CSC 714 Real Time Computer Systems

Active Messages for the Renesas M16 Board

Final Project Report

Manan Shah
Trushant Kalyanpur

Final Project Report

Goals Achieved:

1) We managed to successfully establish communication between the mote and the M16C attached to the Zigbee board. The data payload of TinyOS Active Message sent by the mote was successfully parsed and the data was verified to be correct.
2) We managed to establish some sort of communication in the other direction as well i.e. between the M16C board and the mote. The details of the progress in this direction are included in the report.

Application Tested:

We wrote a sample application to transfer temperature readings from the mote to verify communication between the mote and the M16C board. The mote sends out temperature values at repeated intervals and the M16C correctly extracts this information from the active message received. We verified this data with the message received by another mote connected to a PC, which displayed the packet contents using a Java program.

Issues Solved while achieving our Goals:

On the Mote Side:

Interpretation of the frame format

In the previous report we mentioned that there was a problem in interpreting the format of the packet received on the M1. In order to determine that let us first look at what a typical data packet for the 804.15.4 protocol looks like.

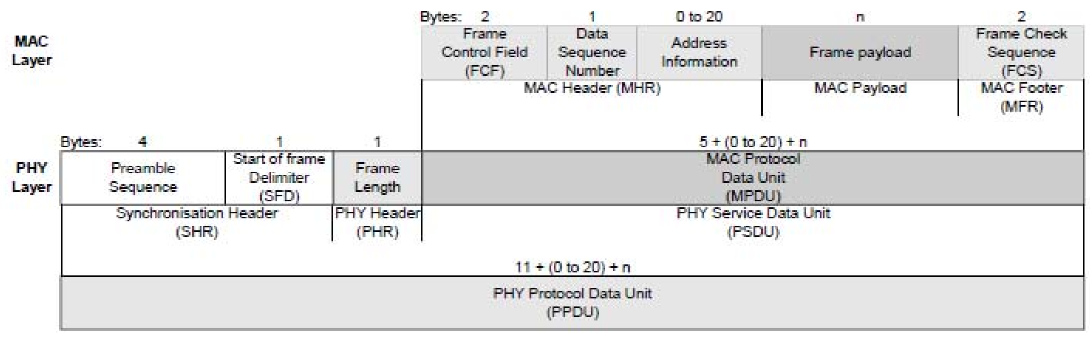


Figure 17. Schematic view of the IEEE 802.15.4 Frame Format [1]

The frame format is as seen in the figure above. The *preamble sequence* for an IEEE 804.15.4 frame is 8 zeros followed by the sequence 0xA7, the byte A7 is stored in the *start of frame delimite*r and the 8 zeros are used for synchronization purposes. The field *frame length* must include the length of the entire payload from the frame control sequence to the frame check sequence. The Frame control field is the key to identifying the type of the packet. The data sequence number is incremented so as to avoid duplicate frames from being accepted. The address information contains the address of the transmitter or the intended receiver. The frame payload contains the data to be transmitted and the frame check sequence verifies the integrity of the frame.

On studying the chip documentation and the Telos packet we observed the following:
The CC2420 chip implements the preamble sequence and the start of frame delimiter by setting the SYNCWORD register to 0xA70f and the preamble field of the MDMCNTRL0 register to 2(Refer the Chipcon CC2420 datasheet Page number 36 for more details).

The CC2420 chip also fills in the frame check sequence on transmission and also verifies the data integrity on reception but the frame check sequence is not written to the receiver or transmitter FIFO.

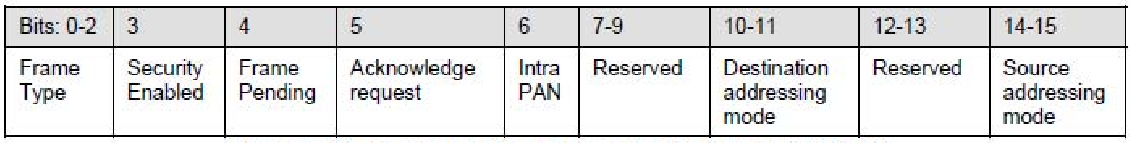
The remaining fields are filled by the Telosb mote by using the following data structure:

```
typedef struct TOS_Msg {


  uint8_t length;
  uint8_t fcfhi;
  uint8_t fcflo;
  uint8_t dsn;
  uint16_t destpan;
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  int8_t data[28];
  uint8_t strength;
  uint8_t lqi;
  bool crc;
  bool ack;
  uint16_t time;
} __attribute((packed))  TOS_Msg;
```

The fields correspond to the frame format shown above. The *destpan*, *addr*, *type* and the *group* fields come under the address information field of the frame. The *strength* and the *lqi* fields are filled in at the receiver side using the signal strength at the receiver. The *data*[28] stores the data packet to be transmitted. The *crc* and the *ack* fields say whether the CRC and the acknowledgements are implemented or not. The *time* field stores the time stamp of the data that is transmitted. The last three fields are the part of the data packet and are parsed upon reception by the receiver.

