt(TConnHandle tHandle, boolean bFirst, uint8 *↩
    pucData, uint16 uiLength)
{

        if(tHandle == connHandle){

        FOTA_handleData(pucData,uiLength);
    }
}
```

Listing 3.5: Incoming ACL data packets with the right connection handler is forwarded to `fota_client.c` and `fota_server.c`.

```c
extern void FOTA_handleData(uint8 *pucData, uint16 uiLength);
```

Listing 3.6: Entry to the state machine in `fota_server.c`.

We started to define some flags for the status field for the FOTA header[4]. The first flags we defined were: `FIRST_PACKET`, `ACK`, `SEQ` and `END`, see listing 3.7. We realized that we would need more flags for the status field but these flags would make a good basis. We built the state machine in `FOTA_server.c` and `FOTA_client.c` according to those flags.

```c
// Flags for Status in FOTA_header
#define FIRST_PACKET      0x00
#define ACK               0x01
#define SEQ               0x02
#define END               0x03
```

Listing 3.7: First definitions of status flags for FOTA header .

The first thing we tried was to send some random data[5] in the data field of the FOTA header with the `FIRST_PACKET` flag, from the client to the module. The module would reply the same data back to the client who printed out the data on console. We made sure that the received data was identical to the data that was sent to the module. We had some

[4]FOTA header, see subsection 3.3.1, page 19.
[5]An array of data starting with `0x00, 0x01, 0x02..` and so on.

problems in the beginning to send big chunks of data to the module. For some reason the module could only receive ACL data packets with 1017 Bytes or less. This was due to the radio settings. We decided to keep this setting and adjust our packets to that size. This was a fairly easy step but it was an acknowledgement that we had two separate programs that could communicate with each other.

The next thing that was needed was to let the client tell the module how much the total amount of bytes it was going to send. We did that by sending a `FIRST_PACKET` flag with a 32 bit word in the data field of the FOTA header, to the module from the client. The reason that we used a 32 bit word was that a 16 bit word was to small to represent the maximum size of a firmware, $(2^{16} - 1)bytes < 2MB < (2^{32} - 1)bytes$. For simulation purpose, we allocated that amount of bytes (i.e the value of the 32 bit word) in the modules RAM. Now we could simulate writing to the modules EEPROM with the pointer given by the allocation. We started to simulate a firmware update that needed 1 MB of space. Since we could only send 1012 bytes of data for each FOTA packet we started using the `SEQ` and `ACK` flags. We implemented a solution that imitated the TCP ACK's and PSH's excluding the sliding window solution. If the module got a Sequence number that was not expected it would reply with an Acknowledge number of the wanted number, i.e a resend. We introduced a 16 bit variable (`static uint16 ACKED`) to keep track of the Acknowledge number. For the client side of the solution an associated variable (`static uint16 SentUnAcked`) was introduced. It was introduced to keep track of any resend requests from the module.

For the module to keep track of how many bytes of data that has been received, we introduced a 32 bit variable: `static uint32 bytesReceived`. This variable was updated for every FOTA packet that contained data, however it was not updated for resend requests. When the module had received all of the data from the client it was time to terminate the connection. To terminate the connection we implemented the module to send an `END` in the Status field of the FOTA header. The client was implemented to terminate the program when the module had sent an `END` in the Status field of the FOTA header.

When we got the communication to work properly we implemented error detection. The client calculated the CRC of the data that would be sent in a FOTA packet and then put the result in the CRC field of the FOTA header. When the module receives a FOTA data packet it calculates the CRC of the data field and compares it with the value of the CRC field. If the CRC is not valid the module will ask for a resend.

3.3.3. Updating the firmware

When we had established a working communication of sending data to the module, we had to figure out how to write the data correctly into the memory of the module. We found a template called `cb_flash.h` which included two promising functions `cbFLASH_write(cb_uint32 addr, cb_uint8* data, cb_uint32 length)` which starts writing the data at a specified address and `cbFLASH_sectorErase(cb_uint32 addr)` which erases the whole sector at the specified address.

3.3.3.1. Erasing old firmware

As a first step we used the function `cbFLASH_read(cb_uint32 addr, cb_uint32 length, cb_uint8* buf)` that reads the values of the memory into a buffer from the specified address. Then we used `cbFLASH_sectorErase` at the specified address to erase the values at this sector of the memory. When using `cbFLASH_read` after erasing the sector we could verify that the values in the buffer called `buf` had been erased.

Since we had to erase one sector at a time with a given address to each sector, we decided to make a sector memory offset map of the application memory sectors, as seen in listing 3.8 on page 24. Each element is the memory offset of each sector from the application address which the application runs from. By doing this we realized that the maximum size of a firmware could be 1.625 MB.

```
static uint32 memOffsets[19] = {
        0x000000        /* Sector 4,  64  KB */,
        0x010000        /* Sector 5,  128 KB */,
        0x030000        /* Sector 6,  128 KB */,
        0x050000        /* Sector 7,  128 KB */,
        0x070000        /* Sector 8,  128 KB */,
        0x090000        /* Sector 9,  128 KB */,
        0x0B0000        /* Sector 10, 128 KB */,
        0x0D0000        /* Sector 11, 128 KB */,
        0x0F0000        /* Sector 12, 16  KB */,
        0x0F4000        /* Sector 13, 16  KB */,
        0x0F8000        /* Sector 14, 16  KB */,
        0x0FC000        /* Sector 15, 16  KB */,
        0x100000        /* Sector 16. 64  KB */,
        0x110000        /* Sector 17, 128 KB */,
        0x130000        /* Sector 18, 128 KB */,
        0x150000        /* Sector 19, 128 KB */,
        0x170000        /* Sector 20, 128 KB */,
        0x190000        /* Sector 21, 128 KB */,
        0x1A0000        /* Sector 22, 128 KB */
};
```

Listing 3.8: Memory offsets

We then decided to make our own function called `prepareMem(uint32 length)`, which took the length of the new firmware as an input argument and from this length calculated the number of sectors that would have to be erased for the new firmware to fit. When the number of sectors had been calculated the function only had to iterate through the number of sectors that needed to be erased and use the code in listing 3.9 on page 24 where `FLASH_ADDR` is the address where the application starts.

