OpenBTS for Rural Communications: Software for Inter-BTS Communication

By

Ahmed Sayeed Mohammed Rawi

Ahmed Fargaly Abd-El Rahman

Ahmed Mohammed Othman

Ahmed Mahmoud Emam

Eslam Ahmed Saad

Under the Supervision of

Dr. Magdi Fikri

A Graduation Project Report Submitted to
the Faculty of Engineering at Cairo University
In Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science
in
Electronics and Communications Engineering

Faculty of Engineering, Cairo University

Giza, Egypt

July 2012

Table of Contents

List of Tables

Please insert your list of tables here. Below is an example of a list of tables generated using MS Word "Insert Table of Figures" feature.

List of Figures

List of Symbols and Abbreviations

Please insert here any symbols and abbreviations that you commonly use in your report.

Example:

SNR	Signal-to-Noise Ratio
Tx	Transmitter
Rx	Receiver

Acknowledgments

Please insert the acknowledgments here. In this section, you may thank those who provided help and support to you during the course of your project.

Abstract

Please insert the project abstract here. The abstract is a one-page summary of the project. It should include clearly what was done in the project and the reached results.

Chapter 1: Introduction To The Project

The Open BTS Project is an open-source UNIX application that uses the Universal Software Radio Peripheral (USRP) to present a GSM air interface ("Um") to standard GSM handset and uses the Asterisk VoIP PBX to connect calls. Open BTS replaces the entire setup in conventional GSM BTS shown in Figure 1, which is a dumb device that is managed externally by a base station controller (BSC) and connects calls in a remote mobile switching center (MSC).

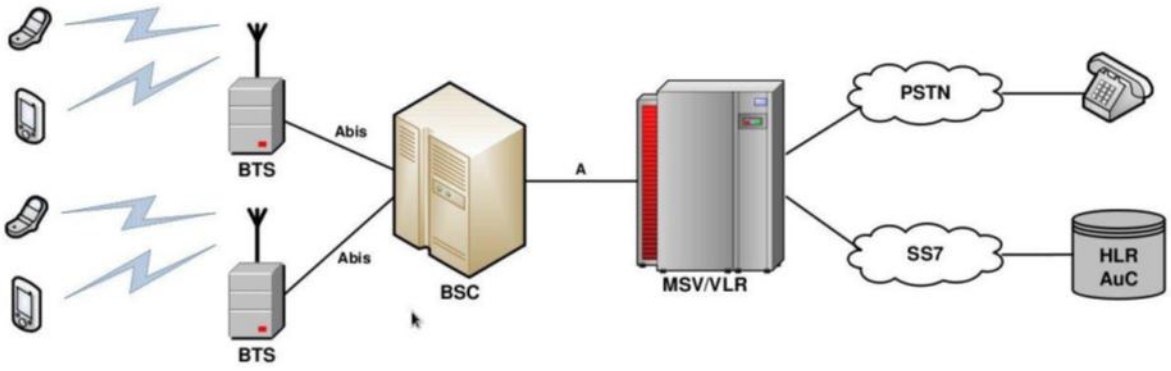

Figure 1: Typical GSM network diagram

The USRP is used to receive and transmit the GSM signaling using GNU Radio as driver software. OpenBTS package plays the role of MSC/VLR and Asterisk software PBX will be used to connect calls. Figure 2 shows a typical OpenBTS network.

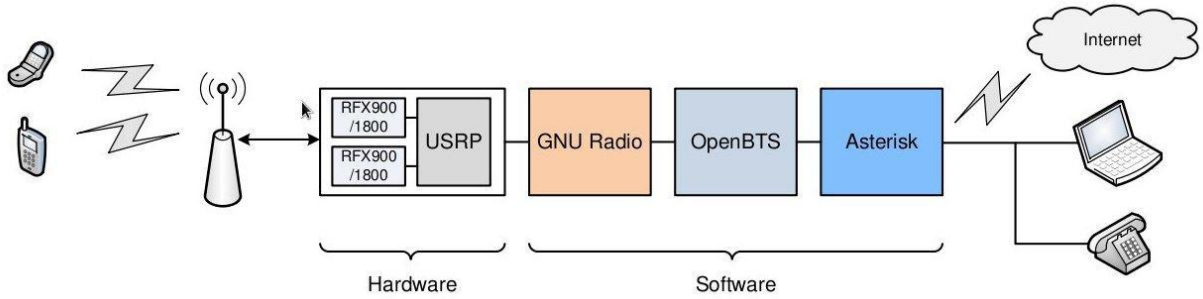

Figure 2: OpenBTS Network

I. Why Build an Open Source GSM Stack?

The combination of the ubiquitous GSM air interface with VoIP backhaul could form the basis of a new type of cellular network that could be deployed and operated at substantially lower cost than existing technologies.

Since these new hybrid networks are not readily compatible with legacy networks, and since radical two-tier pricing would be disruptive for existing carriers, we are not likely to see this kind of innovation from the conventional telecom community. This is the primary motivation for starting this project: a vision of truly universal telephone service.

II. GSM is old and boring. Why Not CDMA?

GSM is a good choice precisely because it is old and boring. Everyone knows it works and 80% of the world's carriers are still using it. It's a proven technology that is well-suited to the target application. The specification is publicly available and in a few more years most of the essential patents will expire.

CDMA physical layers are too complex for an inexpensive all-software radio and do not scale well for low-capacity cells. CMDA capacity comes in increments of 50 or more subscriber lines and the lowest layers of your radio must process all of that bandwidth whether you intend to use it or not. By contrast, GSM capacity comes in increments of 7-8 lines and a well-managed radio can even ignore inactive parts of the signal. Beyond the technical issues, IS-95-style CMDA (including cdma2000) is tightly controlled intellectual property. You can't even get a copy of the specification without signing an NDA and paying several hundred dollars.

III. What about GPRS/EDGE and UMTS?

Future versions of the Open BTS may well support GPRS and EDGE. GPRS, when available, should be a software-only upgrade for any installed Open BTS system. EDGE support may require additional computational resources but the additional software is not complex, at least when compared to the rest of the BTS. UMTS is a radically different CDMA-style physical layer and well outside the current scope of this project.

IV. What's wrong with WiFi, WiMax, WiWhatever?

There are a lot of people out there who would rather not blanket Africa with WiWhatever than find a way to make GSM dirt cheap. It's sexier to talk about the newest air interface and that talk gets a lot a buzz, but the truth is that that WiWhatever is poorly suited to mobile telephony. WiFi range is far too short for mobile coverage in rural areas. For example, you're not going to cover 700 square miles with a single WiFi tower, but that's exactly what GSM was made to do. If access points are connected through different ISPs, handovers will be unreliable. The phones are expensive and power-hungry, and compared to GSM they always will be.

WiFi may become a decent technology for semi-mobile telephones in dense urban areas, but it's not a mobile phone standard and it's not well-matched to the rural cellular application. WiMax has most of the problems of WiFi. To make matters worse, most WiMax bands don't penetrate structures very well. Most WiMax deployers are planning to solve this problem by saturating large buildings with small access points. That's fine in Manhattan and London, but we don't see anyone putting femtocells in a million houses when those households couldn't afford phones in the first place.

More important than all of that is to remember the goal of the Open BTS: universal telephone service. Our project philosophy is that it is much better to give people basic telephone service with an upgrade path to 250 kb/sec EDGE than to generate a lot of hype over a scorching fast broadband technology that can probably never be truly universal. Don't let the perfect be the enemy of the good.

Wi-Whatever's do have a place in Open BTS, though: backhaul. The standard Open BTS backhaul is likely to be redundant-path point-to-point WiFi or WiMax.

1.1 The Main Target

The main goal is to have all functions of BTS, BSC and MSC collapsed in the OpenBTS box. Also one needs to make the OpenBTS able to connect to another OpenBTS. The conventional way of the OpenBTS project to do this is by operating each BTS as an access point to the IP-Network, with a GSM Um interface to connect

the mobile sets (MS). The system architecture in this case is shown in Figure 3, where, for simplicity, two BTSs only are connected via the IP-network.

The Um interface is handled by the GSM side of the OpenBTS to manage connections of the MSs to the BTSs. The software modules of the GSM side are the GSM stack, with its 3–layer model, and the transceiver, which acts as a baseband modem, both running on the host processor. The radio hardware is implemented on the USRP, its daughterboard and external RF components. The software for the IP side is composed of a SIP message handler and VoIP soft switch, the ASTERISK. A GSM/SIP protocol processor module reacts to the L3 messages, including RR, MM and CM messages, and translates the MM and CM messages between the GSM and SIP sides, completing the software suite of the OpenBTS. Non-local calls are routed by the ASTERISK soft switch as VoIP traffic through the IP-Network to the other BTSs.

Such architecture requires an IP-network access at each BTS location, which is a costly solution for isolated rural areas. Hence we propose the following two alternatives for connecting the BTSs. One is through the use of a VoIP/GSM gateway to the BTS and the other is through modifying the OpenBTS software suite to act as a GSM repeater.

