ARADA SYSTEMS

LocoMate Programmer's Guide

Version 2.3

August 2012

Table of Contents

Preface

Locomate Programmer's Guide helps you to use the APIs that interface with the WSMP layer in the WAVE stack.

Audience

This manual is intended for programmers who want to build applications on the WAVE stack.

Related Information

For API-related information, refer to *Locomate Programmer's Guide*.

Contact Us

If you purchased a service program from Arada, you can get help at any time by calling Arada:

+91-80-22107174/7275

Arada would like to know if you found the document useful. If you have any feedback or comments on this document, send us email at:

support@aradasystems.com

Please mention the software, version of the software, and the title of the document in your message.

You can also send us your comments by mail at:

Arada Systems, Inc
4633 Old Ironsides Drive, Suite 415
Santa Clara, CA 95054 (USA)

Conventions

The following conventions are used in the document to help you identify special terms.

Convention	Usage	Example
Bold	The following screen elements: Button List Drop-down menu	Click **OK**.
Italic	Book titles and emphasis	Refer to *Concepts Guide* for more information.
`monospace`	Code samples and commands	To run the installer, enter the following command: `arada# show ip`
`< >`	Mandatory parameters	`arada# show card <unit number>`
`[ ]`	Optional parameters	`arada# configure ip address <ip- address> [subnet mask]`
`|`	Mutually exclusive choices in a command or code	`arada#reboot 1|2|3`

Standards Compliance

Our software stack has been developed to comply with latest standards of

IEEE 802.11p D9.0,

IEEE P1609.4 D9, Aug 2010

IEEE P1609.3 D9, Aug 2010

IEEE P1609.2 D9.3

Abbreviations and Definitions

The following abbreviations are used in the document

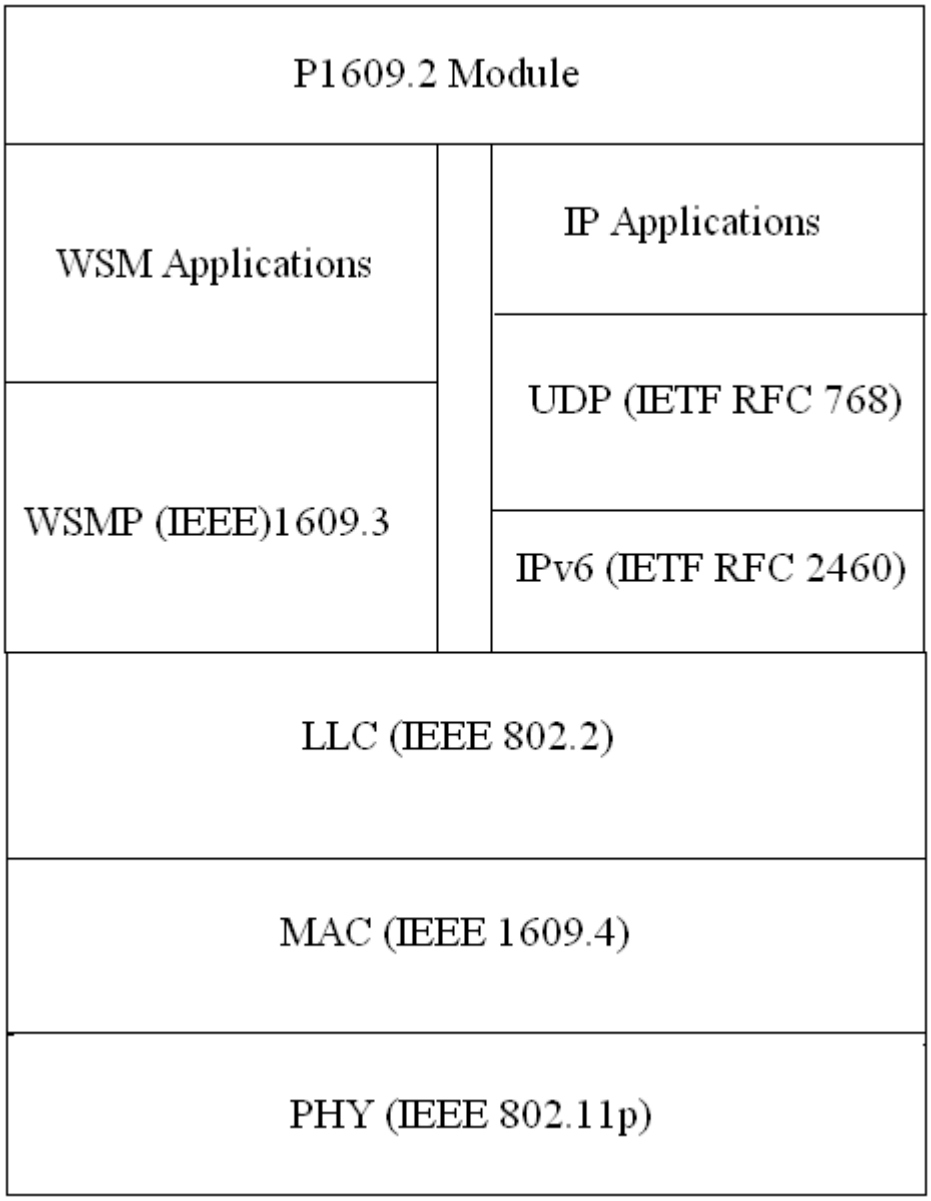
Acronym	Definition
PSID	Provider Service Identifier
CLI	Command Line Interface
BSS	Base Subscriber Station
GPS	Global Positioning System
IP	Internet Protocol
MAC	Medium Access Control
Mb	Megabits
NMEA	National Marine Electronics Association
OBU	On Board Unit
UDP	User Datagram Protocol
WAVE	Wireless Access in Vehicular Environments
WBSS	WAVE Base Subscriber Station
WME	WAVE Management Entity
WSMP	Wave Short Message Protocol
WSS	Wave Security Services

1. API Reference

1.1 WAVE API

This set of API provides a consistent interface for applications that needs to access the services provided by the WME. Figure 1.1 shows the API interfacing with the WAVE stack:

Figure 1.1 API Interfacing With the WAVE Stack



SAP to API Mapping

The SAPs are defined as in IEEE P1609.3/D9, August 2010 & IEEE P1609.4/D09, August 2010 specifications.

Table 1.1 SAP to API Mapping

SAP	Libwave API
P1609.3 SAPs	
WME-ProviderService.request/ WME-ProviderService .confirm	int registerProvider(int pid,WMEApplicationRequest *appreq) int removeProvider(int pid,WMEApplicationRequest *appreq)
WME-UserService.request/ WME-UserService.confirm	int registerUser(int pid,UWMEApplicationRequest *appreq); int removeUser(int pid, WMEApplicationRequest *appreq);
WME-Notification.indication API	void registerWMENotifIndication (void (*wmeNotifIndCallBack) (WMENotificationIndication *));
WME-WRSSREQUEST.request	int getWRSSReport(int pid, WMEWRSSRequest *req);
WME-WRSSREPORT.indication	void registerWRSSIndication (void (*wrssIndCallBack) (WMEWRSSRequestIndication *));
WSM-WaveShortMessage.request/ WSM-WaveShortMessage.confirm	int txWSMPacket(int pid, WSMRequest *req);
WSM-WaveShortMessage.indication	int rxWSMPacket(int pid, WSMIndication *ind); void registerWSMIndication (void (*wsmIndCallBack)(WSMIndication *));
WSM-TimingAdvertismentService.request/ WSM-TimingAdvertismentService.confirm	int transmitTA(WMETARequest *);
P1609.4 SAPs	
MLME-CANCELTX.request	int cancelTX(int pid, WMECancelTxRequest *req);
P1609.2 SAPs	
WSS-SM.request (Wave Security Services–Signed Message)	EscMsg_Sign_Msg_Req1_t; EscMsg_Sign_Msg_Req2_t; EscMsg_Sign_Msg_Req3_t;
WSS-SM.confirm (Wave Security Services–Signed Message)	EscMsg_Sign_Msg_Res_t;
WSS-SMV.request (Wave Security Services–Signed	EscMsg_Verify_Msg_Req1_t; EscMsg_Verify_Msg_Req2_t;

Message Validation)	
WSS-SMV.confirm (Wave Security Services–Signed Message Validation)	EscMsg_Verify_Msg_Res1_t; EscMsg_Verify_Msg_Res2_t;
WSS-EM.request (Wave Security Services–Encrypted Message)	EscMsg_Enc_Msg_Req1_t; EscMsg_Enc_Msg_Req2_t;
WSS-EM.confirm (Wave Security Services–Encrypted Message)	EscMsg_Enc_Msg_Res1_t; EscMsg_Enc_Msg_Res2_t;

1.2 Data Structures

Based on the IEEE 1609.3 WME specifications, various requests and responses between the WME and Application have been encapsulated as structs in a header file called wave.h.