```
res = cbFLASH_sectorErase(memOffsets[i] + FLASH_ADDR);
```

Listing 3.9: Erasing a sector

When running this code the program crashed and could not start again when restarting the module. We quickly realized that the FOTA firmware had started to erase itself on the module because the FOTA firmware was running at the application address at this moment.

In order to solve this we changed the address and offset where Visual Studio puts and executes the FOTA program in the memory of the module.

When the code for FOTA executed from the memory address `0x801B0000`, the FOTA code was out of range for the `prepareMem` function to erase it. We could now verify with the function `cbFLASH_read`, that the sectors had been erased.

However the `prepareMem` function later had to be changed slightly so that it erased all of the application sectors instead of only erasing the sectors needed for the firmware. This was done because when the firmware was updated with real firmware some bytes were left unchanged from the old firmware that somehow interacted with the new firmware. Specifically the text strings for the radio remote name and the extended inquiry response were affected depending on what firmware used to be on the module. By clearing all the sectors for each update this problem was avoided.

3.3.3.2. Writing data to the memory

By writing some random data with the function `cbFLASH_write` into the memory followed by `cbFLASH_read` we could verify that the function `cbFLASH_write` was working properly.

Since the FOTA handler has a variable called `bytesReceived` that keeps track on how many bytes that have already been received of the new firmware the function `cbFLASH_write` simply had to write the incoming data to the address `FLASH_ADDR+bytesReceived`. When this function had successfully written the data to the memory the variable `bytesReceived` was simply incremented with the data length.

When `bytesReceived` was equal to the firmware length we used our self made function `void cbHW_forceFOTABoot(cb_uint32 status)` in the `cb_hw.c` file to restart the module into the bootloader.

3.3.3.3. Generating real firmware in binary code

Now when the module was able to erase old data and write new data into the memory we wanted to try and send some real firmware in binary code. We made an identical copy of our FOTA project but changed the text strings to the HCI commands that changed the remote name and the extended inquiry response on the radio. By doing this we could identify if the firmware had been downloaded by the module by using HCITester to send a remote name request and an extended inquiry request. This project was then compiled and the binary file of the program was altered so it was in the format of a byte array and inserted as temporary firmware in our client project as a byte array.

3.3.3.4. Sending binary program to the module

At first glance it seemed to work perfectly to send the firmware to the module. The remote name and extended inquiry response had changed to the new intended responses. Note that there was no firmware at the application address at this try. The memory was clean.

We wanted to try and change existing firmware at the application address for new firmware. By using Visual Studio to insert another copy of our FOTA code at the application address but with a different remote name and extended inquiry response. Then we started the updating sequence. The module restarted and we once again used HCITester to check the name request response and extended inquiry response. The module was able to respond to the two different HCI requests, however the responses looked rather strange. They looked like they were a mixture of the old firmwares response and the new firmwares response.

We decided to check the memory with the program STM32 ST-Link utility using an ST-link and export the binary file of the program which was now in the modules memory and compare this binary file with binary file of the program which had been sent from the client. The two binary files were identical except for the part which decided the remote name of the module and the extended inquiry response text field.

The two responses seemed to depend on the previous firmwares response so we tried to change the `prepareMem` function so that it cleared all the application sectors instead of just the sectors needed for the size of the firmware to make sure there was no code left from the previous firmware. This proved successful and the firmware update functionality was complete.

3.3.4. Initiating FOTA

We implemented a simple application so that we could simulate a firmware already running on the module. The application could make the module accessible and connected with another device on a HCI level. We wanted to provoke the application to start FOTA. As mentioned in section 3.3.3.2 we implemented a function (`void cbHW_forceFOTABoot(cb_uint32 status)`) in `cb_hw.c` that restarts the module and writes a value to a backup register. This register can be accessed by the module at any time. The value of the argument is stored in the backup register.

3.4. The Bootloader

At first we used u-blox own bootloader without any alterations however when we wanted to start FOTA by connecting to the module. The bootloader had to be altered a little bit. We had to implement a check in the bootloader which checked a certain byte in the backup register to see if this was equal to `0x01` which meant that FOTA had been activated. If this byte was equal to `0x01` then the bootloader would execute the program and run from the FOTA address instead of the application address. If the byte is something other than `0x01` the bootloader will try to run the application. If the module lost its power then the information in the backup register would be lost.

When the first part of security was implemented, a simple naive hash of the program, the bootloader had to be altered a little again. The bootloader had to read the stored hash and the stored length of the application from the EEPROM and calculate the hash of the program from the application address to check if the program was correct before executing the application at the application address.

When the RSA signing had been implemented the bootloader was changed so that it calculated the hash of the firmware, then evaluated this hash with signed hash stored in the EEPROM using the stored 512 byte private key. See chapter 3.5.7.2 at page 33 for example of this RSA verification.

The bootloader refuses to start the application if the first bytes at the application address have the value `0xFFFFFFFF`, which means that there is no application to be run.

3.5. Data security

When we started this project we knew that data security would have a big impact on the outcome. We decided quite early to follow three basic principles: *Integrity*, *Availability* and *Authenticity*. We decided to use RSA keys for our project since it is well known, well tested and well documented.

3.5.1. Master key

Our idea was that u-blox should possess a big private RSA key for every product type. Every unit associated with that particular product type has the public RSA key of that product type. We call this key: "*Master key*". Since the security of RSA is based on factorization of prime numbers it will be harder to break with larger prime numbers. This means that we want a big Master key. According to BlueKrypt [4] both NIST and ANSSI recommend a key of length 2048 bit, hence using an RSA key of 512 bytes (4096 bit) should be considered safe. We thought it was a good idea to let the module authenticate every FOTA data packet that was received. So we decided to sign each packet of data that is sent to the module with an RSA signature.

3.5.2. Representing big numbers in `C`

In order to support a 512 byte RSA key in `C` we needed to represent bigger numbers (and their mathematical operations) than what standard `C` could. At first we tried implementing this ourselves using vectors of 32 bit numbers. However, this proved rather troublesome. We researched if there were any big number libraries available for `C` and we noticed that u-blox already had licensed code for RSA that had implemented a way to use big numbers and their operations which was used in their WLAN project. We decided to include the files `rsa.c`, which has everything needed for RSA encryption and signing, and `bignum.c` which has big number representation with all the mathematical operations that are needed for RSA.

3.5.3. Session key

Using a 512 bytes signature from the Master key in every FOTA data packet would limit the maximum of data in each frame to 500 bytes:

$$1012_{DataFieldSize} bytes - 512_{SizeOfSignature} bytes = 500 bytes$$

We did not want to use a smaller Master key and we wanted to be able to send bigger amounts of data in every FOTA data packet, to speed up the transmission. To fix this problem we decided to introduce a temporary RSA key that the module and the client only used during the session of the firmware update. We call this RSA key: "*Session key*".

The Session key should only exist a short period of time and does not need to be as large as the Master key. We decided to use a 128 byte (1024 bit) RSA key as the Session key. By using a 128 byte RSA key we increased the maximum limit of bytes of data in a FOTA data packet from 500 bytes to 884 bytes:

$$1012_{DataFieldSize} bytes - 128_{SizeOfSignature} bytes = 884 bytes$$

This introduced a problem for us, how can the module verify this temporary Session key if there is a new Session key for every session? The module already has the public Master key stored, so we decided to sign the public Session key with the private Master key for every new firmware update and send it to the module.

3.5.4. Generating the RSA keys

By using the two functions `rsa_init(rsa_context *ctx, int padding, int hash_id)` and `rsa_gen_key(rsa_context *ctx, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng, unsigned int nbits, int exponent)` in `rsa.c` we were able to generate a private and a public Master key on 512 bytes with the code in listing 3.10 on page 30. Where the key information is saved in the `rsa_context` variable `root_keys`. The fourth input parameter is the length of the key in bits, 512 bytes is 4096 bits so this was inserted as the fourth input parameter to the function.