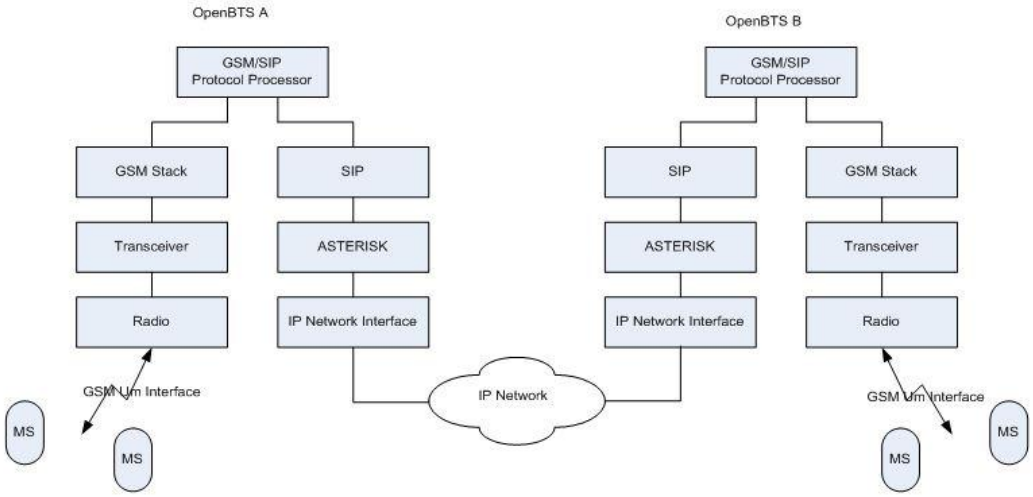

Figure 3: Original OpenBTS Connections Architecture.

We propose another alternative for connecting the BTSs through a GSM Um repeater, as shown in Figure 4 and described below.

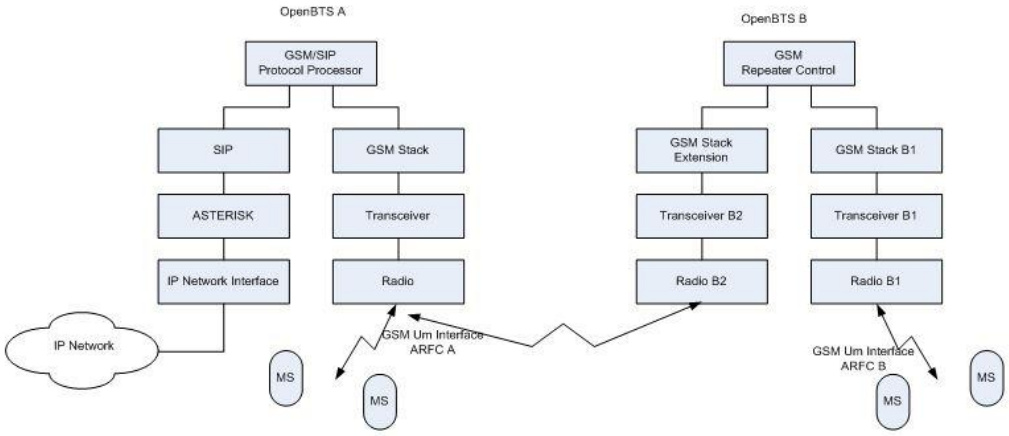

Figure 4: OpenBTS Connection through GSM Um Repeater.

In this architecture OpenBTS A operates as a master BTS, with its Asterisk's data base covering the other "slave" BTSs in the service area, and operates at an ARFCN A. OpenBTS B represents one of the slave BTSs, connected to its local MSs at an ARFCN B, and repeating MM and CC signaling messages, as well as some traffic bursts, to OpenBTS A at one or more time slots of ARFCN A.

The modules of OpenBTS B, other than the GSM Repeater Control module, pertain to either the MSs Side; which operates as a BTS in the same manner as the original OpenBTS modules, or the master BTS side; which operate as a MS attached to the master BTS. The differences between both side and the main functions of the GSM Repeater Control module are explained as follows.

Radio B2: This could be another USRP, just as Radio B1, with the down-link and up-link frequencies tuned to match receive and transmit frequencies of ARFCN A, respectively.

Transceiver B2: This is another instantiation of the transceiver module, with modifications to handle some of the up-link and down-link channels that are not used in the original OpenBTS code. These channels are the up-link frequency correction (FCCH), synchronization (SCH), access grant (AGCH) and paging (PCH) channels, and the down-link RATCH channel. This module should start by searching for the FCCH burst of the OpenBTS A and thence correct any frequency offset to synchronize its carrier with ARFCN A. Then it looks for the SCH burst of OpenBTS A and gets the frame number (FN) and time-slot number (TN) from it - with the help of the GSM Stack Extension module - and synchronize its FN and TN to the master BTS timing.

GSM Stack Extension: This module will instantiate some more objects from the original GSM Stack module to handle L1, L2 and L3 layers for the extra channels mentioned above paragraph to implement the MS mode of operation.

GSM Repeater Control: This module shall manage the operation of the repeater through proper initialization and attachment to the master BTS, then it continuously receive L3 messages from both the MS side and the BTS side and react to them in order to fulfill the repeater function as follow:

- RR Messages: Messages received from the MS side shall be reacted by assigning the required radio resources locally (SDCCH and TCH), in a way similar to the original OpenBTS code. RR messages from the master BTS side shall be used to extract and assign the SDCCH and TCH of the master BTS side. A table shall be held for the correspondence between the resources allocated to both sides.
- MM and CM Messages: Messages originating from one side shall be switched to the other side using the above RR correspondence table.

The above way of operation has the advantage of simplified control, but burdens the link between the master and slave with extra local traffic channels. This could be extended later by modifying the "alerting" CM message to indicate if a call is local to the slave BTS, and modify the TCH assignment to be a local time slot instead of going through the BTS's link.

It is worth noting that the all the slave BTS software can be run on a single PC, with 2 USB outlets connected to 2 USRPs, one for each ARFCN. There are also other modifications that should be made in the dial plan of the master ASTERISK, but this should be simple.

Another extension that makes use of one USRP can be made by reserving few time slots of ARFCN A for the local traffic of OpenBTS B, and operating the slave BTS at the same ARFCN A at the reserved time slots. This, however, requires fast uplink – downlink frequency switching during the burst time.

Also, the BTSs link time-slots capacity can also be tripled by using 8-PSK modulation instead of the binary GMSK, as used in EDGE.

1.2 Hardware Requirements

OpenBTS is based on series of hardware combined together to operate system in a proper way. These hardware parts are listed as follow:

- Computer: The essential part to run the OpenBTS code on terminal of a Unix-based system like Ubuntu 11.04. The USRP boards are connected to it through USB ports to be controlled and handled by the software compiled code.
- USRP 1: (Universal Software Radio Peripheral) board which contains four main ICs: FPGA (Altera - Cyclone), two Analog Devices MxFE and MCU. It's shown in figure 3.
- Daughterboard: is a small board installed on the USRP motherboard, it's used to get a specific range of GSM band. The used daughterboard in the implemented system is WBX which supports range 64 MHz.
- Antenna: WBX daughterboard can support two antennas, one TX/RX and the other just RX. Antennas make the air interface part in the OpenBTS system to receive calls from mobile sets and let USRPs talk together, the air interface between the master and slave USRPs.
- Mobile Phones: used to make calls on the supported range of frequency. By switching the automatic network to the manual scan and select OpenBTS range.

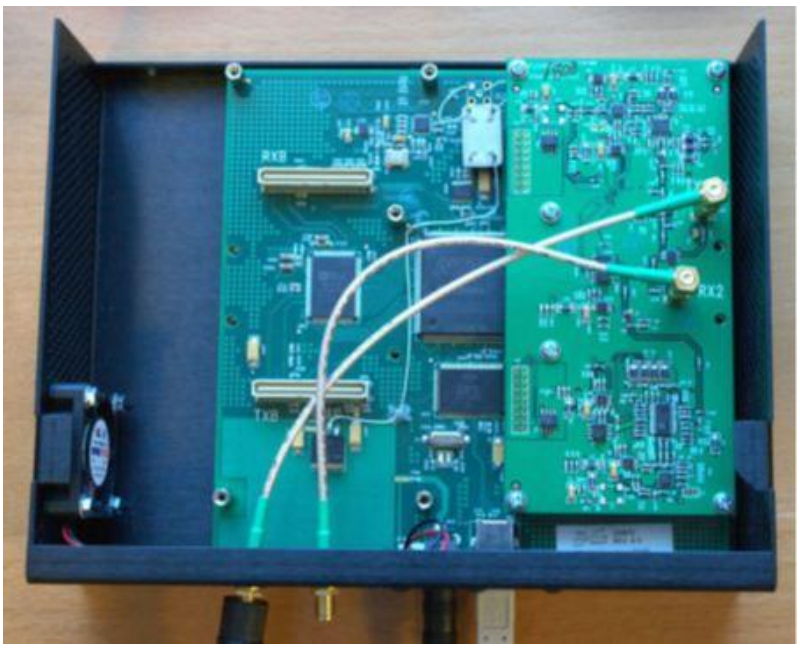

Figure 5: USRP 1 Rev 4.5 with additional 52MHz clock

1.3 Software Requirements

- Linux: a distribution of Unix-based operating system which has terminal that could be used to run OpenBTS code. Ubuntu 11.04 is used.
- GNURadio: is a free and Open-Source software development toolkit that provides signal processing blocks to implement software radios.
- Asterisk: is an open source framework for building communications applications. Asterisk turns an ordinary computer into a communications server.
- OpenBTS: is a software-based GSM access point, allowing standard GSM-compatible mobile phones to be used as SIP endpoints in Voice over IP (VOIP) networks.