You can find wave.h in sample applications(section 2.1).

These data structures shall be used by the applications to exchange data with the WME and therefore, each application should maintain their local copies of the relevant structs. As an example, if an application needs to obtain a WRSS report, it should populate a variable of type WMEWRSSRequest and call a function called getWRSSReport which accepts pointer to a WMEWRSSRequest variable. Applications may, similarly, receive indications from WME through a structure called WMENotificationIndication.

1.3 Functions

The functions provided by the WAVE API allow an application to perform various management tasks such as provider/user registration/un-registration, WBSS management and WSM Packet Tx/Rx. These functions as such, are divided into two categories:

1.3.1 Initialization Functions

The following functions enable an application to open and initialize the WAVE driver as well as register the callback functions for receiving the Indications.

```
int setWMEApplRegNotifParams (WMEApplicationRequest *req);
void registerWMENotifIndication ( void
       (*wmeNotifIndCallBack)(WMENotificationIndication *) );
void registerWSMIndication (void
      (*wsmIndCallBack)(WSMIndication *));
void registerWRSSIndication (void
(*wrssIndCallBack)(WMEWRSSRequestIndication *));
void registertsfIndication(void
               (*tsfIndCallBack)(TSFTimer *));
```

Note:

The functions with **register** in their name, are used to initialize the call back functions

for WME to Application SAPs.

1.3.2 WME Interface Functions

These functions are used by the application interface functions to send requests to the WME and receive the indications. Applications requiring a direct access to the driver may also use these functions.

```
static void (*receiveWMEIndication)(WMENotificationIndication *);

int generateWMEApplResponse(WMEApplicationResponse *req);

int generateWMEApplDelRequest();

int generateWMEApplRegRequest(void  *req);

int generateWMETARequest(WMETARequest *req);

int generateWMEWRSSRequest(WMEWRSSRequest *req);

int generateWMECancelTxRequest(WMECancelTxRequest *req);
```

1.4 API Details

A library called libwave has been implemented for the WAVE API. The library demonstrates the abstractions provided by the API of the various WME management functions. The WSMP demo application has been designed to make use of all of the features provided by the WAVE library. As such, the wsmp demo application now acts as a data entry tool for various WME requests and all the driver specific requests has been moved to the libwave. The procedure for using the libwave for WME services by an application can be summarized in the following steps.

1. Register the callback functions with the WAVE library.
2. Invoke the driver.
3. Formulate the requests and call the appropriate function.
4. Take action based on the response.
5. Un-register when done.

1.4.1 Invoking the WAVE Device

The function call `int invokeWAVEDevice(int type, int blockflag)` instructs the `libwave` to open a connection to a wave device either on the local machine or on a remote machine. The type argument can be either of `WAVEDEVICE_LOCAL` or `WAVEDEVICE_REMOTE`. When the type is `WAVEDEVICE_LOCAL` the second argument, blockflag, specfies whether the read/write operations are blocking or not. For `type = WAVEDEVICE_REMOTE`, before calling `invokeWAVEDevice(int type, int blockflag)` make a call to API Details `int setRemoteDeviceIP(char *ipaddr)` to set the IP address of the remote wave device. The function return 0 on success.

Note:

Invoke the wave device before issuing any request to the wave device.

1.4.2 Registering Call Back Functions for Receiving WME Notifications

The library provides a function called registerWMENotifIndication which accepts a function pointer as its only argument.

```
void registerWMENotifIndication( void (*wmeNotifIndCallBack)
(WMENotificationIndication *));
```

Assuming that the application has a function called `receiveWME_NotifIndication` defined as

```
void receiveWME_NotifIndication( WMENotificationIndication
*wmeindication);
```

The application may call the register function as shown
`registerWMENotifIndication(receiveWME_NotifIndication);`

Any WME notifications subsequent to the above call will result in the calling of the `receiveWMENotif` function of the application.

1.4.3 Registering an Application as a Provider or a User

The following functions enable the application to register a provider or a user respectively:

```
int registerProvider(int pid,WMEApplicationRequest *appreq);

int registerUser(int pid, WMEApplicationRequest *appreq );
```

The functions return the status of the call to the application. The second argument is a pointer to a variable of type `WMEApplicationRequest *` which contains request specific parameters as defined in the WME documents. The details of these structures can be found in the file wave.h.

When the device has been invoked with `type = WAVEDEVICE_REMOTE` call the function `int setWMEApplRegNotifParams(WMEApplicationRequest *req)` to notify libwave of the notification IP address and notification port where WME notifications should be recived.

```
typedef struct {
        u_int32_t psid;
        char  acf[OCTET_MAX_LENGTH];
        u_int8_t priority;
        u_int8_t requestType;
        u_int8_t macaddr[IEEE80211_ADDR_LEN];
        u_int8_t repeats;
        u_int8_t persistence;
        u_int8_t channel;
        u_int16_t localserviceid;
        u_int8_t action;
        u_int8_t dstmac[6];
        u_int8_t wsatype;
        u_int8_t channelaccess;
        u_int8_t repeatrate;
        u_int8_t ipservice;
```

```
        struct in6_addr ipv6addr;
        u_int16_t serviceport;
        u_int8_t rcpithres;
        u_int8_t wsacountthres;
        u_int8_t userreqtype;
        struct ieee80211_scan_ssid ssid;
        u_int8_t linkquality;
        u_int8_t schaccess;
        u_int16_t schextaccess;
        u_int16_t notif_port;
} WMEApplicationRequest;
```

Only the *notif_ipaddr* and *notif_port* fields need to be populated before calling *setWMEApplRegNotifParams(...).* The call to *registerProvider(...)* or *registerUser(...)* must be made after the call to *setWMEApplRegNotifParams(...)*. Similarly, before registering set the *ipaddr* and *serviceport USTEntry/PSTEntry* fields to recive WSMIndications from the remote device.

Note:

Your application's call back functions will be run as and when a WME-Notification or WSM-Indication arrives.

1.4.4 Removing a Provider or a User

The functions :

```
int removeProvider(intpid, WMEApplicationRequest *appreq);

int removeUser(int pid, WMEApplicationRequest *appreq);
```

allow the application to remove a registered provider or a registered user respectively. The functions return the status of the call to the application. The second argument is a pointer to a variable of type WMEApplicationRequest which contains request specific parameters as defined in the WME documents. The details of these structures can be found in the file wave.h.

1.4.5 Getting a WRSS Report

The function :

```
int getWRSSReport(int pid, WMEWRSSRequest *req);
```

obtains the WRSS report from the WME. Before calling this function the *registerWMENotifIndication* should have been called by the application. The functions return the status of the call to the application. The second argument is a pointer to a variable of type WMEWRSSRequest which contains request specific parameters as defined in the old WME draft. The details of this structure can be found in the file wave.h.

1.4.6 Transmitting a WSM Packet

There are two options available to the application for transmitting a WSM packet. The function

```
int txWSMPacket(int pid, WSMRequest *req);
```

allows the application to pack WSMPacket variable and pass a pointer to it for transmission.

1.4.7 Receiving a WSM Packet

The function

```
int rxWSMPacket(int pid, WSMIndication *ind);
```

returns a packet in WSMP format in the structure WSMPacket.

This function must be continously polled when the wave device was invoked with *type = WAVEDEVICE_LOCAL*. For type = *WAVEDEVICE_REMOTE* the call back function registered with a call to *registerWSMIndication(...)* is called by libwave whenever it receives a WSM.indication from the WME.

1.4.8 Receiving GPS data using GPSC

GPSC is used for getting GPS data in applications . It is used for getting GPS data in applications.

```
typedef struct  {
        char    nmea[GPS_STRSIZE];
        double          actual_time;            //no. of sec from jan 1, 1970 00:00:00
        double          time;
        double          local_tod;
        uint64_t        local_tsf;
        double          latitude;
        char            latdir;
        double          longitude;
        char            longdir;
        double          altitude;
        char            altunit;
        double          course;
        double          speed;
        double          climb;
        double          tee;
        double          hee;
        double          vee;
        double          cee;
        double          see;
        double          clee;
        double          hdop;
        double          vdop;
        uint8_t         numsats;
        uint8_t         fix;
```

```
double      tow;
int         date;
} GPSData;
```

For using gpsc it is necessary to include gpsc_probe.h into application and statically compile the gpsc_probe.c file with application.

API'S related to GPSC

Connection-:

int gpsc_connect();

It creates a socket and returnes the file descriptor and also keeps one file descriptor with it in global variable gpscsockfd

.

Termination-:

int gpsc_close_sock();

It terminates the connection according to the file descriptor stored in the global variable.