```
rsa_context root_keys;

rsa_init( &root_keys, RSA_PKCS_V15, 0);

if(rsa_gen_key( &root_keys, &myrand, NULL, 4096, 0x10001) != 0){
    printf("FAILED GENERATING KEY\n");
}
```

Listing 3.10: Code that generates the Master keys into the variable root_keys.

The private key information was then written to a file to be stored with the function `mpi_write_file( const char *p, const mpi *X, int radix, FILE *fout )` in `bignum.c`, this function takes the big number structs called mpi, which are stored in the `rsa_context` struct as input argument and writes them to a file given the file stream and the radix format you want the keys to be stored in. We decided to save them as hexadecimal by writing 16 as the input. An example of writing the big numbers to a file is given in listing 3.11.

```
        fileP = fopen("P.bin", "wb");

        i = mpi_write_file( NULL, &root_keys.P, 16, fileP );

        if ( i != 0){
                printf("ERROR: write to file: %d\n", i);
        }

        fclose(fileP);
```

Listing 3.11: An example of writing big numbers to file.

The public Master key was inserted into the EEPROM on the module and was accessed both in the FOTA authentication process and in the bootloader to check if the firmware is valid u-blox firmware at FOTA start up and module start up respectively.

By using the same two functions but with different input parameters we could also generate a public and private Session key on 128 bytes. Only two things had to be changed, the `rsa_context` variable `root_keys` was changed to `session_keys` and the parameter deciding the length of the key was changed from 4096 bits to 1024 bits since the Session keys should be 128 bytes long. These Session keys are generated every time an authentication of the firmware with the Master key has been successful.

3.5.5. Message digest function

In order to sign a message we needed a message digest function, a so called hash function that can be given an RSA signature. The message digest function is used both when signing and when verifying. We decided to use SHA-1 for this. The output of SHA-1 will always be 20 bytes regardless the length of the message you want to digest. We found a free to use `C`-file from google code [3], which was written by Wei Dai, that calculates the SHA-1 hash which we called `sha1.c`. The function calls can be seen used in listing 3.13 at page 33.

3.5.6. Signing

Since we decided to use RSA keys to verify data etc, we had to add signing features. The `rsa.c` had a wide selection of RSA signing functions using different hash functions. Luckily for us, they had support for SHA-1 signing. We used an already existing function to sign our hash sums. The function that we used was `rsa_rsassa_pkcs1_v15_sign( rsa_context *ctx, int mode, int hash_id, unsigned int hashlen, const unsigned char *hash, unsigned char *sig )`.

It uses the private RSA key that is contained in `rsa_context *ctx` to sign the hash (`const unsigned char *hash`). In order for the function to know what kind of hash to sign, SHA-1 in our case, `int hash_id` has to be set to a value that corresponds to SHA-1. This was predefined in `rsa.c` as: ”`#define SIG_RSA_SHA1 5`”. The function needs the length of the hash (`unsigned int hashlen`) if one does not specify what kind of hash is being used. A pointer where the result will be placed (`unsigned char *sig`) is required. The parameter `int mode` specifies if the function should sign with the public RSA key or the private RSA key of the `rsa_context *ctx`. It might seem obvious but we used the private RSA key for every signing. If the signing was successfully done the function would return `0`. An example of using the function can be seen in listing 3.12 on page 32.

```c
if( rsa_rsassa_pkcs1_v15_sign( &session_key, RSA_PRIVATE, SIG_RSA_SHA1,↩
     0, calculatedHASH, &signedHASH) != 0)
{
    printf( "Failed signing\n" );
}else{
    printf( "Signing success\n" );
    }
}
```

Listing 3.12: Example of RSA signing.

3.5.7. Verification

In order to verify the RSA hash signature, the module needs to calculate the SHA-1 hash of the same data that has been signed. When the hash of the data has been calculated we used the public RSA key to extract the SHA-1 hash from the signed hash. If the hash sums are equal then the signature is considered valid. We used a predefined function in `rsa.c` that can verify signatures with public RSA keys. This is the function: `int rsa_pkcs1_verify( rsa_context *ctx, int mode, int hash_id, unsigned int hashlen, const unsigned char *hash, unsigned char *sig )`.

The function is similar to the function described in subsection 3.5.6 with the main difference that `unsigned char *sig` is the pointer to the signature. The function returns `0` if the signature is considered valid.

The verification is done in four different parts on the module of which two are made with the Master key, the Master key is used when a FOTA upgrade is initiated and in the bootloader before it runs the firmware to make sure that the program has not been altered. The third verification of data is done with the smaller Session key, this verification is done when the firmware version and length of the firmware is sent to the module. The last verification is done with the smaller Session key as well, but this verification is done many times, it verifies each data packet of the firmware that is received.

3.5.7.1. Validation in FOTA

We made a function called `validSignature`, seen in listing 3.13 at page 33, that calculates the hash of data with the specified length and returns TRUE if the rsa.c function rsa_pkcs1_verify(&key, RSA_PUBLIC, SIG_RSA_SHA1, 0, rsa_SHA1_hash, signature) succeeds, in other words if the SHA-1 hash function of the data is equal to the hash received when the public key is applied to the signed hash that was sent to the module. Both the validation of Master and Session key signatures use this function.

The public key for the Master key is stored in the EEPROM while the public key for the Session key is received from the client and stored as a variable temporarily during the update.