1.4 Layers of OpenBTS

The OpenBTS application contains:

- L0 Transceiver.
- L1 TDM functions.
- L1 FEC functions.
- L1 closed loop power and timing controls.
- L2 LAPDm
- L3 radio resource management functions.
- L3 GSM-SIP gateway for mobility management.
- L3 GSM-SIP gateway for call control.
- L3 GSM-SIP gateway for text messaging.

The general design approach of OpenBTS is avoid implementing any function above L3, so at L3 or L4 every sub-protocol of GSM is either terminated locally or translated through a gateway to some other protocol for handling by an external application. Similarly, OpenBTS itself does not contain any speech transcoding functions above the L1 FEC parts.

1.5 Core Modules

Responsible for data traffic from the USRP interface (GNU Radio) to the SIP (Asterisk)

- **Transceiver**: Implements the physical layer and responsible for Modulation and Synchronization.
- **TRX Manager**: Links Transceiver and GSM modules (Transceiver interface).
- **GSM**: Implements the GSM stack layers L1, L2 and L3.
- **Control**: It is located midway between GSM stack and the SIP server. It interprets their commands from GSM stack (Downlink) or from SIP (Uplink) and executes the corresponding functions.
- **SIP**: Implements the SIP protocol and connects to OpenBTS to the Asterisk server.

1.6 Original Data Flow

Typical "upstream" data flow:

1. Radio bursts arrive at the USRP and are digitized. The resulting samples are transferred to the transceiver software in the host CPU in time-tagged USB packets, using the standard USRP interface.
2. The transceiver syncs the USRP timetags with the GSM master clock, isolates each radio burst and demodulates it into a vector of symbol likelihoods (soft symbols). The modulation format is defined in GSM 05. 04 and the burst formats are described in GSM 05. 02 Section 5. 2.
3. The soft symbol vector for each radio burst is time tagged with the GSM frame clock and transferred to the GSM stack via a datagram interface.
4. In the GSM stack, the TDM sublayer (of L1) demultiplexes each burst according to its time tag and sends it to the appropriate logical channel.
5. The logical channel passes each burst into its L1 FEC processor according to the rules of GSM 05. 02.
6. The L1 FEC processor performs the FEC decoding described in GSM 05. 03. The output is a sequence of L2 frames taken by the logical channel and sent up to an L2 processor.
7. The L2 processor runs the LAPDm state machine that performs acknowledgments, retransmissions and segmentation. This state machine is defined implicitly in GMS 04. 06 and given explicitly in ITU-T Q. 921. When an incoming L3 frame has been verified and assembled, it is placed into a queue for consumption by L3. In the course of operation, LAPDm also injects L2 frames into the downstream flow for acknowledgment and retransmission requests.
8. In L3, a dispatch function determines the message protocol and type and calls the appropriate control function to deserialize the message and act on its content, generally producing an L3 response on the downlink. These control functions also interact with the outside world via SIP and other protocols. The above steps are clearly illustrated through Figure 7.

Typical “downstream” data flow:

1. In L3, a control function generates an L3 message, serializes the message into an L3 frame and sends it into the logical channel, which in turn passes it down to L2.
2. The L2 processor breaks the L2 frame into segments, wraps each segment in an L2 frame. Each L2 frame is sent down to L1 according to the LAPDm state machine of GSM 04. 06 and ITU-T Q. 921.LAPDm may also generate additional L2 frames on its own according to its acknowledgment and retransmission rules.
3. The L1 FEC processor encodes each L2 frame according to the rules of (GSM 05. 03) generating four outgoing radio bursts. Each radio burst is time tagged with its intended transmission time according to the TDM rules of GSM 05. 02. These bursts are passed on to the TDM interface.
4. The downstream TDM sub- layer is just a mutex-controlled socket interface where the radio bursts from L1 are reformatted into messages on the transceiver‘s datagram interface.
5. Upon arriving in the transceiver, the outgoing radio bursts are sorted into a priority queue according to transmission time. Bursts are pulled from the queue as they become ready for transmission and the modulated according to GSM 05. 04. The modulated waveform samples are sent to the USRP over the standard time tagged USB interface. If no burst is ready for transmission at a given time the transceiver generates an appropriate filling sequence.
6. In the USRP the samples are converted to an analog waveform for transmission over the radio channel.

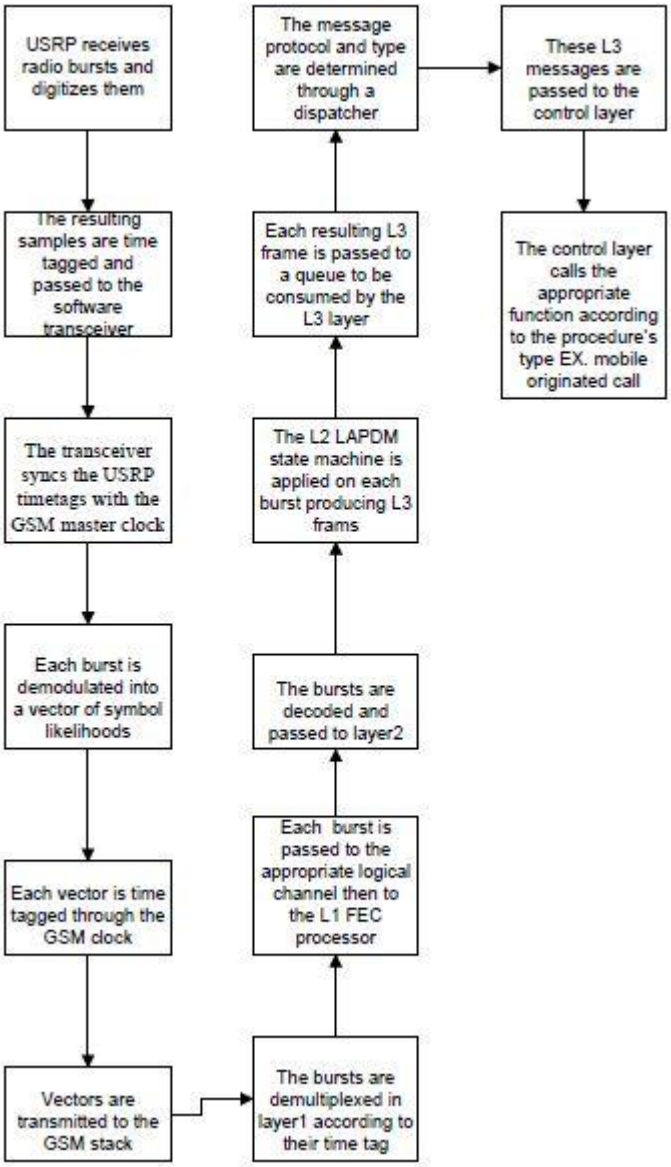

Figure 6: Typical "upstream" data flow in the GSM air interface of the OpenBTS

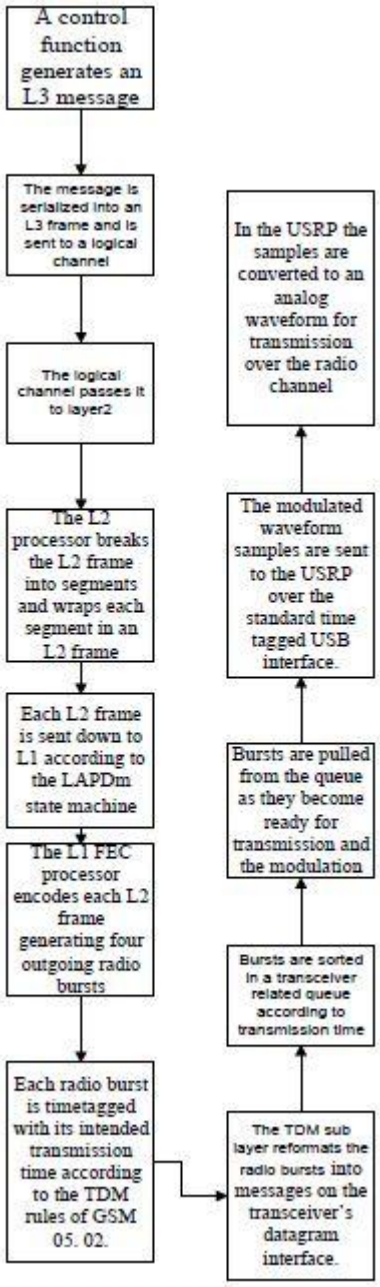

Figure 7: Typical "downstream" data flow in the GSM air interface of the OpenBTS.