Getting GPS data-:

For getting GPS data in application it is necessary to write one character to the socket with the write API

.

write(<file descriptor>,&ch,size of character);

Then only the application can get the whole GPS structure with the read API .

read(<file descriptor>,gpsdat,sizeof(GPSData))

1.4.9 Transmitting a TA(TimingAdvertisement)Packet

```
int transmitTA(WMETARequest *tareq);
```

Function requests a Timing Advertisement frame be transmitted.

1.5 P1609.2 WaveSecurityServices API's

For WSS related operations, Socket calls act as a interface between Application/libWAVE and security module. All the ASM request/response packet transmitted/received through socket calls. Response packet contains signed and/or encoded data as per the request.

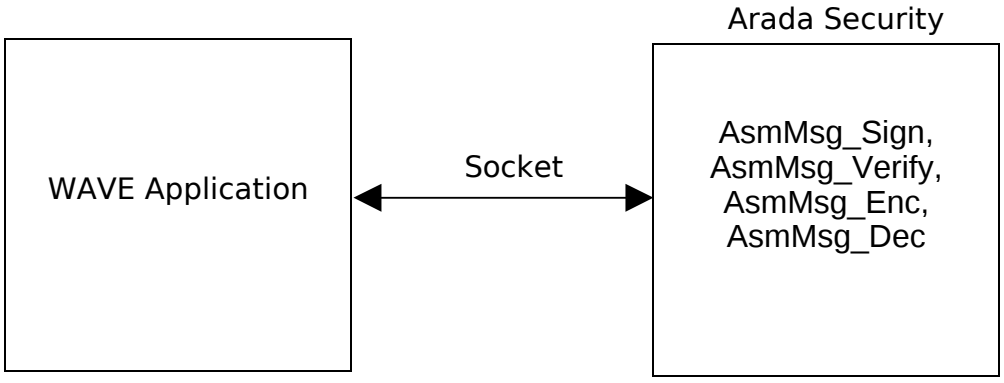


Figure1.5.1. Security Library Overview

AsmMsg_Sign: Generate a signature for the passed application data and return the entire OTA (Over The Air) signed message (incl. 1609 header, payload and signature).

AsmMsg_Verify: Verify the passed signed message and return the result.

AsmMsg_Enc: Encrypt the passed application data and return the entire OTA message (incl. 1609 header and encrypted payload).

AsmMsg_Dec: Decrypt the passed encrypted message and return the entire

OTA message (incl. 1609 header and decrypted payload).

SAP	***Message structure***
WSS-SM.request (Wave Security Services–Signed Message)	typedef struct{ UINT8 command; UINT32 handle; Time64 actual_time; UINT8 signed_message_type; UINT32 application_data_length; UINT8 application_data[1]; } EscMsg_Sign_Msg_Req1_t; typedef struct{ UINT8 psid [1]; } EscMsg_Sign_Msg_Req2_t; typedef struct{ BOOL use_generation_time; BOOL use_expiry_time; BOOL use_generation_location; Time64 expiry_time; SINT32 generation_location_latitude; SINT32 generation_location_longitude; UINT8 generation_location_elevation[2]; UINT8 signer_identifier_type; SINT32 signer_info_cert_chain_length; } EscMsg_Sign_Msg_Req3_t;
WSS-SM.confirm (Wave Security Services–Signed Message)	typedef struct{ UINT8 command; UINT32 handle; UINT32 signed_message_length; UINT8 signed_message_data[]; } EscMsg_Sign_Msg_Res_t;

WSS-SMV.request (Wave Security Services–Signed Message Validation)	typedef struct{ UINT8 command; UINT32 handle; Time64 actual_time; UINT32 signed_message_length; UINT8 signed_message_data[]; }EscMsg_Verify_Msg_Req1_t; typedef struct{ BOOL perform_cryptographic_verification; BOOL detect_replay; BOOL require_generation_time; UINT64 message_validity_period; Time64 WithConfidence generation_time; UINT32 generation_time_confidence_multiplier; BOOL use_expiry_time; Time64 expiry_time; BOOL require_generation_location; UINT32 message_validity_distance; ThreeDLocationAndConfidence generation_location; UINT32 generation_location_horizontal_confidence_multiplier; UINT32 generation_location_elevation_confidence_multiplier; SINT32 local_location_latitude; SINT32 local_location_longitude; UINT32 overdue_CRL_tolerance; }EscMsg_Verify_Msg_Req2_t;

WSS-SMV.confirm (Wave Security Services–Signed Message Validation)	typedef struct { UINT8 command; UINT32 handle; } EscMsg_Verify_Msg_Res1_t; typedef struct { SINT32 CRL_staleness; } EscMsg_Verify_Msg_Res2_t;
WSS-EM.request (Wave Security Services–Encrypted Message)	typedef struct { UINT8 command; UINT32 handle; Time64 actual_time; UINT32 application_data_length; UINT8 application_data_type; UINT8 application_data[]; } EscMsg_Enc_Msg_Req1_t; typedef struct { UINT16 recipient_number; UINT8 recipient_certs[]; } EscMsg_Enc_Msg_Req2_t;
WSS-EM.confirm (Wave Security Services–Encrypted Message)	typedef struct { UINT8 command; UINT32 handle; UINT32 encrypted_message_length; UINT8 encrypted_message_data[]; } EscMsg_Enc_Msg_Res1_t; typedef struct { UINT16 failed_recipient_number; UINT8 failed_recipient_certs[]; } EscMsg_Enc_Msg_Res2_t;

WSS-SMDE.request
 	
(Wave Security Services-Secured Message Data Extraction)
	typedef struct{
	UINT8 command;
	UINT32 handle;
	Time64 actual_time;
	UINT32 encrypted_message_length;
	UINT8 encrypted_message_data[1];
	} AsmMsg_Dec_Msg_Req1_t;
	typedef struct
	{
	BOOL allow_no_longer_valid_cert;
	} AsmMsg_Dec_Msg_Req2_t;

WSS-SMDE.confirm (Wave Security Services-SEcured Message Data Extraction)	typedef struct{ UINT8 command; UINT32 handle; UINT8 content_type; UINT32 application_data_length; UINT8 application_data[1]; } AsmMsg_Dec_Msg_Res_t;

Table1.5.1. SAP-Message Structure Mapping

API's to create security request/response packets

AsmMsg_Sign, AsmMsg_Verify, AsmMsg_Enc, AsmMsg_Dec request packets are constructed by using following functions

```
void msg_create_sign_msg(UINT8* buff, UINT8* buf,int* buffSize,int);

void msg_create_verify_msg(const UINT8* signBuff, UINT8* buff, int*
                    buffSize);

void msg_create_enc_msg(UINT8* buff, UINT8* buf,int* buffSize,int);

void msg_create_dec_msg(const UINT8* encBuff, UINT8* buff, int* buffSize);
```

Response to AsmMsg_Dec request should be decoded by using this API

```
void msg_decode_dec_msg(UINT8* app_data,UINT8* decBuff,int* buffSize);
```

TCP connection related API's

```
int tcp_init(const char* addr, const int port,int);
```
// port should be 50000

```
int tcp_send(const void* buff, const int buffSize,int);
```
//Used to send all EscMsg_* Requests.

```
int tcp_recv(void* buff, const int buffSize,int);
```
// Used to recv responses to EscMsg_* requests.

```
void tcp_close(int);
```

1.6 Supporting Host-Target Environment

1.6.1 Invoking a WAVE Device

```
int invokeWAVEDevice(int type, int blockflag);
```

Where type is one of: WAVEDEVICE_LOCAL and WAVEDEVICE_REMOTE, blockflag is ignored for type=WAVEDEVICE_REMOTE.

Note: If the type is WAVEDEVICE_REMOTE, make a call to `int setRemoteDeviceIP (char *ipaddr)` to set the IP address of the TARGET wave device.

1.6.2 Registering call back to receive WSMIndications from TARGET

```
void registerWSMIndication (void(*wsmIndCallBack)(WSMIndication *));
```

When the libwave receives a `WSMIndication` it calls `wsmIndCallBack` on the HOST's wave application with a pointer to the received indication.

Registeringan Application on the TARGET

```
```

The procedure to register an application on the TARGET involves the following steps:

1. Filling the appropriate WMEApplicationRequest entries, including the notif_ipaddr and notif_port.
2. Filling the PSTEntry / USTEntry, including the ipaddr and serviceport. For a PSTEntry set the contents as follows:

```
pstentry.contents |= WAVE_IPV6ADDR
```

Note:

The current version of libwave and 11p driver only deal with IPV4 addresses.

3. Set the notification receive address and port in the libwave by calling

```
int setWMEApplRegNotifParams(WMEApplicationRequest *req);
```

After filling the request parameters and pst/ust entries call `registerProvider(...)` or `registerUser(...)` to register the application.