```
static cb_boolean validSignature(rsa_context key, unsigned char *↩
    calculateHashOf, cb_uint32 len, unsigned char *signature){

        unsigned char *rsa_SHA1_hash;

        sha1_init();
        sha1_write(calculateHashOf, len);
        rsa_SHA1_hash = (unsigned char *) sha1_result();

        if(rsa_pkcs1_verify(&key, RSA_PUBLIC, SIG_RSA_SHA1, 0, ↩
            rsa_SHA1_hash, signature) == 0 )
                return TRUE;
        return FALSE;
}
```

Listing 3.13: The validSignature function.

3.5.7.2. Validation in the bootloader

The validation function in the bootloader called isValidFirmware, seen in listing 3.14 on page 34 is basically the same except a few things. It checks if the first bytes in the firmware are 0xFFFFFFFF, which means that there is no firmware on the module. It also checks if the length of the firmware, which is stored at a specific address called FW_LENGTH_ADDR in the EEPROM, is too big to fit on the module. Other than that, the function is the same except that it reads the public Master key from the addresses PUBLIC_N_ADDR and PUBLIC_E_ADDR inside the EEPROM.

```
static cb_boolean isValidFirmware(){

        cb_uint32 length;
        cb_uint8 i;
        cb_boolean valid;
        rsa_context root_key;
        unsigned char *rsa_SHA1_hash;

    valid = FALSE;

        memcpy(&length, FW_LENGTH_ADDR , sizeof(cb_uint32));

        if(*(cb_uint32*)APPLICATION_ADDR == 0xFFFFFFFF)
                return valid;

        if(length => 0x1A0000 || length == 0)
                return valid;

        rsa_init(&root_key, RSA_PKCS_V15, 0);
        root_key.len = ROOT_KEY_LEN;
        mpi_read_binary(&root_key.N, PUBLIC_N_ADDR, ROOT_KEY_LEN);
        mpi_read_binary(&root_key.E, PUBLIC_E_ADDR, 4);

        sha1_init();
        sha1_write(APPLICATION_ADDR, length);
        rsa_SHA1_hash = (unsigned char *) sha1_result();

        if(rsa_pkcs1_verify(&root_key, RSA_PUBLIC, SIG_RSA_SHA1, 0, ↩
            rsa_SHA1_hash, SIGNED_HASH_ADDR) == 0 ){
                // This was not signed with the root key from u-blox
                valid = TRUE;
        }

        return valid;
}
```

Listing 3.14: The `isValidFirmware` function in the bootloader.

3.5.8. Authentication data packet

We decided to make a data packet to manage authentication of the Session key, called *"Authentication data packet"*. The Client would send an Authentication data packet via a FOTA data packet. The Authentication data packet can be seen in figure 3.3, on page 35 .

We introduced a 32 bit variable (`uint32 current_KEY_version`) that was stored in the module's EEPROM. During the initiation of the FOTA the module was implemented to read the stored value from a defined ad-

dress (KEY_VERSION_ADDR). When the Authentication data packet is received from the client the module compares
current_KEY_version with the KEY_version data field in the Authentication header. If the KEY_version is an older version then the module will respond with AUTH_DENY, as shown in listing 3.15.