1.7 Network Organization

In the simplest network, with a single access point, all of the applications in the suite run inside the access point on the same embedded computer. This is shown in Figure 8.

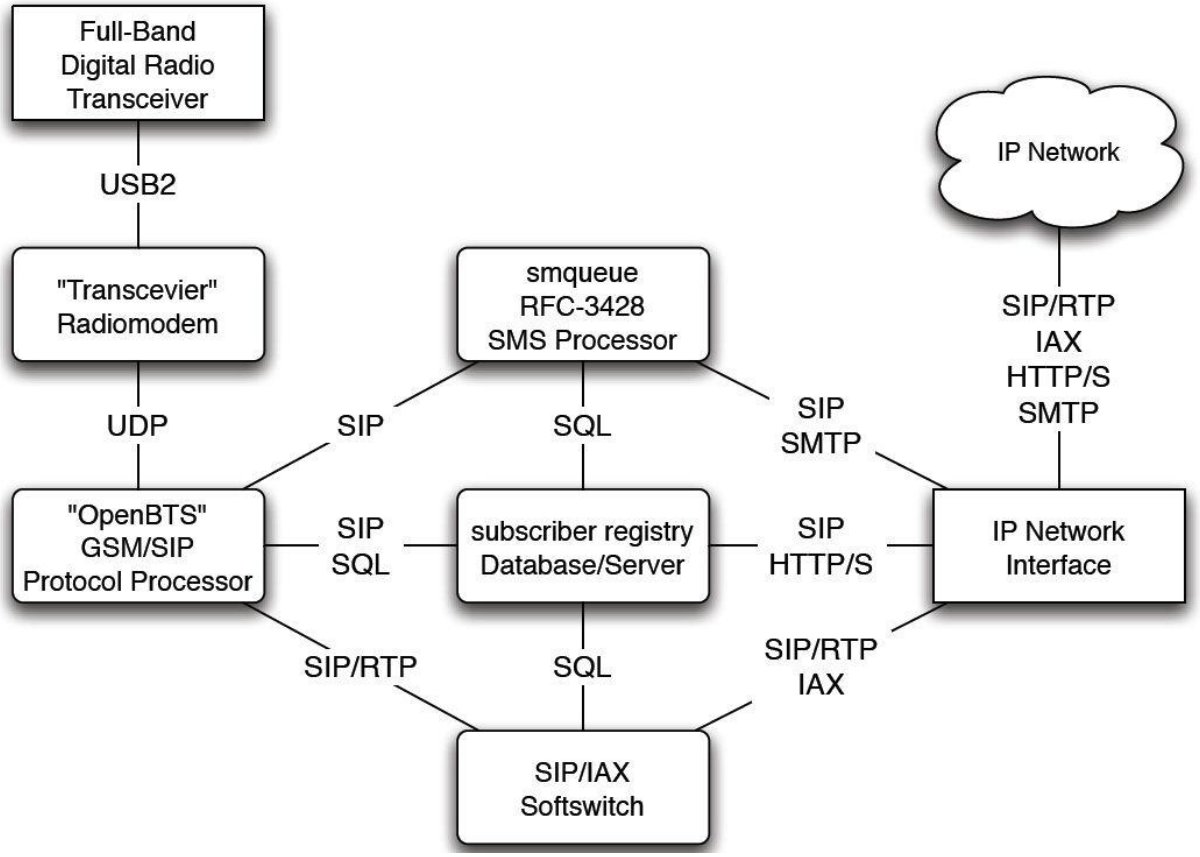

Figure 8: Components of the OpenBTS application suite and their communication channels as installed in each access point. Sharp-cornered boxes are hardware components. Round-cornered boxes are software components.

1.8 GSM Control Repeater

The main idea was to link two USRPs to implement the slave device where the first USRP acts as BTS for mobile sets allocated below it and the second USRP acts as a mobile and communicate with master USRP through an air interface. To implement this structure, A GSM control repeater has to be implemented to repeat GSML3Messages from the first USRP GSM stack flow to the second USRP flow. That can be shown in Figure 4.

A thread will be used to make that link between the different flows at slave. To ensure proper functionality for logical control messages, all logical control channels should be bidirectional channels to be able to forward all type of messages and provide smooth flow without errors and without losing any amount of data. Due to that we listed the unidirectional channels and write functions to make the reverse operation. The original bidirectional channels are lifted without modification. The following list shows logical control channels needed modification with a specification for the needed modification.

Table 1: Needed modification on control channels

Logical Channel	Needed modification
AGCH	Decoder
NCH	Decoder
PCH	Decoder
RACH	Encoder
FCCH	Decoder
SCH	Decoder
BCCH	Decoder
FACCH	----
TCH	----
SDCCH	----

As the implementation of Encoders and Decoders is existed at layer 1, the modified functions and added operations are implemented in two specified files GSML1FEC.cpp and GSML1FEC.h.
Depending on GSM concepts, encoding is applied for messages the moves from BTS to MS (Mobile Station) and decoding will be for the reverse direction. The following graph shows the flow of encoding and decoding and the full encoding and decoding steps.

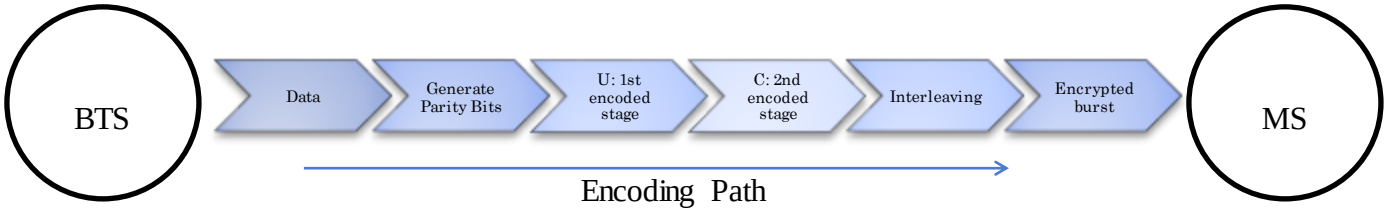

Figure 9: The sequence of encoding data to generate encrypted burst

The complete process to make encoding:

1. ‘d’ data bits. The actual payloads from L2 and the vocoders.
2. ‘p’ parity bits. These are calculated from d.
3. ‘u’ uncoded bits. A concatenation of d, p and inner tail bits.
4. ‘c’ coded bits. These are the convolutionally encoded from u.
5. ‘I’ interleaved bits. These are the output of the interleaver.
6. ‘e’ encrypted bits. These are the channel bits in the radio bursts.

It’s necessary to notify that not all logical channels need the full encoding path to be encoded, e.g. RACH encoding and decoding processes doesn’t need interleaving step.

Chapter 2: Synchronization and freq. correction in OpenBTS code

2.1 Introduction

The goal is to implement a totally new mobile station system in the OpenBTS. As OpenBTS hierarchy has been changed and the concept of master and slave OpenBTSs has been added. In the new system, each slave consists of two USRP kits with two OpenBTS codes running simultaneously. The first USRP will act as BTS and the second USRP will act as a mobile station. As mobile station needs different control channels to make time and frequency synchronization with the master BTS, FCCH and SCH control channels are implemented in OpenBTS code to enable it to act as a mobile station. So, an Uplink path has been added to make the signal flow in OpenBTS bidirectional. To achieve that goal, a detecting code for FCCH and SCH has been added at transceiver (Layer 0) and new decoders are added for both control channels at GSM Stack (Layer 1) then receiving SCH in physical layer level to decode it and obtain included information like NCC, BCC and Frame number.

2.2 ARFCN Mapping

This diagram shows a sample Multiframe with logical channels mapped to time slots and TDMA frames. This is just one possible configuration for an ARFCN.