During our presentation we had pointed out that the FCF fields were interchanged. Let us first understand what the FCF frame looks like:



Bits: 0-2	3	4	5	6	7-9	10-11	12-13	14-15
Frame Type	Security Enabled	Frame Pending	Acknowledge request	Intra PAN	Reserved	Destination addressing mode	Reserved	Source addressing mode

Figure 19. Format of the Frame Control Field (FCF) [1]

As seen above the Telos structure divides this frame into two parts namely:

```
uint8_t fcfhi;
uint8_t fcflo;
```

They are initialized by the following two constants in the CC2420Const.h

```
#define CC2420_DEF_FCF_LO          0x08
#define CC2420_DEF_FCF_HI          0x01  // without ACK
#define CC2420_DEF_FCF_HI_ACK      0x21  // with ACK
```

The following two fields were interpreted by us as follows:
For the high byte: Dest Addressing Mode and the source addressing mode is set to 0 in case of 0x01 and 0x21
For the low byte the frame type is zero; the security enabled bit is set; the frame pending the acknowledge request and the intra pan are set to zero.

Now we see a few anomalies in this. First the HI byte is defined for *with ACK* and *without ACK*, while the ACK field is in the lower byte. Also there is no support for encryption and the bits set in the high frame come under the reserved field.

The frame field being set to zero is interpreted as a beacon and the mote has TinyOS data structure for the packet does not support the beacon format. It exactly resembles the data format.

Therefore we realized that the bits have been reversed. On making the assignment as follows:

```
#define CC2420_DEF_FCF_HI          0x08
#define CC2420_DEF_FCF_LO          0x01  // without ACK
#define CC2420_DEF_FCF_LO_ACK      0x21  // with ACK
```

Now the high byte the destination address is set to a value of 2.
For the low byte the value of the ACK is set and reset accordingly and the frame type bit is set to 1, which indicates a data frame.

The problem gets interesting due to the fact that the incorrectly transmitted frame of 0x0108 which was happening in the previous case was interpreted correctly as 0x0801 at the receiver board. This implies the receiver read the data in the opposite manner; hence the incorrect code seems to work.

We have therefore realized that the interpretation of the FCF field is somewhat ambiguous.

Data transmission from the Mote:

The active messages generated by the applications are passed on to the component AMStandard.nc which sets the following fields: *length*, *addr*, *type* and *group*.

The AMStandard passes the message to the CC2420RadioM component which further sets the following fields: *fcfhi*, *fcflo*, *destpan*, adjust the destination in the correct byte order, the length of the message (*length*) and the sequence number of the message(*dsn*).

The message is next passed to the component which writes the data to the transmit buffer and the data is sent through the radio antenna.

Data Reception on the Mote:

When the data is received on the mote, the receive buffer gets full and the FIFOP interrupt is generated on receiving this interrupt. The program checks whether the FIFO has overflowed, if it has then the FIFO is flushed and no further action is taken. Otherwise a function named "*delayedRXFIFO*" is called. It first checks if a previous packet is still in the process of being read, if this condition is true then the task waits till the read operation has finished. If no packet is being received then the task calls the Chipcon component to read the FIFO and passes to it the *rxbufptr* to store the read value. Once the read process is over, it notifies the cc2420Radio component by signalling the *RXFIFODone* function.

This function checks if the anticipated data packet is of the correct size or if it is not an empty packet. If any of the conditions are false the packet is flushed. It then checks if acknowledgement is required for the given packet, if enabled then an acknowledgement is sent. It also checks the FCF bits to see that the packet received is a valid packet. After all the checking is done it computes the length, address and the CRC of the packet. It also computes the signal strength and the LQI, this data is provided by the chip upon reception of the packet. It then calls the *PacketRcvd*() task which signals the *AMStandard.nc* component the packet is received. This components checks for a valid CRC bit, the group, and the address. If all these fields match it signals the application layer that the packet has been successfully received.

For this project, when the packet is sent from the board to the mote, initially we did not receive any indication at the application layer of the packet being received. In order to make packet reception possible we had to turn off all the packet checking mechanisms since we did not know about the format of the received packet. We used the LEDS to trace the passing of the received packet from the lower layer to the upper layers; we toggled the LEDS in each of the functions so that we knew till what stage the packet reached. After disabling all the checking mechanisms we were able to make the packet reach the application layer. The next step is to determine the exact structure of transmission if possible and then write the same structure on the reception side and parse the packet.

Problems with the parsing of the data packet:

We observed that the data packet transmitted from the Zigbee board conforms to the 802.15.14 standard, as shown above but the TOS message structure was not exactly equivalent to the 802.15.14 standard. During reception the address fields and the data fields on the mote side are not parsed correctly.