```
//auth: pointer to Authentication header
if(auth->KEY_version <= current_KEY_version){
    // This is an old key version, not valid for the firmware update
    // Reply with AUTH_DENY
    sendData((uint8*) &sendingHeader, FOTA_HEADER_SIZE + 1);
    cbLED_setColor(cbLED_OFF);
    cbOS_delay(3000000); // 3s
    cbHW_forceFOTABoot(0);
}
```

Listing 3.15: Validating if it is an old Session key.

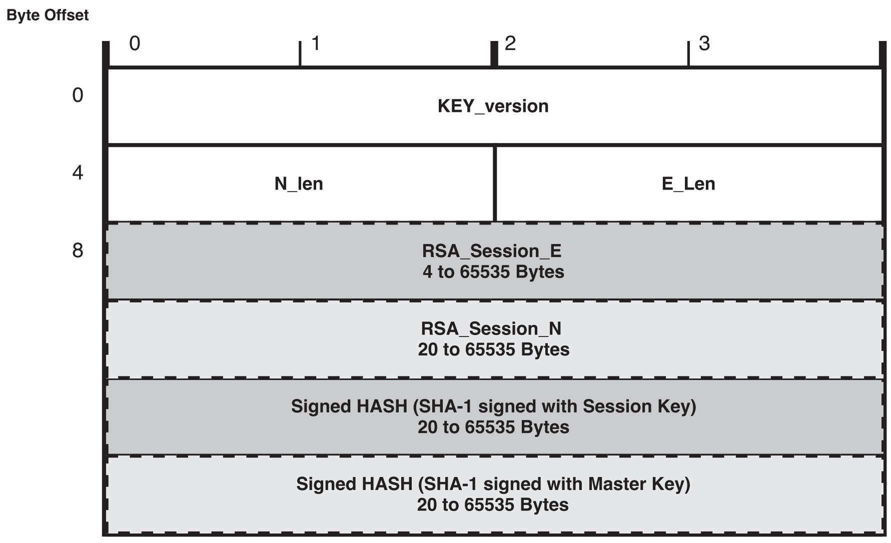


Figure 3.3.: Structure of a theoretical Authentication data packet

If the KEY_version is considered a newer key version, the public Session key is extracted with the N_len and E_len data fields. These fields simply tell the byte offset where the parameters of the public Session key is located. The reason that the minimum size of the public exponent (RSA_Session_E) is 4 bytes, is because it is represented in series of 32 bit words. This applies to the public modulus as well, but the minimum size of `RSA_Session_N` is 20 bytes. The reason for this is that we use *SHA-1* as the hash function, see subsection 3.5.5, page 31. To validate that the public Session key is associated with the private Session key we decided to sign the Authentication header (`KEY_version`, `RSA_Session_E` and `RSA_Session_N`) on the client side of the solution. This signature can be seen in figure 3.3. The size of the signature is the same size as the public modulus, `RSA_Session_N`. The module verifies that the Session key works with the verification function, subsection 3.5.7 on page 32. To make the module validate that the Session key is a trusted RSA key from u-blox, we signed the whole Authentication header and the signature from the Session key with the Master key. The signature can be seen on figure 3.3 on page 35. The signature from the Master key is validated with the public Master key stored on the module.

3.5.9. Verifying the firmware

We wanted the module to only accept newer firmware version, i.e we did not want to downgrade the module. Once the Session key has been validated by the module we sent a *First-packet* data packet to the module. Figure 3.4 shows the structure of the packet.

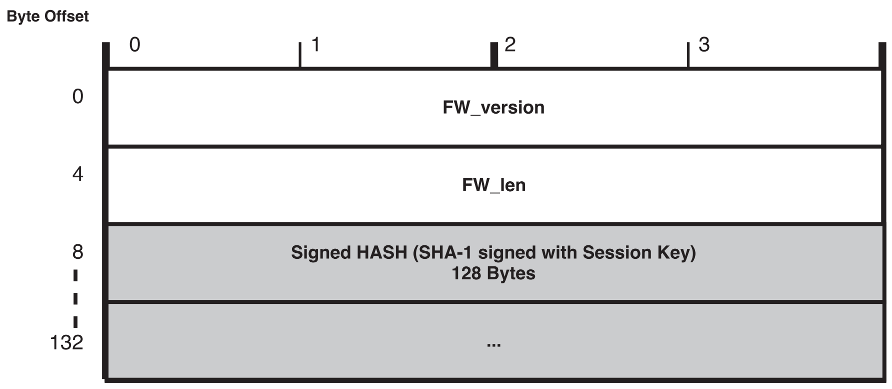


Figure 3.4.: Structure of First-packet data packet.

We introduced an 32 bit variable, `static uint32 current_FW_version`, to keep track on what firmware version was running on the module. The value is stored in the module's EEPROM at a dedicated address (`FW_VERSION_ADDR`). During the start up of FOTA, `current_FW_version` is assigned the value contained in `FW_VERSION_ADDR`. If `FW_version` from the received First-packet is larger than the stored `current_FW_version` it is considered as an update of the current firmware on the module. `FW_len` is the length of the firmware and is treated as in subsection 3.3.2 on page 20. We signed the First-packet with the Session key so that the module can verify the firmware version.

4 Result

The result of this project is a working solution of updating firmware on the u-blox ODIN-W26x series with support to recover from a failed update. The program also protects the module from receiving and executing firmware that is not authenticated by u-blox with a 512 byte RSA key.

4.1. Bootloader

The resulting bootloader of the module became rather simple. As can be seen in figure 4.1 on page 40. The bootloader simply checks if a FOTA update has been activated, if it has then it starts the FOTA update. If a FOTA update has not been initiated it checks if the firmware on the module is valid firmware by applying the public Master key on the signed hash of the firmware, which is stored in the EEPROM, and compares it with the hash of the firmware. If the firmware is not valid, it needs new firmware and starts the FOTA update state machine and waits for a new connection to start a firmware upgrade. If FOTA has not been initiated and there is valid application firmware on the module, then the bootloader starts the application.

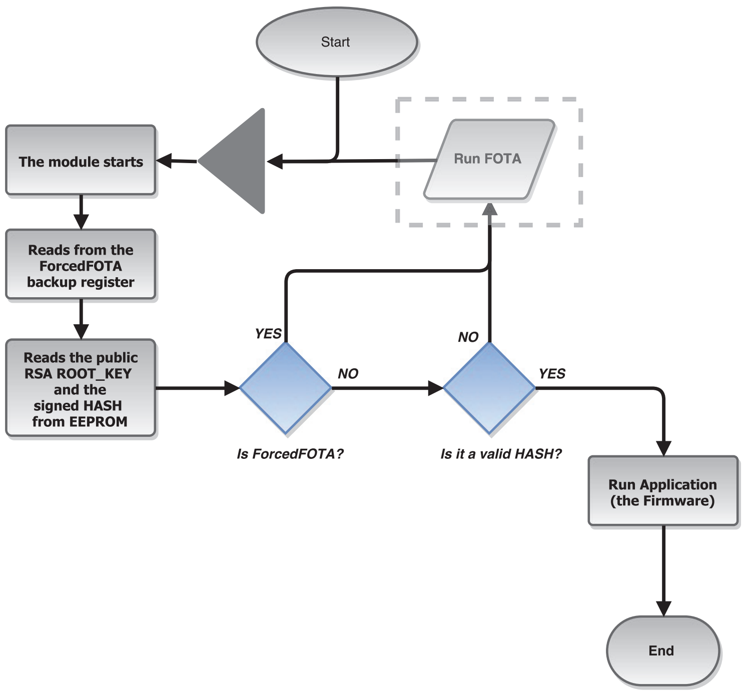

Start
The module starts
Run FOTA
Reads from the ForcedFOTA backup register
Reads the public RSA ROOT_KEY and the signed HASH from EEPROM
YES
NO
NO
YES
Is ForcedFOTA?
Is it a valid HASH?
Run Application (the Firmware)
End

Figure 4.1.: A flow chart diagram of the bootloader.

4.2. Signing the Firmware

The module can authenticate the firmware and verify whether the firmware is acceptable. The security of our solution is dependent on the private RSA key. The private RSA keys are never transmitted and can therefore never be sniffed. The module does not have access to the private RSA keys and therefore no one that has physical access to the module can extract them from the module's memory. We sign every packet that is going to be sent to the module and store it in a file. The file is then provided to the client. Since the file has been pre-signed by us, the client has no information about the private RSA keys.

4.3. FOTA