Note:

For communication between host and target only steps 1-3 are different. The rest of the procedure is same as interacting with local device.

1.7 Applications for Tx/Rx ASN.1 encoded messages

It supports Basic Safety Message (BSM), Probe Vehicle Data (PVD), Road Side Alert (RSA) applications . These applications transmit & receive the messages using WSM protocol. The message payload is always encoded in the ASN.1 format referred to as BER-DER.

This section describes the process of Transmitting and Receiving the basicSaftyMessage/ probeVehicalData/ roadSideAlert in either application mode as USER or PROVIDER.

1. If WAVE application mode is USER, Invoke the WAVE Device by calling `invokeWAVEDevice().`
2. If WAVE application mode is PROVIDER, Invoke the WAVE Device by calling `invokeWAVEDriver()` and register the WME, WRSS, TsfTimer indication handling function by calling respective register function calls (As per Initialization functions clause in 1.3.1)
   - Advertise the timing(TA) request through `transmitTA().`

Further application functionality is done by three threads, each for rx, tx, wrssi clients.
For example autowbss_tx_rx_wrss.c.

1. In rx_client: as per the WAVE app mode (USER/PROVIDER) registration process will happen by calling either `registerUser()` or `registerProvider().`
   - rx_client task will receive the WSM packets through `rxWSMMessage().`
   - *RxWSMMessage()* function receives the packets by calling `rxWSMPacket().` Received packet which will be in ASN encoded format is decoded by generic BER decoder , i.e.. decodeMessage().

2. In tx_client: as per the msgType (bsm/pvd/rsa) tx_client task built the respective message request packet of type BasicSafetyMessage_t / ProbeVehicleData_t / RoadSideAlert_t and then encode that packet into ASN encoded format with the help of Canonical DER encoder, i.e. der_encode_to_buffer().
   - der_encode_to_buffer() decodes the data into the provided buffer.
   - Encoded packets are transmitted by calling txWSMPacket().

3. wrssi_client, periodically receive the WRSS report by calling getWRSSReport(). Based on the WRSS value application appropriate actions.

Compiling the application: To compile the application statically include wavelogger.o and wavegps.o files to your application. Also include the pthreads library. For example,

```
$ mips-linux-uclibc-gcc -lpthread -o executable yourapp.o
wavelogger.o wavegps.o
```

1.8 Logging GPS Data

The logging feature enables you to store the received GPS data. The steps explained in this section explains how to build an application that logs GPS packets to a file.

1.8.1 Settingup the logger

1. Set up the logging mode to WSMP by the call:

   ```
   set_logging_mode(MODE);
   ```

Note:

The MODE must be 1 if *xmitgpswave* and *wsmpdemo* are invoked with udp and 2 if they are invoked with
IP.

---

2. Set up the log file:

   ```
   set_logfile("myfile");
   ```

3. Set the address where to listen for forwarded packets:

   ```
   void set_logging_addr(struct sockaddr_in ip,
   uint16_t port)
   ```

   where IP and PORT are as provided to the *wsmpdemo* in above step.

4. Set the logging format by calling `void set_logging_format(int format),` where format = 0 if the logs are to be generated in default Arada format, 1 for XML and 2 for CSV. For XML format the filename specified in Step 2 will be suffixed with .xml and for CSV with a .csv.

1.8.2 Logging the Packets

1. Call void *open_log()* to open the log file in the specified format.
2. Call *start_logging();*
3. Do a *while(1)* to keep logging.

Note:

The contents of the logfile are over-written every time logging starts t

---

1.8.3 Stop Logging

1. Call *stop_loggging()* to stop listening for forwarded packets.
2. Call *close_log()* to close the log file.

1.8.4 Compiling your application

After including neccessary header files in your code, the compilation will look simillar to:

```
$ mips-linux-uclibc-gcc -lpthread -o yourapp yourapp.o
        wavelogger.o wavegps.o nmea_decode.o
```

1.8.5 Attributes of received GPS packes

1. The packets forwarded by xmitgpswave are first parsed by the wavelogger before being appended to the logfile. Each entry is of the form

   ```
   [BEGIN] <attrib=value>* [END]
   ```

2. The attrib, value pairs depend on the type and contents of the received packet. All entries however have the following attribs:

   - <seq=int> :The sequence number of the log entry begining from 1, since the call to start_logging()
   - <logtime=int>: The log time expressed in seconds, begining 1970 (check ur system mannual)
   - <src=string(mac_address)> :The mac address of the orignal transmitter (not the forwarder)
   - <packet=string(packet type)>: The packet type is one of the following gps_ip or gps_udp for this case. The other values may be gps_wsmp and wsmp.

   If packet= gps_ip or gps_udp (also in case of gps_wsmp) the following attribs are also present

   - <packetnum=uint> : The packet number at the transmitter.
   - <rssi=uint> :The RSSI at the forwarder.

   For GPS packets (gps_ip, gps_udp and gps_wsmp) the following fields are present depending on the contents.

   - <gpsstring=string>: The data string as transmitted if the TX side config file has GPS_STR.
   - <gpstime=double> :The UTC time in seconds.
   - <latitude=double> , <longitude=double>, <altitude=double> : The postional information
     <speed=double> , <direction=double>: The velocity information <hdop=double> , <vdop=double> : The horizontal and vertical dilution of precision <hee=double> , <vee=double> : The horizontal and vertical errors of estimate <nsv=uint> : The number of staellites in view <fix=uint>: The fix quality indicator.

   If the packet is gps_wsmp or wsmp the following attribs are also present:

   - <psid=uint>: The PSID
   - <ver=uint>: The version information
   - <sec=uint>: The security information

- <channel=uint>, <rateindex=uint>, <txpower=uint>: The channel, rateindex and txpower information.
- <data=string>: The payload of the WSM packet. For gps_wsmp the data attrib is GPSDATA <data=GPSDATA>. If the data (for gps_wsmp) is in string format the attrib gpsstring is present, as described above.

1.8.6 Example Log in XML Format

```
<XMLLOG>

<logentry>

<loginfo>

<seq> 1 </seq> <logtime> <seconds> 1300451291 </seconds> <microseconds>
854351 </microseconds> </logtime> <src> 00:26:ad:01:20:4d </src>

</loginfo>

<packet>

<type> gps_wsmp </type>

<wsmp>

<header>

<psid> 9 </psid> <ver> 1 </ver> <sec> 0 </sec> <channel> 255 </channel>
<rateindex> 3 </rateindex> <txpower> 15 </txpower>

</header>

<data> GPSDATA </data>

</wsmp>

<nodeinfo>

<packetnum> 111 </packetnum> <rssi> 188 </rssi>

</nodeinfo>

<gps>

<gpstime> 1177501002.000000 </gpstime> <latitude> 12.957100 </latitude>
<latdir> N </latdir> <longitude> 77.605948 </longitude> <londir> E </londir>
<altitude> 944.900000 </altitude> <speed> 0.320000 </speed> <direction> 0.000000
</direction> <hdop> 1.460000 </hdop> <vdop> 2.780000 </vdop> <nsv> 9 </nsv> <fix>
1 </fix> <gpstow> 432000.000000 </gpstow>

</gps>

</packet>

</logentry>

...

</XMLLOG>
```

1.8.7 Example Log in CSV Format

```
gps_wsmp,  9, "", 7, 8, 255, 3, 15, "GPSDATA", 1, 1300451356, 1300451356,
00:26:ad:01:20:4d, 24059, 188, 1177501103.000000, 12.957050, N, 77.606007, E,
966.500000, 0, 0, 1.000000, 2.620000, nan, nan, 8, 1, 432000.000000, "end"

 gps_wsmp,  9, "", 7, 8, 255, 3, 15, "GPSDATA", 2, 1300451357, 1160202108,
00:26:ad:01:20:4d, 24059, 188, 1177501103.000000, 12.957050, N, 77.606007, E,
966.500000, 0, 0, 1.000000, 2.620000, nan, nan, 8, 1, 432000.000000, "end"

 gps_wsmp,  9, "", 7, 8, 255, 3, 15, "GPSDATA", 3, 1300451358, 1161202856,
00:26:ad:01:20:4d, 24059, 188, 1177501103.000000, 12.957050, N, 77.606007, E,
966.500000, 0, 0, 1.000000, 2.620000, nan, nan, 8, 1, 432000.000000, "end"
```

1.9 Logging WSMP Packets

This section explains how to use wavelogger to LOG wsmp packets. This documents assumes that the developer is familliar with LIBWAVE.