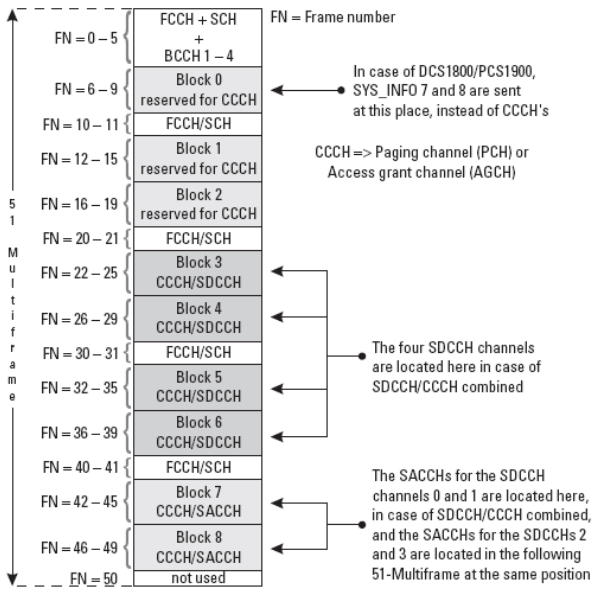

Figure 10: example of the mapping of logical channels

2.3 Frequency correction Channel (FCCH)

2.3.1 Why?

2.3.1.1 What's FCCH?

FCCH is used to synchronize frequency of mobile set with the frequency of BTS. Its burst is equivalent to an unmodulated carrier with a specific frequency offset. The repetition of this burst is called FCCH. The fixed input bits are all zeros, causing the modulator to deliver an unmodulated carrier with an offset of 1,625124 kHz above the nominal carrier frequency. In FCCH, Tail and guard bits are the same as in normal burst. FCCH burst is illustrated in Figure 11.

BTS Info: is the transmitted carrier frequency to mobile station.
MS Info: is the received BCCH carrier frequency from BTS. MS identifies it and synchronizes with it.

2.3.1.2 FCCH burst content

- 142 fixed bits filled with zeros and used for MS frequency synchronization.
- 3 tail bits repeated twice.
- 8.25 bit Guard Period.

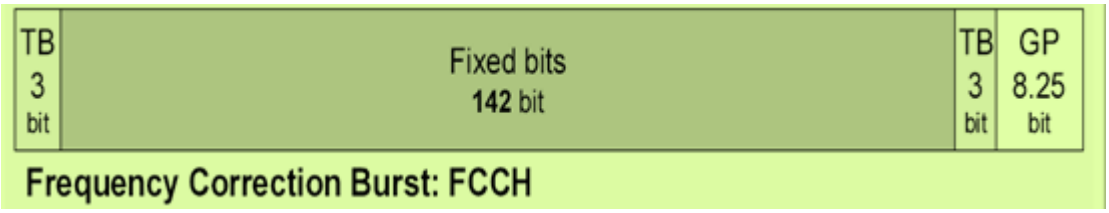

Figure 11: FCCH Burst

2.3.1.3 FCCH Scenario

BTS sends FCCH at time slot 0 in BCCH-TRX bursts. All 142 bits of data are filled with zeros. There're exactly 5 FCCHs per 51 multiframes as in Figure #. FCCH enables MS to identify the frequency of BTS. To make correct synchronization for frequency and time, FCCH should be sent first then SCH. *GSM 05.01, 05.02.*

2.3.1.4 Requirements

It's required to detect FCCH burst and obtain the correct frequency, fix any error in the frequency or any shift happened which may cause wrong reception.

2.4 Synchronization Channel (SCH)

2.4.1 Why?

2.4.1.1 What's SCH?

SCH is implemented to make time synchronization for MS time it contains additionally TDMA frame number and BSIC.

BTS Info: Transmitted information about the TDMA frame structure in a cell (e.g. frame number) and the Base Station Identity Code (BSIC).

MS Info: Synchronizes with the frame structure within a particular cell and ensures that the chosen BTS is a GSM BTS – BSIC. It can only be decoded by an MS if the BTS belongs to a GSM network.

2.4.1.2 SCH burst content

- 64 bits as Training Sequence for initial precise MS time synchronization.
- 39 bits repeated twice contains necessary information to initial MS access (BSIC, TDMA frame number, NB training sequence used in this cell…etc.).

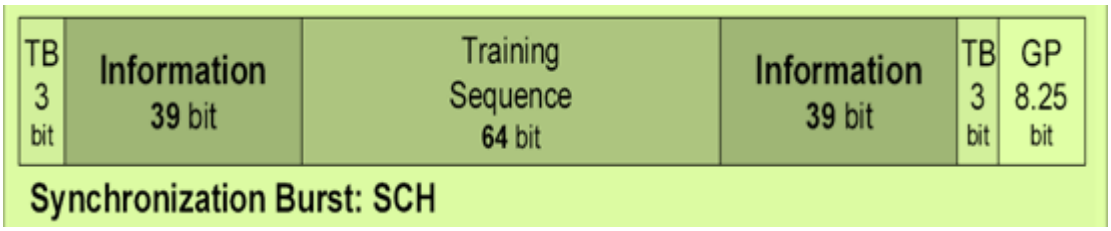

Figure 12: SCH Burst

2.4.1.3 SCH parameters calculation

- The BSIC consists of six bits. Three bits of it represent the PLMN color code with a range from 0 to 7, and the other three represent BS color code with a range from 0 to 7.
- The RFN is 19 bits long and consists of :
  Tl (11 bits) = FN div (26 x 51) Range from 0 to 2,047

  T2 (5 bits) = FN mod (26) Range 0 to 25

  T3 (3 bits) = (FN mod (51) - 1) div(l0) Range 0 to 4

  Where, FN represents the TDMA frame number. The layout of BSIC and RFN exists within the message part of the synchronization sequence and is shown in Figure 12. SCH, FACCH, and BCCH channels cannot be frequency hopped as these channels carry synchronization and system-related information which have exactly known location at mobile station.

- In the BSIC, the channel exists in the downlink direction only and is used for point-to-multipoint communication.

2.5 FCCH & SCH Detection and SCH Decoding

2.5.1 Where?

1. "TRXManager.h","TRXManager.cpp"
2. "BitVector.h"
3. "GSML1FEC.h","GSML1FEC.cpp"

2.5.2 How?

FCCH and SCH Code

2.6 Testing and Verification results

2.6.1 Testing Codes

2.6.1.1 FCCH and SCH detection code

2.6.1.2 SCH decoding program

2.6.2 Code Verification Results

2.6.2.1 FCCH and SCH detection results

2.6.2.2 SCH decoding code results

EMPTY SECTIONS

Chapter 3: GSM CONTROL REPEATER IN OpenBTS

3.1 Introduction

This module shall manage the operation of the repeater through proper initialization and attachment to the master BTS, then it continuously receive L3 messages from both the MS side and the BTS side and react to them .

The main steps in order to fulfill the repeater functions as follow:

- RR Messages: Messages received from the MS side shall be reacted by assigning the required radio resources locally (SDCCH and TCH), in a way similar to the original OpenBTS code. RR messages from the master BTS side shall be used to extract and assign the SDCCH and TCH of the master BTS side. A table shall be held for the correspondence between the resources allocated to both sides.
- MM and CM Messages: Messages originating from one side shall be switched to the other side using the above RR correspondence table.

The above way of operation has the advantage of simplified control, but burdens the link between the master and slave with extra local traffic channels. This could be extended later by modifying the "alerting" CM message to indicate if a call is local to the slave BTS, and modify the TCH assignment to be a local time slot instead of going through the BTS's link.

It is worth noting that the all the slave BTS software can be run on a single PC, with 2 USB outlets connected to 2 USRPs, one for each ARFCN. There are also other modifications that should be made in the dial plan of the master ASTERISK, but this should be simple.

3.2 OpenBTS Connection through GSM Um Repeater

As it's needed to transfer L3message to the Asterisk server which could be accessed only by the Master BTS –no asterisk at slave-, a repeater at Slave is used to forward

message to MS side of the Slave BTS to transmit it to Master BTS through air interface "UM".

In this architecture OpenBTS A operates as a master BTS, with its Asterisk's data base covering the other "slave" BTSs in the service area, and operates at an ARFCN A. OpenBTS B represents one of the slave BTSs, connected to its local MSs at an ARFCN B, and repeating MM and CC signaling messages, as well as some traffic bursts, to OpenBTS A at one or more time slots of ARFCN A.

Control is a hybrid module used as interface layer between SIP module and GSM Stack in traditional OpenBTS.

Most GSM L3messages and VoIP messages terminate at Control module. Everything in this control directory should be in the Control namespace.