When the FOTA software is executed by the bootloader it reads the information needed for authentication from the EEPROM and starts a connection timer, the client will have 10 seconds to connection to the module or it will restart and enter the bootloader. If the client connects to the module the connection timer is cancelled and another timer is started. This timer resets every time a new data packet is received. When this is done the FOTA software enters the authentication handshake which is described in subsection 3.5.8, page 34. If the authentication fails the module will restart itself into the bootloader.

However if the authentication is successful, then the client will verify that the firmware is valid, this part of the program is described in 3.5.9, page 36. If it is a valid firmware the module is ready to receive the firmware, described in subsection 3.3.3 on page 23. Regardless if the firmware has been successfully sent or if it has failed, the module will restart and run the bootloader. This part of the program can be viewed in figure 4.2 on page 42.

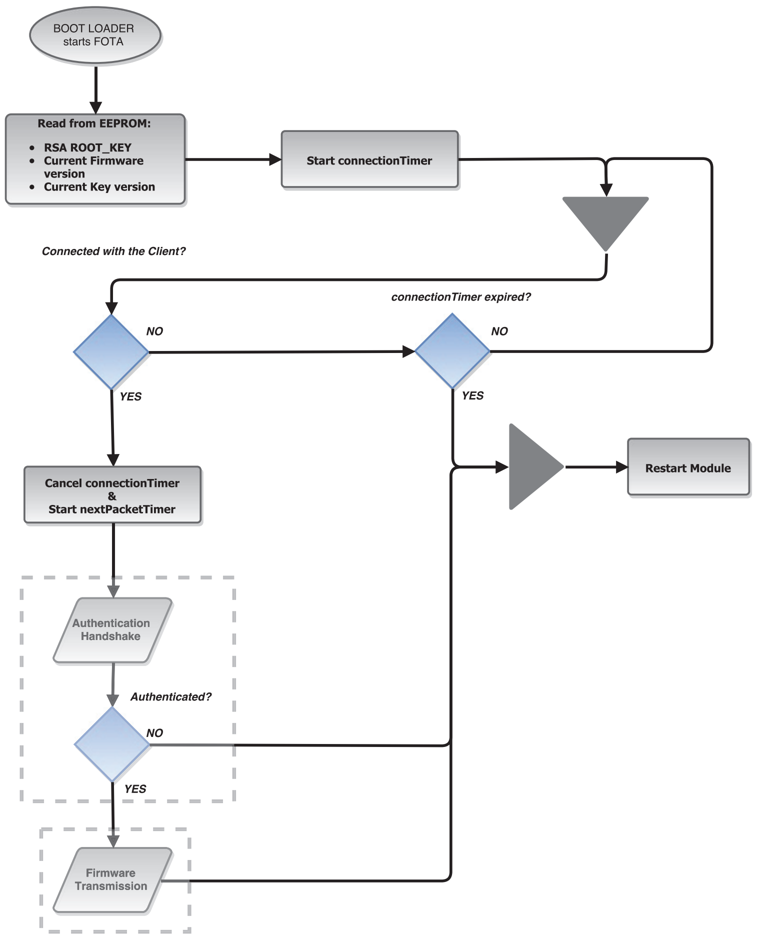


Figure 4.2.: A flow chart diagram of FOTA.

4.4. Authentication handshake

In figure 4.4 on page 44 you can see our final state machine of the authentication handshake process. The program waits for an Authentication data packet. If an Authentication data packet has not been received within 10 seconds, the module will exit FOTA by restarting itself. If an Authentication data packet is received the module will try to verify that the firmware is from u-blox, as described in subsection 3.5.8 on page 34. Since our solution uses two kind of RSA keys (Master key and Session key) with the sizes 512 bytes and 128 bytes, our Authentication is designed as in figure 4.3. If the authentication fails the module will send an `AUTH_DENY` to the client and restart itself.

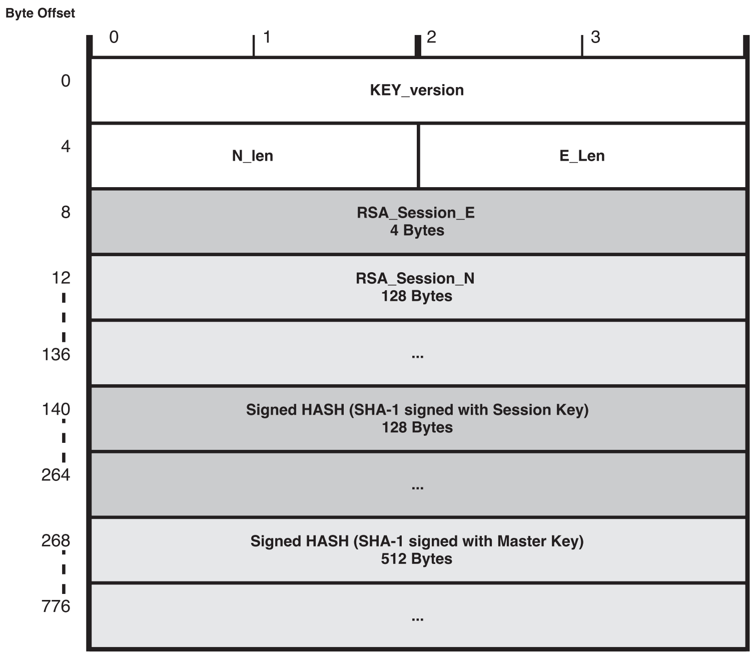


Figure 4.3.: Authentication data packet used in our solution.

If the authentication succeeds the module send `AUTH_OK` to the client. The module now possesses the Session key which is used for verifying the received FOTA data packets. Then the module will wait for a First-packet.

If a First-packet is not received within 1 second, the module will restart itself and enter the bootloader. If a First-packet is received the module will verify that the firmware is newer version than the current one. The module will also check if the firmware is to large for the module. If the firmware is not a newer one or the length of the firmware is to large, the module will respond with an `AUTH_DENY`. If they are correct the module will respond with an `AUTH_OK` and erase the old firmware. The module is now ready for the firmware transmission.

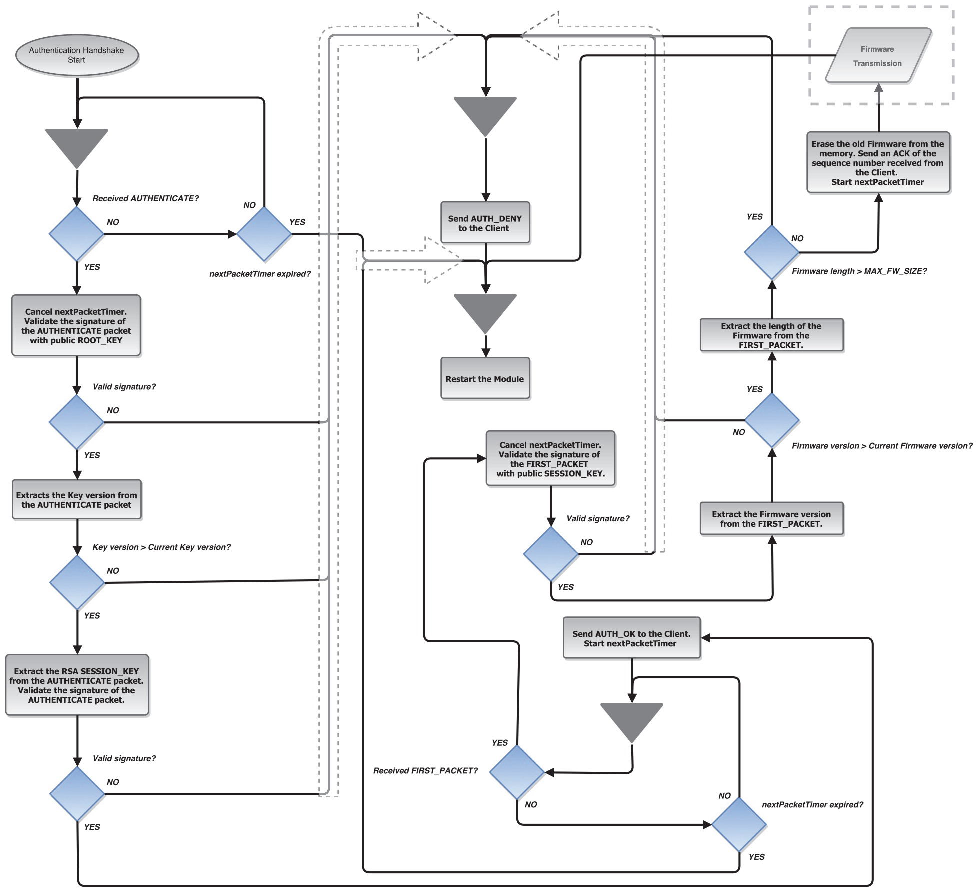

Authentication Handshake Start
Received AUTHENTICATE?
NO
YES
NO
YES
nextPacketTimer expired?
Cancel nextPacketTimer. Validate the signature of the AUTHENTICATE packet with public ROOT_KEY
Valid signature?
NO
YES
Extracts the Key version from the AUTHENTICATE packet
Key version > Current Key version?
NO
YES
Extract the RSA SESSION_KEY from the AUTHENTICATE packet. Validate the signature of the AUTHENTICATE packet.
Valid signature?
NO
YES
Send AUTH_DENY to the Client
Restart the Module
Cancel nextPacketTimer. Validate the signature of the FIRST_PACKET with public SESSION_KEY.
Valid signature?
NO
YES
Send AUTH_OK to the Client. Start nextPacketTimer
YES
Received FIRST_PACKET?
NO
NO
nextPacketTimer expired?
YES
Extract the Firmware version from the FIRST_PACKET.
Firmware version > Current Firmware version?
NO
YES
Extract the length of the Firmware from the FIRST_PACKET.
Firmware length > MAX_FW_SIZE?
YES
NO
Erase the old Firmware from the memory. Send an ACK of the sequence number received from the Client. Start nextPacketTimer
Firmware Transmission

Figure 4.4.: A flow chart diagram of the authentication handshake.

4.5. Firmware transmission

In figure 4.5 on page 46 you can see the final state machine of the module when firmware is being received. The program waits for the data to start sending, if it does not receive any new firmware packet within a second the module restarts.

If it however gets a new packet it evaluates the data packet with the public Session key to make sure that the packet is valid. If the signature does not match, then the module breaks the connection and restarts.

If the signature was accepted then the program checks if the incoming data is really the next data in line by comparing the sequence number with the ACK number, in case a packet of data has been lost on the way. If the sequence number is not equal to the ACK number, then the module asks for the client to re-send the lost data packet. If the packet actually is the next packet in line, then the program writes the data to the memory and increments the ACK number and the number of bytes received by the module with the length of the data.