1.9.1 Receiving WSM Packets

Local Device

The packets for a registered application are pulled from the device using the libwave call `rxWSMPacket(int pid, WSMIndication rxpkt);`

Remote Device

The packets for the application are pushed by libwave by calling a function with the prototype

```
void myrcvfunction(WSMIndication *rxpkt);
```

The above function has to be registered with the libwave with a call to `receiveWSMIndication(...);`

Logging normal WSMP packets (psid != 9)

1. On obtaining a rxpkt either by a executing `rxWSMPacket(...)` or in `myrcvfunction(...)` check if the packet is a normal packet (whose psid is not 9)

2. Allocate a char array of big enough to store the message, and an integer to store its length;

```
int len;
char logbuf[BIGENOUGH];
```

3. Fit memcpy(&addwsmp.packetnum...) in a single line.

4. Call the wavelogger function that parses the packet and builds the log file entry as follows:

```
len = build_gps_logentry(0, logbuf, rkpkt, NULL, NULL, 0);
```

//rxpkt is a pointer to WSMIndication Structure.

len stores the length of the built entry, it return -1 on error.

5. Now write the log entry to the file.

```
write_logentry(buf,len);
```

Logging GPS WSMP packets (psid == 9)

1. On obtaining a rxpkt either by a executing `rxWSMPacket(...) or in myrcvfunction(...)` check if the packet is a GPS WSMP packet (whose psid is 9)
2. Allocate a char array of big enough to store the message, and an integer to store its length;

```
int len;
char logbuf[BIGENOUGH];
```

3. Fit `memcpy(&addwsmp.packetnum...)` in a single line.
4. Allocate a structure of type additionalWSMP and type GPSData as follows:

```
additionalWSMP addwsmp;

GPSData rxgpsdata;
```

5. Extract the node information from the wsm data as follows:

```
memcpy(&addwsmp.packetnum, rxpkt.data.contents, 4);
memcpy(&addwsmp .rssi, rxpkt.data.contents + 4,1);


memcpy(addwsmp.macaddr, rxpkt.data.contents +5, 6);
//macaddr is an array so no need of an &
```

6. Parse the GPS data using the wavegps function as follows.

```
parseGPSBinData(&rxgpsdata, rxpkt.data.contents + 11,
rxpkt.data.length - 11);    //Skip first 11 bytes
```

7. Build the GPSWSMP log entry

```
len = build_gps_logentry(0, buf, rkpkt, &addwsmp,
&rxgpsdata, get_gps_contents());
```

   The wavegps function get_gps_contents() returns an integer flag that describes the conetents of the most recent parsed data.

8. Write the logentry as before using `write_logentry(buf, len);`

Closing the log file at the end of logging

1. Call `close_log();`

1.10 Logging BSM/PVD/RSA Messages

1. On obtaining a rxmsg check if the packet is of type BSM/PVD/RSA by calling a function as :

```
rxWSMIdentity(&rxmsg,Content_type);
```

2. Allocate a char array of big enough to store the message, and an integer to store its length;

```
int len;
char logbuf[BIGENOUGH];
```

3. Allocate pointers to point structures of type BasicSafetyMessage_t, ProbeVehicleData_t & RoadSideAlert_t for BSA,PVD & RSA respectively and allocate a void pointer to point logData as follows:

```
BasicSafetyMessage_t *bsmLog;

ProbeVehicleData_t *pvdLog;

RoadSideAlert_t *rsaLog;

Void *logData;
```

4. Extract the information from the received data as follows :

```
memcpy(&rxpkt.data.contents,&send_buff_Inter,recv_size);
```

5. Assign the data to respective structures as follows :

```
// For BSM messages
bsmLog = (BasicSafetyMessage_t *)rxmsg.structure;
logData = (void *)(bsmLog->status->fullPos);

// For PVD messages
pvdLog = (ProbeVehicleData_t *)rxmsg.structure;
logData = (void *)(&pvdLog->startVector);

// For RSA messages
rsaLog = (RoadSideAlert_t *)rxmsg.structure;
logData = (void *)(rsaLog->position);
```

6. Build the logentry as follows :

```
len = AsnLog(pnum, rxmsg.type, logformat, logbuf, logData,
&rxpkt);
```

7. Write the logentry as follows :

```
ret = write_logentry(logbuf, len);
```

1.11 WSMP Safety Supplement(WSMP-S)

The purpose of WSMP-S is to provide information to remote devices about the channel switching operation of the local device. This potentially allows the remote devices to make choices concerning the transmit channel and timing of their safety WSMs to maximize channel capacity while avoiding missed messages due to channel switching. Note that only the sending side processing and information formats are specified at this time. The WAVE short message information passed to the higher layer at the receiving side is the same regardless of whether WSMP-S is used or not.

WSMP-S adds control information to the short messages transferred via WSMP.

1.11.1 WSM-Transmission

On receipt of WSMS-WaveShortMessage.request from a higher layer, WSMP-S shall construct the WSMP-
S Control field and prepend it to the Payload parameter to form the WSMData parameter of a WSM-

WaveShortMessage.request primitive, which it sends to WSMP.

1.11.2 WSM-Reception

On receipt of WSM-WaveShortMessage.indication from WSMP, WSMP-S shall remove the WSMP-S Control field from the Data parameter. The remaining Payload is passed to the destination higher layer entity via a WSMS-WaveShortMessage.indication, with the destination higher layer entity determined from the WSMP header ProviderServiceIdentifier and the MIB WsmServiceRequestTable.

1.11.3 Enabling WSMP-S

To enable WSMP-S service enable wsmps flag of the following structure.
struct wsm_request {

u_int32_t psid;

struct channelinfo chaninfo;

u_int8_t version;

u_int8_t security;

u_int8_t txpriority;

u_int8_t wsmps;

u_int64_t expirytime;

WSMData data;

u_int8_t macaddr[IEEE80211_ADDR_LEN];

} __attribute__((__packed__));

1.12 Bluetooth APIs

This section explains how to use bluetooth APIs to send and receive wsmp packets using blueZ stack. This documents assumes that the developer is familliar with LIBWAVE.

1.12.1 Get Bluetooth Adapter ID

Local Bluetooth adapters are assigned identifying numbers starting with 0, and a program must specify which adapter to use when allocating system resources. Use the following API to get the Bluetooth adapter ID.

int hci_get_route(bdaddr_t *bdaddr);

Usually, there is only one adapter or it doesn't matter which one is used, so passing `NULL` to *`hci_get_route()`* will retrieve the resource number of the first available Bluetooth adapter.

1.12.2 Get Bluetooth Device Information

The following function call will return the local Bluetooth device information into the structure-

```
struct hci_dev_info {

        uint16_t dev_id;
        char     name[8];
        bdaddr_t bdaddr;
        uint32_t flags;
        uint8_t  type;
        uint8_t  features[8];
        uint32_t pkt_type;
        uint32_t link_policy;
        uint32_t link_mode;
        uint16_t acl_mtu;
        uint16_t acl_pkts;
        uint16_t sco_mtu;
        uint16_t sco_pkts;
        struct   hci_dev_stats stat;
};
```

The function syntax is :-

```
int hci_devinfo(int dev_id, struct hci_dev_info *dev_info);
```

1.12.3 Scan Nearby Bluetooth Devices

After choosing the local Bluetooth adapter to use and allocating system resources, the program is ready to scan for nearby Bluetooth devices. *`hci_inquiry()`* performs a Bluetooth device discovery and returns a list of detected devices and some basic information about them in following structure.

```
typedef struct {
        bdaddr_t        bdaddr;
        uint8_t         pscan_rep_mode;
        uint8_t         pscan_period_mode;
        uint8_t         pscan_mode;
        uint8_t         dev_class[3];
        uint16_t        clock_offset;
}__attribute__ ((packed)) inquiry_info;
```

Function syntax :-

```
int hci_inquiry(int dev_id, int len, int max_rsp, const uint8_t *lap, inquiry **ii, long flags);
```

`hci_inquiry()` function requires the use of a resource number instead of an open socket, so we use the *`dev_id`* returned by *`hci_get_route()`*. The inquiry lasts for at most 1.28 * *`len`* seconds, and at most *`max_rsp`* devices will be returned in the output parameter *`ii`*, which must be large enough to accommodate *`max_rsp`* results.

1.12.4 Opening the Local Bluetooth Device

Most Bluetooth operations require the use of an open socket. *`hci_open_dev()`* is a convenience function that opens a Bluetooth socket with the specified resource number.

Function syntax :-

```
int hci_open_dev(int dev_id);
```

The socket opened by *hci_open_dev()* represents a connection to the specified local Bluetooth adapter, and not a connection to a remote Bluetooth device.

1.12.5 Get Bluetooth Device Name

Once a list of nearby Bluetooth devices and their addresses has been found, the program determines the user-friendly names associated with those addresses and presents them to the user. The function used is -

Function syntax :-

```
int hci_read_remote_name(int sock, const bdaddr_t *ba, int len, char *name, int timeout);
```

hci_read_remote_name() tries for at most *timeout* milliseconds to use the socket *sock* to query the user-friendly name of the device with Bluetooth address *ba.* On success, *hci_read_remote_name()* returns 0 and copies at most the first *len* bytes of the device's user-friendly name into *name.* On failure, it returns -1 and sets *errno* accordingly.