Components:

1. Radio Resource: Functions for RR procedures (paging, access grant).

2. Mobility Management: Functions for MM procedures (CM service, location updating).

3. Call Control: Functions for CC (mobile originated, mobile terminated).

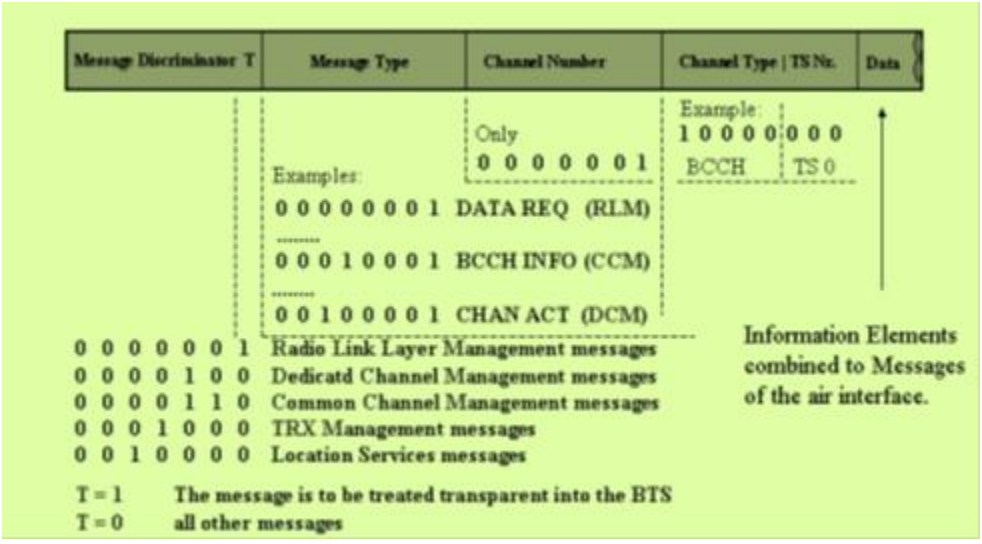

Figure 13: Layer 3 in the Abis interface

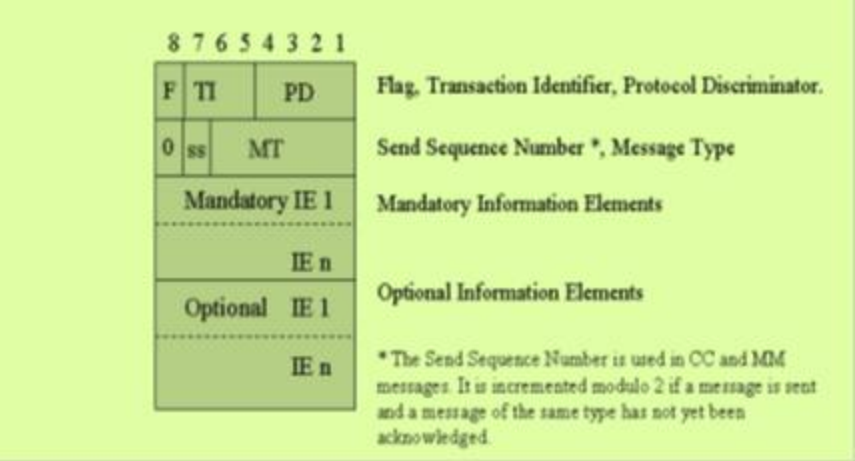

Figure 14: The Construction of a Layer 3 header

By using the function parsel3( ) we can detect

1. PD "Protocol Discriminator" PD ( )
2. MTI "Message Type Identifier" MTI ( )

3.3 GSM Control Repeater (GCR)

3.3.1 Why?

As explained before the needing of GCR arise due to introducing the new hierarchy shown at figure 4. Asterisk server should be accessed to establish correct call. In the modified OpenBTS system, only master BTS has the ability to access the asterisk server. So GCR is needed to forward L3 messages through master BTS to Asterisk server.

3.3.1.1 GCR Content

- Threads.
- User Datagram Protocol Sockets "UDP".
- Container to store received l3frames.

3.3.2 How?

The first task was to find the part in OpenBTS code which forwards L3 messages from GSM Stack and repeats them to the second USRP kit in slave. This USRP kit will act as mobile station and will forward messages to master BTS.

The second task is to implement a class to create UDP sockets which used to make time transfer application. Then use UDP sockets to transfer messages between two processes running on same PC. The first process will run the BTS code and the second one will run MS code at slave part.

The class should consist of two socket-ports one for receiving and other for transmitting. A thread is used to run the receiving socket independent on code algorithm at anytime.

The socket for transmitting the message runs under mutex “lock & unlock” to avoid failure to bind and to prevent two threads write at the same socket at the same time “mutual exclusion”.

The class has some member functions to allow the start of reception L3Frame thread and sending L3Frame operation.

The function that receives the L3Frame from logical channel to make control function on it is found at ControlCommon file named getmessage ( )

3.3.3 Where?

"ControlCommon.h", "ControlCommon.cpp"

3.3.3.1 Added part at ControlCommon.h

```
/*
   i4:The additional headers needed to include
   "thread.h" due to need to ceate "mRepThread"
   "Sockets.h" due to need to use datagram class for UDP
   "GSMCommon.h" , "GSMTransfer.h" due to l3frame and Bitvector
*/

#include "Threads.h"
#include "Sockets.h"
#include "GSMCommon.h"
#include "GSMTransfer.h"

namespace GSM {

class Time;
class L3Message;
class GSMConfig;
class LogicalChannel;
class SDCCHLogicalChannel;
class CCCHLogicalChannel;
class TCHFACCHLogicalChannel;
class L3Cause;
class L3CMServiceRequest;
class L3LocationUpdatingRequest;
class L3IMSIDetachIndication;
class L3PagingResponse;

//i4:added new forward refs used in RepManager

class L3Frame;
enum Primitive;};

/*i4: This the class responsible for repeation process
        & Receiving messages from other side "listening"

*/
```

```
class RepManager {

private:

	// i4:Two user datagram sockets one for sending "mRepSocketW"
	// and other for Receiving "mRepSocket"

	UDPSocket mRepSocket;
	UDPSocket mRepSocketW;

	// i4:"mRepThread" is thread for listening operation to listen
and receive
	// at any time message from other side come

	Thread mRepThread;

	// i4: "mDataSocketLock" To avoid to be error socket in use
and
	//permit to operation write at same port at same time

	Mutex mDataSocketLock;

public:

	// i4:Constructor for class to assign the UDP sockets needed
for operation

	RepManager(int Srcport,const char* Desddress, int DesPort);

	// i4:Function to start thread of listening

	void start();

	// i4:"send" It perform the operation of sending the "l3frame"
to other side

	void send(GSM::L3Frame *l3frame);

	// i4: "RepHandler" perform the action required inside thread
	//(asking if any message rcv )

	void RepHandler();

	friend void* RepLoopAdapter(RepManager*);

};

void* RepLoopAdapter(RepManager *repm);

// i4:Global object of RepManager class to be availabe to use it's
// member functions where i need

extern RepManager *grep;

}	//Control
```

3.3.3.2 Added part at ControlCommon.cpp

```
// i4: Cointainer to store l3frame rcv untill get channel to send it
downstream

queue <L3Frame> l3fifo;

// i4: the initailization of the object of RepManager class
//  it done here not in header files to avoid multiple defines
problem

namespace Control{
RepManager *grep=new RepManager(7778,"127.0.0.1",8888);
                                }
// i4:added part

      Control::grep->send(rcv);

// i4:Contsructor implementation only has the initialization of
// Read and write UDP sockets

RepManager::RepManager(int  Srcportt,const  char*  Desddresst,  int
DesPortt)
      :mRepSocket(Srcportt,Desddresst,DesPortt),mRepSocketW(Srcportt
-1,Desddresst,DesPortt+1)
{
}


// i4:"Start" responsible for starting the thread for listening

void RepManager::start()
{
      mRepThread.start((void*(*)(void*))RepLoopAdapter,this);
      return;
}

/*
 i4: "send" responsible to take pointer to l3frame required to
 send to other side and convert it to char* to be available to send
 using "mRepSocketW.write(ptr)" by collecting the information inside
 the l3frame (data , primitive) and mapping it in char* as following
*/

void RepManager::send(L3Frame *l3frame)
{
      char ptr[l3frame->size()] ;


      for(int i=0;i<(l3frame->size());i++)
            {
                  if(l3frame->bit(i)){ ptr[i]='1';
                   }
                  else          { ptr[i]='0'; }
            }



      switch(l3frame->primitive())
      {
            case ESTABLISH:   ptr[l3frame->size()]='0';
```

```
                break;
          case RELEASE:     ptr[l3frame->size()]='1';
                break;
          case DATA:  ptr[l3frame->size()]='2';
                break;
          case UNIT_DATA:   ptr[l3frame->size()]='3';
                break;
          case ERROR: ptr[l3frame->size()]='4';
                break;
          case HARDRELEASE:ptr[l3frame->size()]='5';
           break;

    }

    ptr[l3frame->size()+1]='\0';
    mDataSocketLock.lock();
    mRepSocketW.write(ptr);
    mDataSocketLock.unlock();

}


/*
 i4: "RepHandler" the function called inside listening thread
 which responsible for receiving char* buffer and parse it to
 extract the (data,primitive) and create l3frame and store them
 in queue until send RACH request and take channel to
 send it downstream
*/


void RepManager::RepHandler()
{
    char buffer[MAX_UDP_LENGTH];
    unsigned ra;
    int msgLen = mRepSocket.read(buffer);
    cout<< buffer<< endl;

    if (msgLen<=0) {
          cout << "read error on REPEATER " << msgLen;
          return;
    }
  // buffer buffer 0's and 1's
    Primitive Primitivee;
    BitVector rcv(buffer);
    BitVector rcv_l3frame;
    rcv_l3frame=rcv.segment(0,msgLen-2);
    BitVector rcv_primitive;
    rcv_primitive=rcv.segment(msgLen-2,1);
    char *primit=rcv_primitive.begin();

    switch(*primit)
    {
          case '0':Primitivee=ESTABLISH;break;
          case '1':Primitivee=RELEASE;break;
          case '2':Primitivee=DATA;break;
          case '3':Primitivee=UNIT_DATA;break;
          case '4':Primitivee=ERROR;break;
          case '5':Primitivee=HARDRELEASE;break;
    }
```

```
        L3Frame rcved(rcv_l3frame,Primitivee);
        if (l3fifo.empty()){
                                        //randam send to l1RACHencoder
                                        ra = rand() % 32 + 160 ;
                                        //RACHL1Encoder rach(ra);
                                        }
        l3fifo.push(rcved);
        buffer[msgLen]='\0';

}

/*
 i4:thread function responsible to call "RepHandler"
*/


void* Control::RepLoopAdapter(RepManager *repm)
{
        while (1) {
                repm->RepHandler();
                pthread_testcancel();
                }
        return NULL;
}



// vim: ts=4 sw=4
```