After that the module asks for the next packet unless the number of bytes received is equal to the length of the firmware to be downloaded, because then the firmware is completely downloaded. The program starts the next packet timer and waits one second for the Master key signed hash of the entire firmware. If this signature is accepted by the module then the FOTA program writes the received signed hash of the firmware with the new Key version and the new Firmware version into the EEPROM and notifies the client that the update was completed and restarts the module.

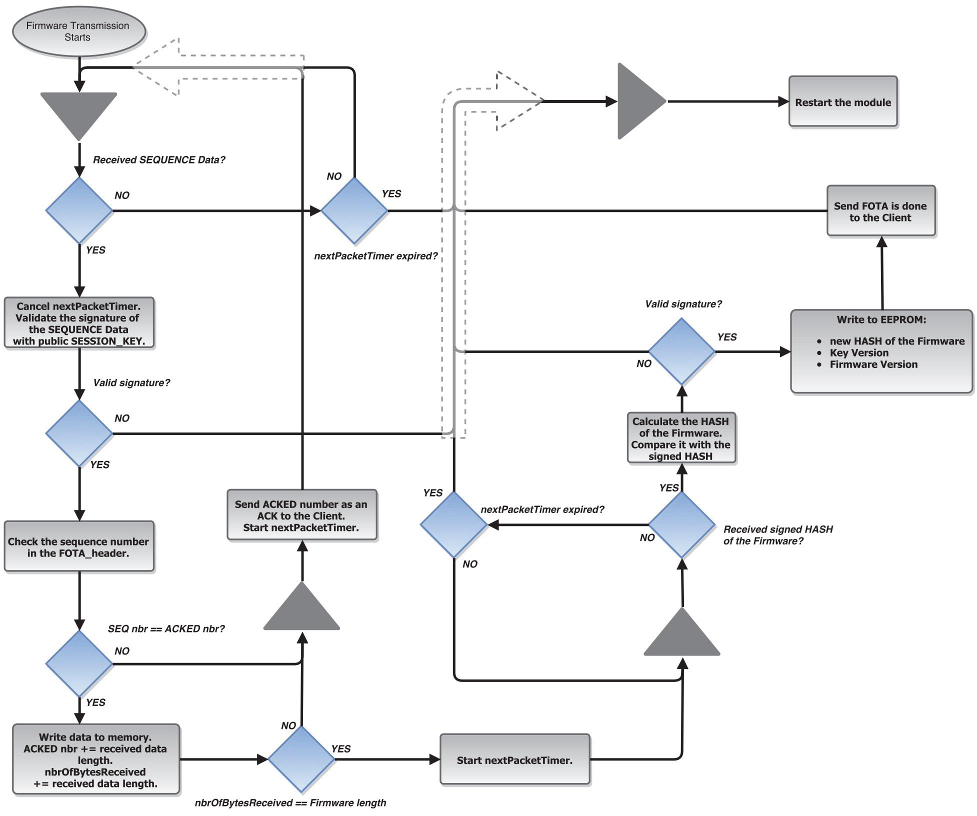

Firmware Transmission Starts
Received SEQUENCE Data?
NO
YES
NO
YES
nextPacketTimer expired?
Restart the module
Send FOTA is done to the Client
Cancel nextPacketTimer. Validate the signature of the SEQUENCE Data with public SESSION_KEY.
Valid signature?
NO
YES
Valid signature?
YES
NO
Write to EEPROM:
• new HASH of the Firmware
• Key Version
• Firmware Version
Calculate the HASH of the Firmware. Compare it with the signed HASH
YES
Send ACKED number as an ACK to the Client. Start nextPacketTimer.
YES
nextPacketTimer expired?
NO
Received signed HASH of the Firmware?
NO
Check the sequence number in the FOTA_header.
SEQ nbr == ACKED nbr?
NO
YES
Write data to memory. ACKED nbr += received data length. nbrOfBytesReceived += received data length.
NO
YES
nbrOfBytesReceived == Firmware length
Start nextPacketTimer.

Figure 4.5.: A flow chart diagram of the firmware transmission.

4.6. Size of firmware and performance

The resulting FOTA firmware is 133 kB in size. When updating the firmware on the module the FOTA firmware has a static erase time of the memory on 27 seconds. When the memory has been erased the module has a firmware transmission rate of 5.2 $kBps$.

5 | Discussion and conclusions

As this master thesis project came to an end there were still thoughts and ideas of improvements for the program, that we simply did not have time to introduce in the program.

5.1. Reducing the program size

The goal of the project was to fit the program so that the size was less than 128 kB. If the size would be less than 128 kB then it would be able to fit within one sector in the memory. The final project size was 133 kB[1] which was a little bit bigger than the goal. The fact that the size got just a little bit bigger than 128 kB meant that another sector of memory would have to be used, a sector of 128 kB. This means that 128 kB that could be used for the application is lost. However, given more time to optimize the code we believe that the code could easily fit within 128 kB.

5.1.1. Duplicate radio patch

A huge problem with the size is the radio patch file, which is 58 kB in size. This radio patch is both used by our FOTA code and the application, which means that there is 58 kB of duplicated code. So instead of having the duplicated code of the radio patch we suggest that the radio patch is stored at a specific address in the memory as a function that can be called. That way you could cut off a huge chunk of data from the FOTA code and fit the program into 128 kB, which also gives more space for the application.

5.1.2. Unused HCI code

By taking a thorough look at the HCI event and HCI command handler which includes several huge switch cases, the compilator can not distinguish that these events or commands are not called from the radio, hence the compilator can not optimize these switch cases. Knowing this one can manually optimize these handlers by removing the events and commands that will never be used from the switch cases.

5.1.3. CRC

By looking closer at ACL data packets we discovered that these packets include a CRC, hence the CRC that we implemented in our project is

[1]The FOTA program's actual size is 135500 bytes

redundant since the ACL data packet covers the information we send. Removing the CRC calculations and checks on the module would save a little bit of memory and process time.

5.2. Using the One Time Programmable memory

The OTP (One Time Programmable) memory is a part of the memory that can only be written data to once. During the lifetime of the module we do not want the Master key to be changed so the key is supposed to be stored in the OTP, however in the current program this is not the case. The key is stored in the EEPROM of the modules memory in the current project. The reason for this is because we simply did not want to ruin the modules memory by writing something into the OTP that later could not be changed. If however the code would be used in u-blox products later then the public Master key should be stored in the OTP memory. This could be done in manufacturing of the module. This is also the reason we choose a 512 bytes RSA key because the OTP memory is 528 bytes and the RSA key would have to fit in the OTP.

5.3. SPP stack

An interesting thought about this project is if one would extend the Bluetooth stack so that SPP (Serial Port Profile) was used. The reason why this is interesting is because Android phones use SPP. The project could be extended so that smart phones are able to connect and update the firmware on the module if the memory restrictions allow it.

5.4. Security

5.4.1. SHA-1 hash