1.12.6 Searching a Bluetooth Device for a Service

To detect the services on the remote device Service Discovery Protocol(SDP) is used.

1.12.6.1 Connect to SDP server running on Remote Machine

The function used to connect to sdp server on the remote device is -

```
sdp_session_t *sdp_connect(const bdaddr_t *src, const bdaddr_t *target,  uint32_t flags);
```

where 'target' is the bluetooth address of the remote device. You can obtain it using the above *hci_read_remote_name()* function.

Specify the UUID of the application you're searching for. The uuid_t data type is used to represent the 128-bit UUID that identifies the desired service. To obtain a valid *uuid_t* create an array of 16 8-bit integers and use the *sdp_uuid128_create* function, which is similar to the *str2ba* function for converting strings to *baddr_t* types.

e.g. - For a UUID 0xABCD, you can follow the following steps :-

```
uint8_t svc_uuid_int[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xab, 0xcd};

uuid_t svc_uuid;

sdp_list_t *search_list;

sdp_uuid128_create( &svc_uuid, &svc_uuid_int );
search_list = sdp_list_append( NULL, &svc_uuid );
```

Since C does not have a built in linked-list data structure, and SDP search criteria and search results are essentially nothing but lists of data, the BlueZ developers wrote their own linked list data structure and called it *sdp_list_t* as :-

```
typedef struct _sdp_list_t {
        struct _sdp_list_t *next;
        void *data;
```

} sdp_list_t;

Specify that we want a list of all the matching applications attributes.

e.g.-

uint32_t range = 0x0000ffff;
sdp_list_t *attrid_list;

attrid_list = sdp_list_append(NULL, &svc_uuid);

It's easiest to use the magic number *0x0000ffff* to request a list of all the attributes describing the service, although it is possible, for example, to request only the name of a matching service and not its protocol information.

Get a list of service records that have UUID 0xabcd using the following function.

int sdp_service_search_attr_req(sdp_session_t * session, const sdp_list_t *search_list, sdp_attrreq_type_t reqtype, const sdp_list_t *attrid_list, sdp_list_t **response_list);

sdp_service_search_attr_req() searches the connected device for the desired service and requests a list of attributes specified by *attrid_list.*

1.12.6.2 Parsing and Interpreting an SDP search result

Get a list of the protocol sequences using a function -

int sdp_get_access_protos(const sdp_record_t *rec, sdp_list_t **protos);

The *sdp_record_t* data structure represents a single service record being advertised by another device. *sdp_get_access_protos* is used to extract a list of the protocols for the service record.

typedef struct {
uint32_t handle;
sdp_list_t *pattern;
sdp_list_t *attrlist;
} sdp_record_t;

A list and record must be deallocated using the following functions.

void sdp_list_free(sdp_list_t *list, sdp_free_func_t f); and

void sdp_record_free(sdp_record_t *rec);

1.12.7 Communicating with Remote Device

If the specified device found with required service, then use socket connection to communicate with the remote device.

2 Sample Applications

2.1 The Header File

The header file may contain all #defines, data structures & enumerations, function declarations.

For example here is some part of wave.h.

1) #defines :

```
#define OCTET_MAX_LENGTH 32
#define HALFK 1300
#define FOURK 4096
#define SIXTYFOURK 65000
#define IEEE80211_ADDR_LEN 6
#define ACM_COPY(_a, _b) memcpy((_a), (_b), sizeof(ACM)
#define IEEE80211_RATE_MAXSIZE 15
#define MACADDRSIZE 17
#define HASHSIZE 32

#define BUFSIZE 600
#define WAVE_UDPSERVER_PORT    9999
#define WAVE_UDPSERVER_IP      "127.0.0.1"
#define WME_APPIND_PORT        10023
```

2) Data Structures & enumerations :

```
typedef struct {

        u_int8_t acid;
        ACM acm;
        u_int8_t macaddr[IEEE80211_ADDR_LEN];
        struct in6_addr ipaddr;
        u_int8_t persistence;
} WMEApplicationIndication;


typedef struct wrssrequest_indication {

        u_int8_t macaddr[IEEE80211_ADDR_LEN];
        u_int8_t dialogtoken;
        WMEWRSSReport wrssreport;
} WMEWRSSRequestIndication;


struct wsm_request {

        struct channelinfo chaninfo;
        u_int8_t version;
        u_int8_t security;
        u_int32_t psid;
        u_int8_t txpriority;
        u_int64_t expirytime;
        WSMData data;
        u_int8_t macaddr[IEEE80211_ADDR_LEN];

};


typedef struct wsm_request WSMRequest;
typedef struct {
        u_int8_t aci;
        u_int8_t channel;
```

```
        } WMECancelTxRequest;


        typedef struct {
                u_int32_t psid;
                char  acf[OCTET_MAX_LENGTH];
                u_int8_t priority;
                u_int8_t requestType;
                u_int8_t macaddr[IEEE80211_ADDR_LEN];
                u_int8_t repeats;
                u_int8_t persistence;
                u_int8_t channel;
                u_int16_t localserviceid;
                u_int8_t action;
                u_int8_t dstmac[6];
                u_int8_t wsatype;
                u_int8_t channelaccess;
                u_int8_t repeatrate;
                u_int8_t ipservice;
                struct in6_addr ipv6addr;
                u_int16_t serviceport;
                u_int8_t rcpithres;
                u_int8_t wsacountthres;
                u_int8_t userreqtype;
                struct ieee80211_scan_ssid ssid;
                u_int8_t linkquality;
                u_int8_t schaccess;
                u_int16_t schextaccess;
                u_int16_t notif_port;
        } WMEApplicationRequest;


        typedef struct  {
                        char            nmea[GPS_STRSIZE];
                        double          time;
                        double          local_tod;
                        uint64_t        local_tsf;
                        double          latitude;
                        char            latdir;
                        double          longitude;
                        char            longdir;
                        double          altitude;
                        char             altunit;
                        double          course;
                        double          speed;
                        double          climb;
                        double          tee;
                        double          hee;
                        double          vee;
                        double          cee;
                        double          see;
                        double          clee;
                        double          hdop;
                        double          vdop;
                        u_int8_t        numsats;
                        u_int8_t        fix;
                                double          tow;
                        int              date;

        } GPSData;

        struct wrss_report {
                        u_int8_t    channel;
                        u_int64_t measurementTime;
                        u_int8_t  wrss;
                } __attribute__((__packed__));

        typedef struct wrss_report WMEWRSSReport;

        typedef struct {
                uint8_t action;
                uint8_t repeatrate;
                uint8_t channel;
                uint8_t channelinterval;
```

```
        uint8_t servicepriority;
     } __attribute__((__packed__)) WMETARequest;

struct wsm_packet
{
        u_int8_t version;
        u_int8_t security;
        u_int8_t channel;
        u_int8_t rate;
        u_int8_t txpower;
        u_int8_t app_class;
        ACM acm;
        WSMData data;
} __attribute__((__packed__));

typedef struct wsm_packet WSMPacket;
```

3) Function Declarations :

```
int setWMEApplRegNotifParams(WMEApplicationRequest *req);

int registerProvider(int pid, WMEApplicationRequest *appreq);

int removeProvider(int pid, WMEApplicationRequest *appreq);

int transmitTA(WMETARequest *tareq);

int registerUser(int pid, WMEApplicationRequest *appreq);

int removeUser(int pid, WMEApplicationRequest *appreq);

int getWRSSReport(int pid, WMEWRSSRequest *req);

int txWSMPacket(int pid, WSMRequest *req);

int rxWSMPacket(int pid, WSMIndication *ind);

int cancelTX(int pid, WMECancelTxRequest *req);

int generateWMEApplRequest(WMEApplicationRequest *req);

int generateWMEApplRegRequest(void *req);

int generateWMEApplDelRequest(void);

int generateWMETARequest(WMETARequest *req);

int generateWMEWRSSRequest(WMEWRSSRequest *req);

u_int64_t generatetsfRequest();

int generateWMEApplResponse(WMEApplicationResponse *req);

int generateWSMRequest(WSMRequest *req);

int generateWMECancelTxRequest(WMECancelTxRequest *req);
```