The only fields that are parsed correctly are the *fcf* and the sequence number (*dsn*) and after that there is absolutely no correlation in the transmitted and the received data structures. In fact the data has no correlation and we could not identify the data pattern sent out by the board. This can be corrected in the future by writing a new data structure that conforms to the IEEE standards and then passing it to the application layer using a new interface or by passing the raw unformatted data upwards using a new interface. This unformatted data could be displayed on the computer and then accordingly parsed after some correlation is found.

Problem with the Java program:

It was observed that the Java program can take in packets having a maximum size of 28 bytes. We could not change the packet size since we are not very familiar with Java and the program structure in the header files. In order to get around this problem we tried we split the data packet into two packets and sent it to the Java program. When two packets are sent simultaneously then the data sent in the second packet is incorrect. Therefore we have written two programs, one to transmit the header and the other to transmit the data fields. The lack of support for unsigned numbers in java causes difficulty in interpreting the packets, which is one major disadvantage of using the Java programming language.

On the M16C Side:

Data Reception:

We first confirmed the reception of a message from the mote by using the *mcpsDataIndication* function, which is generated by the MAC layer when a data frame is successfully received. Glowing LED's on the M16 board confirmed the reception of some sort of message being received by the board. The next step was to identify whether the message received was identical to the one sent out by the mote.

To clearly identify the payload field of the Active Message sent out by the mote, we sent a specific data pattern in the payload from the mote. We observed that the expected payload was actually received at an offset from the payload on the M16 board. The data from the mote actually begins from byte 2 of the received payload i.e. *pMsdu*[2] onwards. The first two bytes of pMsdu actually correspond to the *type* and the *group* fields of the active message.

We managed to successfully extract the following fields of the active message:

1) *length* – The length of the payload in bytes.

2) *fcfhi* – The HI byte of the FCF
3) *fcflo* - The LO byte of the FCF
4) *dsn* – The sequence number
5) *destpan* – The destination PAN ID
6) *addr* – The destination Address
7) *type* – The type field
8) *group* – The group field
9) *data[28]* – The data payload (28 bytes)

The problem we faced while extracting this information was that the data passed to the application layer did not contain all the information mentioned above. So we had to use the KD30 debugger to identify the contents of the packet received at the lower MAC layer and we extracted the Active Message information from the MAC layer directly. In particular, the *MAC_RX_INFO* (*mrxInfo*) structure defined in the *mac_rx_engine.h* header file contained the information of the received packet.

Data Transmission:

The packet that is transmitted consists of a header upto 23 bytes (2 byte FCF + 1 byte sequence number + upto 20 bytes of Address information), 102 bytes of the data payload and 3 bytes of the trailer (Frame Check Sequence FCS).

We transmitted a packet using the *mcpsDataRequest* function, and used no transmission options (*txOptions*). The *mcpsDataConfirm* function is called by the MAC layer when a packet to be transmitted has succeeded or failed. We verified the successful transmission of the data by toggling an LED on successful transmission.

The problem however, is that we have not been able to extract the data payload sent by the M16 board on the mote side. We used the Java program to display the contents of the received message. As stated previously the first four bytes (including the FCF and the sequence number) of the message seem to be parsed correctly. The remaining bytes seem to have no correlation to the data being sent by the board.

Unsolved Issues:

The LCD display does not seem to work when both the M16C board and the Zigbee board are connected. If the Zigbee board is disconnected, then the data is displayed correctly on the LCD. After the addition of the Zigbee board, the LCD display goes blank and changing the contrast on the M16C board doesn't seem to have any effect on the screen either. We therefore used the KD30 debugger to view the contents of the message received and correctly verified the temperature readings sent out by the mote.

On the moteside the packet received is not parsed correctly and hence no sense can be made out of the data.

Future Work:

1) Developing a nesC application to correctly parse and extract the message received by the mote and using a Java application to display the data received.
2) This project can be extended to display the information received by the M16C board on a web browser by connecting the M16C board to the SKPCOMMS board. The SKPCOMMS board has an Ethernet controller chip and it communicates to a PC via a LAN cable. Data such as the temperature readings from various motes can then be displayed on a web browser on the PC. Furthermore, a user can remotely monitor the data received by the M16C board by using the web browser from a remote location.

Individual Member Contributions:

Trushant was responsible for doing the work on the zigbee board side which involved receiving the message and parsing it. All the fields were extracted successfully and stored in an array. He was also responsible for formatting the transmission frame.

Manan was responsible for software development on the mote side. He wrote the temperature gathering application on the mote side and transmitted it using the radio channel. He was also responsible for studying the reception process and modifying the TinyOS files so that the message reached the application layer.

Link to the project Website