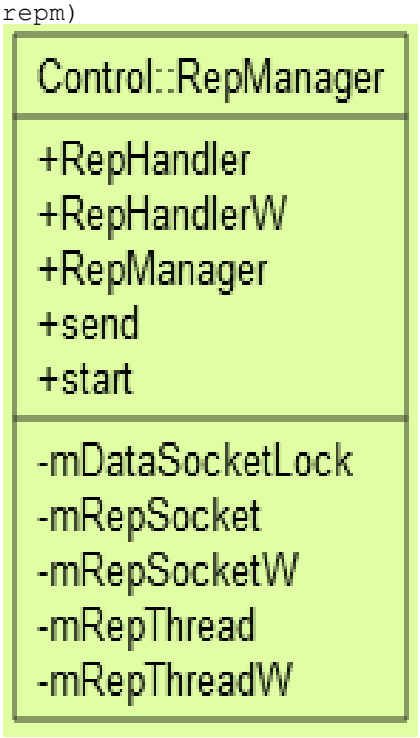

Figure 15: UML Class Diagram "RepManager"

3.4 Testing and Verification results

3.4.1 Testing Codes (rep0.h file)

```
#ifndef REP_H
#define REP_H
#include <stdlib.h>
#include "Sockets.h"
#include "Threads.h"
#include <stdio.h>
#include "GSMTransfer.h"
#include "Vector.h"
#include "BitVector.h"
#include <stdint.h>
#include <cstdlib>
```

```
#include <string>
#include <iostream>
using namespace std;
class RepManager;
queue <L3Frame> l3fifo;
class RepManager {
private:
	UDPSocket mRepSocket;
	Thread mRepThread;
	UDPSocket mRepSocketW;
	Mutex mDataSocketLock;

public:
	RepManager(int Srcport,const char* Desddress, int DesPort);
	void start();
	void send(L3Frame *l3frame);
	void RepHandler();
	friend void* RepLoopAdapter(RepManager*);

};

void* RepLoopAdapter(RepManager *repm);
RepManager *grep=new RepManager(7778,"127.0.0.1",8888);


RepManager::RepManager(int  Srcportt,const  char*  Desddresst,  int
DesPortt)
	:mRepSocket(Srcportt,Desddresst,DesPortt),mRepSocketW(Srcportt
-1,Desddresst,DesPortt+1)
{
}



void RepManager::start()
{
	mRepThread.start((void*(*)(void*))RepLoopAdapter,this);
}


void RepManager::send(L3Frame *l3frame)
{
	char ptr[l3frame->size()] ;


	for(int i=0;i<(l3frame->size());i++)
		{
			if(l3frame->bit(i)){ ptr[i]='1';
			 }
			else		{ ptr[i]='0'; }
		}



	switch(l3frame->primitive())
	{
		case ESTABLISH:	ptr[l3frame->size()]='0';
			break;
		case RELEASE:	ptr[l3frame->size()]='1';
			break;
		case DATA:  ptr[l3frame->size()]='2';
```

```
                break;
            case UNIT_DATA:   ptr[l3frame->size()]='3';
                break;
            case ERROR: ptr[l3frame->size()]='4';
                break;
            case HARDRELEASE:ptr[l3frame->size()]='5';
             break;

      }

      ptr[l3frame->size()+1]='\0';
      mDataSocketLock.lock();
      mRepSocketW.write(ptr);
      mDataSocketLock.unlock();

}



void RepManager::RepHandler()
{
      char buffer[MAX_UDP_LENGTH];
      unsigned ra;
      int msgLen = mRepSocket.read(buffer);
      cout<< buffer<< endl;

      if (msgLen<=0) {
            cout << "read error on REPEATER " << msgLen;
            return;
      }
  // buffer buffer 0's and 1's
      Primitive Primitivee;
      BitVector rcv(buffer);
      BitVector rcv_l3frame;
      rcv_l3frame=rcv.segment(0,msgLen-2);
      BitVector rcv_primitive;
      rcv_primitive=rcv.segment(msgLen-2,1);
      char *primit=rcv_primitive.begin();

      switch(*primit)
      {
            case '0':Primitivee=ESTABLISH;break;
            case '1':Primitivee=RELEASE;break;
            case '2':Primitivee=DATA;break;
            case '3':Primitivee=UNIT_DATA;break;
            case '4':Primitivee=ERROR;break;
            case '5':Primitivee=HARDRELEASE;break;
      }

      L3Frame rcved(rcv_l3frame,Primitivee);
      if (l3fifo.empty()){//rach
                                    ra = rand() % 32 + 160 ;
                                    }
      l3fifo.push(rcved);
      cout<<ra<<endl;
      buffer[msgLen]='\0';

}


void* RepLoopAdapter(RepManager *repm)
```

```
{
      while (1) {
            repm->RepHandler();
            pthread_testcancel();
            }
      return NULL;
}



#endif
```

3.4.2 Testing Codes (SS0.cpp file) 1st process

```
#include "rep0.h"
#include <iostream>
#include <stdlib.h>
#include "Sockets.h"
#include "Threads.h"
#include "Vector.h"
#include "BitVector.h"
#include "GSMTransfer.h"
#include <stdint.h>
#include <cstdlib>
#include <string>
using namespace std;
class RepManager;

int main(){
    grep->start();
      char* ptr;
      BitVector v2("1100000000000000011111111111111110000000000");
      Primitive pri=DATA;
      L3Frame la(v2,pri);
      while (1){
                  grep->send(&la);
                  };

      return 0;
}
```

3.4.3 Testing Codes (SS1.cpp file) 2nd process

```
#include "rep2.h"
#include <iostream>
#include <stdlib.h>
#include "Sockets.h"
#include "Threads.h"
#include "Vector.h"
#include "BitVector.h"
#include "GSMTransfer.h"
#include <stdint.h>
#include <cstdlib>
#include <string>



using namespace std;
class RepManager;
```

```
int main(){
    grep->start();
      BitVector
v2("00000000000000000011000000000000001111111111111110000000000");
      Primitive pri=DATA;
      L3Frame la(v2,pri);
      while (1){
            grep->send(&la);

            };

      return 0;
}
```

3.4.4 Code Verification Results

1st terminal:

```
cd Desktop/....

 g++ - pthread -o 0 ss0.cpp  Sockets.cpp Threads.cpp
Timeval.cpp BitVector.cpp GSMCommon.cpp
```

2nd terminal:

```
cd Desktop/....

g++ -pthread -o 1 ss1.cpp  Sockets.cpp Threads.cpp
Timeval.cpp BitVector.cpp GSMCommon.cpp
```

O/P 1st and 2nd terminals:

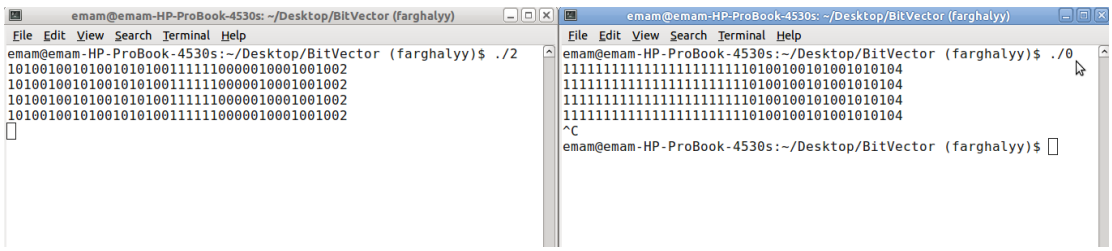

Figure 16: The GCR output in terminal

Chapter 4: L1 Encoders and Decoders

4.1 Random Access Channel In OpenBTS Code

4.1.1 Introduction

In a way to produce architecture like the one in GSM, the BTS must be under controlled by the master BTS. It can be done by using the Mobile Station (MS) concept in the slave BTS.

The slave BTS can call the master BTS by requesting a channel from the Master BTS. The channel request must be sent by the slave to the master on the RACH, then master assigns a new channel to the slave - if available - and the master informs slave using Access Grant.