2.2 Sample Application to Transmit Data

Here is a sample code to transmit data.

```
#include <stdio.h>
#include <ctype.h>
```

```
#include <termio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <time.h>
#include <signal.h>
#include "wave.h"

static WMEApplicationRequest wreq;
static WMEApplicationRequest entry;
static WMETARequest tareq;
static WSMRequest wsmreq;
static WMECancelTxRequest cancelReq;
static int pid;

void    receiveWME_NotifIndication(WMENotificationIndication *wmeindication);
void    receiveWRSS_Indication(WMEWRSSRequestIndication *wrssindication);
void    receiveTsfTimerIndication(TSFTimer *timer);
int buildPSTEntry();
int buildWSMRequestPacket();
int buildWMEApplicationRequest();
int buildWMETARequest();
int txWSMPPkts(int);
void sig_int(void);
void sig_term(void);
static uint64_t packets;
static uint64_t drops = 0;

        struct ta_argument {
                uint8_t  channel;
                uint8_t channelinterval;
                        } taarg;

int main (int argc, char *argv[])
{
         int result ;
         pid = getpid();
        if (argc < 4)
          {
             printf("usage: localtx [sch channel access <0 - alternating> <1 - continous>]
[TA channel ] [ TA channel interval <1- cch int> <2- sch int>] \n");
             return 0;
        }

        taarg.channel = atoi(argv[2]);
        taarg.channelinterval = atoi(argv[3]);
        printf("Filling Provider Service Table entry %d\n",
                                buildPSTEntry(argv));
        printf("Building a WSM Request Packet %d\n",
                                                buildWSMRequestPacket());
        printf("Building a WME Application  Request %d\n",
                                        buildWMEApplicationRequest());
        printf("Builing TA request %d\n", buildWMETARequest());

       if ( invokeWAVEDriver(0) < 0 )
       {
                printf( "Opening Failed.\n ");
                exit(-1);
        } else
        {
                printf("Driver invoked\n");
        }


        registerWMENotifIndication(receiveWME_NotifIndication);
        registerWRSSIndication(receiveWRSS_Indication);
        registertsfIndication(receiveTsfTimerIndication);
```

```
        printf("starting TA\n");
        if (transmitTA(&tareq) < 0)
        {
            printf("send TA failed\n ");
        }
        else
        {
            printf("send TA successful\n") ;
        }
        printf("Registering provider\n ");
        if ( registerProvider( pid, &entry ) < 0 )
        {
                printf("\nRegister Provider failed\n");
                removeProvider(pid, &entry);
                registerProvider(pid, &entry);
        } else {
                printf("provider registered with PSID = %d\n",
                                            entry.psid );
        }
       result =txWSMPPkts(pid);
       if ( result = 0 )
                printf("All Packets transmitted\n");
        else
                printf("%d Packets dropped\n",result);

        return 1;
}

int buildPSTEntry(char **argv)
{
        entry.psid = 5;
        entry.priority = 1;
        entry.channel = 172;
        entry.repeatrate = 10;
        if (atoi(argv[1]) > 1) {
            printf("channel access set default to alternating access\n");
            entry.channelaccess = 0;
        } else {
            entry.channelaccess = atoi(argv[1]);
        }
        return 1;
}


int buildWSMRequestPacket()
{
        wsmreq.chaninfo.channel = 172;
        wsmreq.chaninfo.rate = 3;
        wsmreq.chaninfo.txpower = 15;
        wsmreq.version = 1;
        wsmreq.security = 1;
        wsmreq.psid = 5;
        wsmreq.txpriority = 1;
        memset ( &wsmreq.data, 0, sizeof( WSMData));
        return 1;
}

int buildWMEApplicationRequest()
{
        wreq.psid =5 ;
        printf(" WME App Req %d \n",wreq.psid);
        wreq.repeats = 1;
        wreq.persistence = 1;
        wreq.channel = 172;
        return 1;
}

int buildWMETARequest()
{
```

```
        tareq.action = TA_ADD;
        tareq.repeatrate = 100;
        tareq.channel = taarg.channel;
        tareq.channelinterval = taarg.channelinterval;
        tareq.servicepriority = 1;
}

int txWSMPPkts(int pid)
{
        int pwrvalues, ratecount, txprio, ret = 0, pktcount, count = 0;
        /* catch control-c and kill signal*/
        signal(SIGINT,(void *)sig_int);
        signal(SIGTERM,(void *)sig_term);
        while(1)
        {
                wsmreq.chaninfo.txpower = 15;
                wsmreq.chaninfo.rate = 1;
                wsmreq.txpriority = 1;
                usleep(2000);
                ret = txWSMPacket(pid, &wsmreq);
                if( ret < 0)
                {
                        drops++;
                }
                else
                {
                        packets++;
                        count++;
                }

                printf("Transmitted #%llu#                                        Dropped #
%llu#\n", packets, drops);
        }
        printf("\n Transmitted =  %d dropped = %d\n",count,drops);
        return drops;
}
void receiveWRSS_Indication(WMEWRSSRequestIndication *wrssindication)
{
        printf("WRSS recv channel %d",(u_int8_t)wrssindication->wrssreport.channel);

        printf("WRSS recv reportt %d",(u_int8_t)wrssindication->wrssreport.wrss);
}

void sig_int(void)
{
        int ret;
        ret = stopWBSS(pid, &wreq);
        removeProvider(pid, &entry);
        signal(SIGINT,SIG_DFL);
        printf("\n\nPackets Sent =  %llu\n",packets);
        printf("Packets Dropped = %llu\n",drops);
        printf("localtx killed by control-C\n");
        exit(0);
}
void sig_term(void)
{
        int ret;
        ret = stopWBSS(pid, &wreq);
        removeProvider(pid, &entry);
        signal(SIGINT,SIG_DFL);
        printf("\n\nPackets Sent =  %llu\n",packets);
        printf("\nPackets Dropped = %llu\n",drops);
        printf("localtx killed by control-C\n");
        exit(0);
}
```

2.3 Sample Application to Receive Data

Here is the sample code to receive data.

```
#include <stdio.h>
#include <ctype.h>
#include <termio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <time.h>
#include <signal.h>
#include "wave.h"

void sig_int(void);
void sig_term(void);
static int pid ;
static WMEApplicationRequest entry;
static  uint64_t count = 0, blank = 0;
void set_args( void * ,void *, int );
enum { ADDR_MAC = 0, UINT8 };

struct arguments{
    u_int8_t macaddr[ 17 ];
    u_int8_t channel;
};
int main(int arg, char *argv[])
{
        WSMIndication rxpkt;
        int i, attempts = 10, drops = 0, result;
        int  ret = 0;
        struct arguments arg1;
        if (arg < 4) {
            printf("usage: localrx [user req type<1-auto> <2-unconditional> <3-none>] [imm
access] [extended access] [channel <optional>] [PROVIDER MAC <optional>]\n");
            return 0;
        }
        pid = getpid();
        memset(&entry, 0 , sizeof(WMEApplicationRequest));
        entry.psid = 5;
        if ((atoi(argv[1]) > USER_REQ_SCH_ACCESS_NONE) || (atoi(argv[1]) <
USER_REQ_SCH_ACCESS_AUTO)) {
            printf("User request type invalid: setting default to auto\n");
            entry.userreqtype = USER_REQ_SCH_ACCESS_AUTO;
        } else {
            entry.userreqtype = atoi(argv[1]);
        }
        if (entry.userreqtype == USER_REQ_SCH_ACCESS_AUTO_UNCONDITIONAL) {
            if (arg < 5) {
               printf("channel needed for unconditional access\n");
                return 0;
            } else {
                entry.channel = atoi(argv[4]);
            }
      }
           entry.schaccess  = atoi(argv[2]);
        entry.schextaccess = atoi(argv[3]);
        if (arg > 5) {
            strncpy(arg1.macaddr, argv[4], 17);
            set_args(entry.macaddr, &arg1, ADDR_MAC);
        }
        printf("Invoking WAVE driver \n");
        if (invokeWAVEDevice(WAVEDEVICE_LOCAL, 0) < 0)
        {
```

```
                printf("Open Failed. Quitting\n");
                 exit(-1);
        }
        printf("Registering User %d\n", entry.psid);
        if ( registerUser(pid, &entry) < 0)
        {
              printf("Register User Failed \n");
              printf("Removing user if already present  %d\n", !removeUser(pid, &entry));
              printf("USER Registered %d with PSID =%d \n", registerUser(pid, &entry),
entry.psid );
        }
        /* catch control-c and kill signal*/
        signal(SIGINT,(void *)sig_int);
        signal(SIGTERM,(void *)sig_term);
           while(1)  {
                ret = rxWSMPacket(pid, &rxpkt);
                if (ret > 0){
                        printf("Received WSMP Packet txpower= %d, rateindex=%d Packet No =#
%llu#\n", rxpkt.chaninfo.txpower, rxpkt.chaninfo.rate, count++);
                } else {
                       blank++;
                }
        }
}

void sig_int(void)
{
            int ret;
           removeUser(pid, &entry);
           signal(SIGINT,SIG_DFL);
           printf("\n\nPackets received = %llu\n", count);
           printf("Blank Poll = %llu\n", blank);
           printf("remoterx killed by kill signal\n");
           exit(0);
}



void sig_term(void)
{
           int ret;
           removeUser(pid, &entry);
           signal(SIGINT,SIG_DFL);
           printf("\n\nPackets received = %llu\n", count);
           printf("Blank Poll = %llu\n", blank);
           printf("remoterx killed by kill signal\n");
           exit(0);
}


int extract_macaddr(u_int8_t *mac, char *str)
{
           int maclen = IEEE80211_ADDR_LEN;
           int len = strlen(str);
           int i = 0, j = 0, octet = 0, digits = 0, ld = 0, rd = 0;
           char num[2];
           u_int8_t tempmac[maclen];
           memset(tempmac, 0, maclen);
           memset(mac, 0, maclen);
           if( (len < (2 * maclen - 1)) || (len > (3 * maclen - 1)) )
        return -1;
     while(i < len)
    {
        j = i;
       while( str[i] != ':' && (i < len) ){
```

```
         i++;
      }
        if(i > len) exit(0);
        digits = i - j;
        if( (digits > 2) ||  (digits < 1) || (octet >= maclen)){
            return -1;
        }
         num[1] = tolower(str[i - 1]);
        num[0] = (digits == 2)?tolower(str[i - 2]) : '0';
        if ( isxdigit(num[0]) && isxdigit(num[1]) ) {
            ld  =  (isalpha(num[0]))? 10 + num[0] - 'a' : num[0] - '0';
            rd  =  (isalpha(num[1]))? 10 + num[1] - 'a' : num[1] - '0';
            tempmac[octet++] =  ld * 16 + rd ;
        } else {
            return -1;
        }
        i++;
    }
           if(octet > maclen)
           return -1;
    memcpy(mac, tempmac, maclen);
    return 0;
}

void set_args( void *data ,void *argname, int datatype )
{
    u_int8_t string[1000];
    int i;
    int temp = 0;
    u_int8_t temp8 = 0;
    struct arguments *argument1;
    argument1 = ( struct arguments *)argname;
    switch(datatype) {
        case ADDR_MAC:
            memcpy(string, argument1->macaddr, 17);
            string[17] = '\0';
            if(extract_macaddr( data, string) < 0 )
            {
                printf("invalid address\n");
            }
            break;
        case UINT8:
                memcpy( data, (char *)argname, sizeof( u_int8_t));
            break;
}
```