When we developed the program we did it step by step increasing the functionality and security of the program as the project progressed. We started with the SHA-1 hash just to have a hash that could be signed with RSA. After a while we realized that the most vulnerable part of our data security solution over the air is our SHA-1 hash. Since the SHA-1 hash is only 20 bytes it is much easier to find malicious code that has hash collisions with the valid code. This malicious code could then be sent to the module with the same RSA signed hash and get accepted. According to BlueKrypt [4], NIST does not recommend the use of SHA-1 for digital signatures in the year 2015. They recommend using any of the following SHA hash functions: SHA-224, SHA-256, SHA-384 or SHA-512. So in order to increase the security of our solution the SHA-1 hash would have to be switched for a more advanced hash function.

5.4.2. Physical access to the module

The absolute biggest problem with our security is if someone would get physical access to the module. If one has access to the module's memory one can simply replace the public Master key with another public RSA key. The module would not notice that the Master key has been replaced. This means that the module will not accept any firmware updates from u-blox. People can then use the private RSA key to update the module with firmware that is not intended for the module.

However, if a person has physical access to the module hir[2] could just simply erase the whole memory and load it with whatever hir wants.

5.4.3. Encryption of the data

In our project we use RSA keys to sign and verify data. RSA keys can also be used for encryption of the data. We thought it was a good idea to encrypt the firmware since anyone could sniff our wireless traffic, but we did not do it. The reason for this is that the firmwares that u-blox provide to their modules is not secret, one can simply download binary files from their homepage. If the firmware is publicly known it does not matter if one can sniff it.

5.5. Robustness

Another goal of the project was to make the updating program robust, so that it could recover from a failed firmware update whether it being a power loss, incorrect firmware or an aborted update. Since the bootloader

[2]A gender-neutral personal pronoun. Pronunciation: ”here”

starts into FOTA if anything is wrong with the firmware in most cases it is robust because it can always recover by receiving a new update of the firmware. However, if the module would receive a power loss in the unlikely event that it is currently erasing the Key version to write down the new one. Then the Key version would have the value `0xFFFFFFFF`, which is the highest possible value for the Key version to be. There is no way for the module to accept new firmware. There are ways of covering for this unlikely event, one way would be to store another copy of the Key version in another part of the EEPROM, but that would cost more memory.

5.6. Performance

As mentioned in result the firmware transmission rate was 5.2 $kBps$. This result might seem a little low but we have to bear in mind that the maximum allowed size of the application firmware is 1.625 MB. If a firmware of maximum size was downloaded to the module then the download would take approximately 5 minutes and 47 seconds. This includes the 27 seconds of erasing the memory.

The performance on the time it takes to update the module has never been a high priority in this project because the modules do usually not need many firmware upgrades in their lifetime. The most important part was to make the FOTA firmware fit within reasonable memory size so the application firmware size was not affected too much.

5.6.1. Increasing the throughput

There are however some ideas on how to increase the throughput of the data. Our solution sends an ACK for every FOTA data packet it receives. Instead of reply on every packet the module could use a buffer to store several packets, like a sliding window used in TCP. The client does not have to wait for ACKs as often.

The same thing applies for the verification of the data packets. Instead of signing every packet one can simply sign a bunch of them. There would be less signatures to be verified, hence more space can be used for the firmware data in the FOTA data packets.

A | List of Acronyms

ACK Acknowledgement
ACL Asynchronous connection-less
CRC Cyclic Redundancy Check
EEPROM Electrically Erasable Programmable Read-Only Memory
FOTA Firmware update Over The Air
HCI Host Controller Interface
L2CAP Logical Link Control and Adaptation Protocol
TCP Transmission Control Protocol
UART Universal Asynchronous Receiver/Transmitter

Bibliography

[1] Paul C. van Oorchot snd Scott A. Vanstone Alfred J. Menezes. *HANDBOOK of APPLIED CRYPTOGRAPHY*. CRC Press, 1996.

[2] Jennifer Bray and Charles F Sturman. *BLUETOOTH 1.1 Connect Without Cables*. Bernard Goodwin, 2001.

[3] Wei Dai. Sha-1. `http://oauth.googlecode.com/svn/code/c/liboauth/src/sha1.c`.

[4] Damien Giry. Bluekrypt: Cryptographic key length recommendation. `http://www.keylength.com/en/compare/`.

[5] Texas Instruments. Hcitester. `http://processors.wiki.ti.com/index.php/LPRF_BLE_HCITester`.