4.1.2 How?

It can be done by two directions:

- It is needed to reverse all functions of the RACH Decoder.
- It is needed to find the Encoder which takes the same way of the RACH Encoder.

Referring to GSM Standard 05.03 (see below), It is noticed that RACH takes the same way in encoding as Synchronous channel (SCH).

The encoding process takes the following steps:

- Encryption process.
  - Generation of Data bits.
  - Generation of Parity bits.
  - Generation of Data's Tail bits.
  - Convolution Code.
- Adding Synchronous bits.
- Adding Tail bits.
- Transmission of the burst.

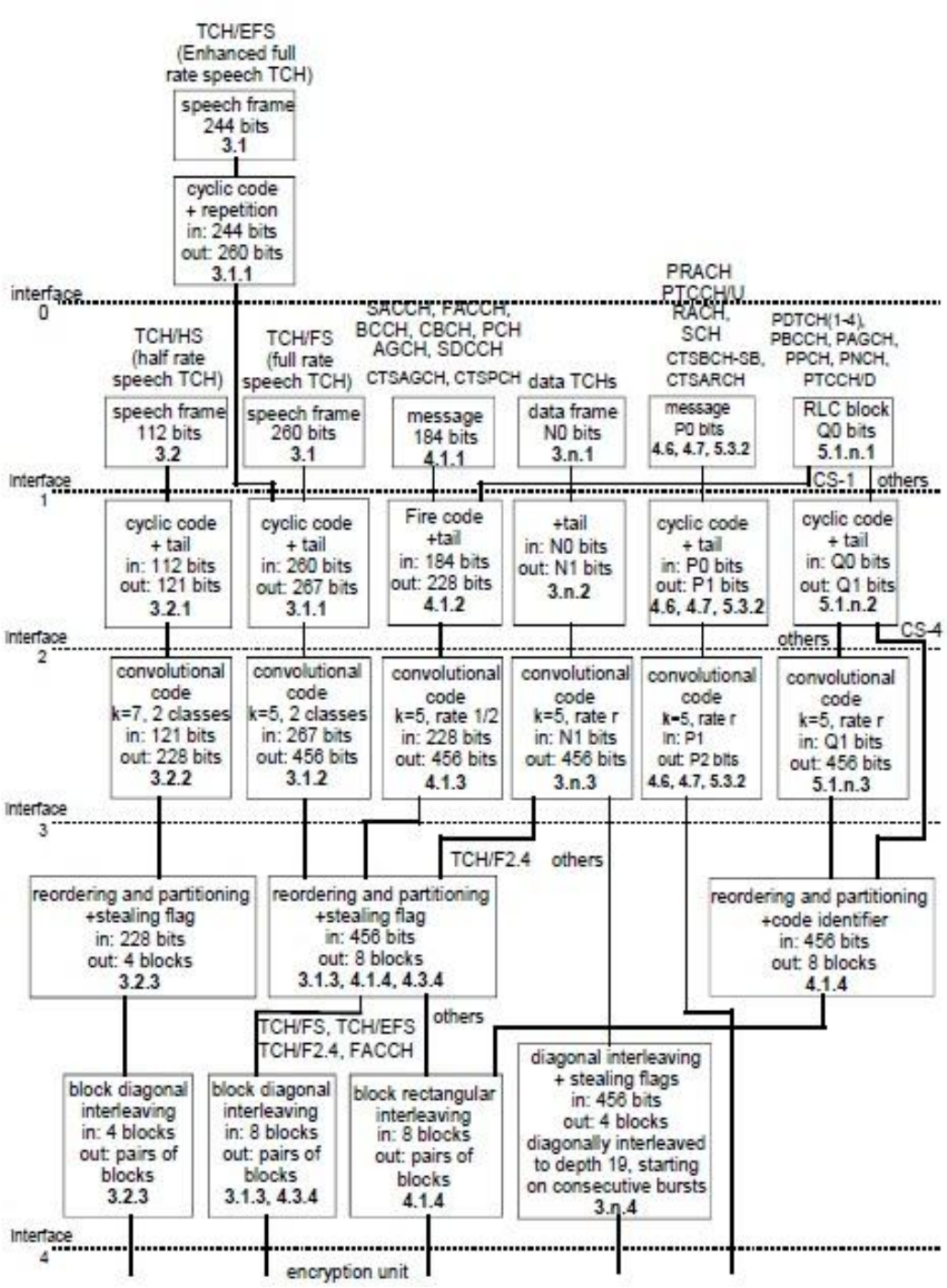

Figure 17: Channel Coding and Interleaving Organization

4.1.3 Access burst

Table 2: Access burst Organization

Field Name	# Bits	Content
Tail	8	0011 1010
Sync. sequence	41	01001011011111111001100110101010001111000
RACH data	36	(8 Data + 6 Parity +4Tail) * 2 Convolution code
Tail	3	000
Guard	68.25	----------

To ensure that an access burst arrives at the BTS during the proper time period the number of bits for the access burst was set to only 88 bits with a larger guard period of 68.2 bits. The maximum distance between BTS and MS is, with this timing, about 35 km.

The normal burst would not fit into the receiver window if the unknown propagation delay was greater than zero. That is the reason why the normal burst is used only after the distance of the MS from the BTS is determined, and the MS is able to adjust its transmission accordingly.

4.1.4 Access burst contents

Referring to GSM 05.03, the burst caring the random access uplink message has a different structure. It contains:

- 8 information bits d(0), d(1), ..., d(7).
- Six parity bits p(0), p(1) ,... , p(5) are defined in such a way that the binary polynomial:
  d(0)D13 +...+ d(7)D6 + p(0)D5 +...+ p(5), when divided by:
  D6 + D5 + D3 + D2 + D + 1 yields a remainder equal to
  D5 + D4 + D3 + D2 + D + 1.
- The six bits of the BSIC, {B(0),B(1),...,B(5)}, of the BS to which the Random Access is intended, are added bitwise modulo 2 to the six parity bits, {p(0), p(1) ,... ,p(5) }. This results in six colour bits, C(0) to C(5) defined as:
  C(k) = b(k) + p(k) (k = 0 to 5) where:
  b(0) = MSB of PLMN color code, b(5) = LSB of BS color code.
- This defines {u(0),u(1),..., u(17)} by:

  u(k) = d(k) for k = 0,1,...,7

u(k) = C(k-8) for k = 8,9,...,13

u(k) = 0 for k = 14,15,16,17 (tail bits)

- The bits {e(0),e(1),..., e(35)} are obtained by the same convolutional code of rate 1/2 as for TCH/FS, defined by the polynomials:

G0 = 1 + D3 + D4

G1 = 1 + D + D3 + D4

and with:

e(2k) = u(k) + u(k-3) + u(k-4)

e(2k+1) = u(k) + u(k-1) + u(k-3) + u(k-4) for k = 0,...,17 ; u(k) = 0 for k < 0

4.1.5 Access burst Data (8 bit)

Referring to GSM standard 04.08, request message does not follow the basic format. The message is only one octet long, coded as shown in figure below.

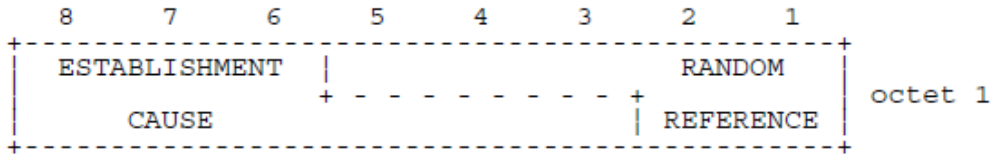

Figure 18: Channel request message content

4.1.6 Access burst Encoding

4.1.6.1 Where?

"GSML1FEC.h", "GSML1FEC.cpp"

4.1.6.2 ESTABLISHMENT CAUSE (octet 1)

This information field indicates the reason for requesting the establishment of a connection. This field has a variable length (from 3 bits up to 6 bits).

4.1.6.3 RANDOM REFERENCE (octet 1)

This is an unformatted field with variable length (from 5 bits down to 2 bits).
The Channel Request message is coded as follows:
(Random Reference field is filled with 'x'). For more details, see 04.08.

MS codes bits 8 1	According to Establishment cause:
101xxxxx	Emergency call
110xxxxx	Call re-establishment; TCH/F was in use, or TCH/H was in use but the network does not set NECI bit to 1
011010xx	Call re-establishment; TCH/H was in use and the network sets NECI bit to 1

Figure 19: Channel request data part

Chapter 5: Conclusion

Please follow the guidelines provided in this document as closely as possible!

References

Below are some examples of citations following the IEEE standard:

A. B. Author, "The paper title," *Journal name*, vol. 50, no. 12, pp. 2702–2712, Dec. 2002.

C. D. Another, *Book title*, 2nd ed. New York: Wiley, 1998.

Appendix

Insert your appendix (appendices) here. The appendix should be used to provide any useful information, material, or derivations that is relevant to the project but should not be written in the report body in order not to interrupt the flow of the text.