2.4 Sample Application to Transmit Data from a Remote Target

Here is the sample code to transmit data from a remote target.

```
#include "wave.h"
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <ifaddrs.h>
#include <errno.h>
#include <fcntl.h>
static WMEApplicationRequest wreq;
static WMEApplicationRequest entry;
static WSMIndication wsmrxind;
static WSMRequest wsmreq;
```

```
static WMECancelTxRequest cancelReq;
static int devicemode = WAVEDEVICE_REMOTE;
static WMEApplicationRequest aregreq;
static int pid;
static WMETARequest tareq;

void    receiveWME_NotifIndication(WMENotificationIndication *wmeindication);
void    receiveWSMIndication(WSMIndication *wsmindication);
int buildWMETARequest();
int buildPSTEntry();
int buildWSMRequestPacket();
int buildWMETARequest();
int buildWMEApplicationRequest();
int txWSMPPkts(int);
static uint64_t packets = 0;
static uint64_t drops = 0;
void sig_int(void);
void sig_term(void);

/*This program demonstrates how to start a WBSS on a TARGET device*/
struct ta_argument {
        uint8_t channel;
        uint8_t channelinterval;
} taarg;

int main (int argc, char *argv[])
{
        int ret;
        int blockflag = 0 ;
#ifdef  WIN32
                char szHostName[255];
                char *szLocalIP;
                struct hostent *host_entry;
                WIN_SOCK_DLL_INVOKE
//    This is required to invoke DLLs for Sockets in WIN32              //
               gethostname(szHostName, 255);
#endif
        pid = getpid();
       if(argc < 5) {
                printf("usage: remotetx <TARGETIP> [sch-channel access <0-alternating> <1-
continous> [TA Channel ] [TA Channel Interval <1-cch int> <2-sch int> ]\n");
               exit(-1);
        }
            taarg.channel = atoi(argv[3]);
            taarg.channelinterval = atoi(argv[4]);
        /*Initialize the data structures*/
        printf("Filling Provider Service Table entry %d\n", buildPSTEntry() );
        printf("Building a WSM Request Packet %d\n", buildWSMRequestPacket() );
        printf("Building a WME Application  Request %d\n", buildWMEApplicationRequest() );
        printf("Building TA Request %d\n", buildWMETARequest() );
        /*Provide the IP address of the TARGET WAVE-device*/
        setRemoteDeviceIP(argv[1]);
        devicemode = WAVEDEVICE_REMOTE;
        ret = invokeWAVEDevice(devicemode, blockflag); /*blockflag is ignored in this case*/
        if (ret < 0 )
        {
                /*Error*/
        }
else {
                printf("Driver invoked\n");
        }
#if 0
        /*Get the IP address of eth0*/
        sfd = socket(AF_INET6, SOCK_STREAM, 0);
        if(sfd >= 0) {
```

```
              memset(&ifaddr, 0, sizeof(ifaddr));
#ifdef WIN32
          if((host_entry=gethostbyname(szHostName))!=NULL){
                  szLocalIP = inet_ntoa (*(struct in_addr *)*host_entry->h_addr_list);
                  sin->sin_addr.s_addr = inet_addr(szLocalIP);
                  aregreq.ipv6addr = sin->sin_addr;
                  entry.ipaddr = sin->sin_addr;
                }
#else
            if( getifaddrs(&ifaddr) == 0 )
             {
                 for( ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next )
                  {
                      if( ifa->ifa_addr == NULL ) continue;
                        if( ( ifa->ifa_flags & IFF_UP ) == NULL ) continue;
                            if(ifa->ifa_addr->sa_family == AF_INET )
                             {
                                sin4 = (struct sockaddr_in *)(ifa->ifa_addr);
                                     if(inet_ntop(ifa->ifa_addr->sa_family,(void *)&(sin4-
>sin_addr), str, sizeof(str)) == NULL )
                                       {
                                         printf("IPV4 Interface = %s: inet_ntop failed\n",
ifa->ifa_name );
                                       }
                                    else
                                    {
                                       inet_pton( AF_INET, argv[1], &inaddr );
                                       printf("IPV4 Interface =%s      %s\n",ifa->ifa_name,
str );
                                    }
                                }
             else if( ifa->ifa_addr->sa_family = AF_INET6 )
                                {
                                        sin6 = (struct sockaddr_in6 *)(ifa->ifa_addr);
                                        if(inet_ntop(ifa->ifa_addr->sa_family,(void
*)&(sin6->sin6_addr), str, sizeof(str)) == NULL )
                                        {
                                               printf("IPV6 Interface = %s: inet_ntop
failed\n", ifa->ifa_name );
                                        }
                                        else
                                        {
                                                inet_pton( AF_INET6, argv[1], &inaddr );
                                                printf("IPV6 Interface =%s      %s\n",ifa-
>ifa_name, str );
                                        }
                                }
                        }
                        aregreq.ipv6addr = sin6->sin6_addr;
                        entry.ipv6addr = sin6->sin6_addr;
                }
                else {
                                ret = inet_pton(AF_INET, argv[1], &inaddr );
                                if( ret != 1 )
                                {
                                        ret = inet_pton(AF_INET6, argv[1], &inaddr );
                                        if( ret != 1 )
                                                perror("inet_pton() failed");
                                }
                                aregreq.ipv6addr = inaddr;
                                entry.ipv6addr = inaddr;
                        }
#endif
        }
#endif
```

```
        getUSTIpv6Addr(&entry.ipv6addr,"eth0");
        aregreq.ipv6addr = entry.ipv6addr;
        /*Register a call back function with LIBWAVE to receive WME Notifications and
WSMIndications from TARGET*/
        registerWMENotifIndication(receiveWME_NotifIndication);
        registerWSMIndication(receiveWSMIndication);
        /*Set the notification IP and PORT*/
        aregreq.notif_port = 6666;
        /*Tell LIBWAVE where to listen for notifications*/
        setWMEApplRegNotifParams(&aregreq);
        if (transmitTA(&tareq) < 0)  {
            printf("send TA failed\n ");
        } else {
            printf("send TA successful\n") ;
        }
       printf("Registering provider\n ");
            /*NOTE:If the TARGET device is not up or the link is down the libwave calls will
wait indefinetly*/
        removeProvider(pid, &entry );
        /*Register a  Provider on the TARGET, Note: Most of the libawave functions are
identical whether the device is local or remote*/
        /*the only difference being that remote calls may hang (when no reply comes from
TARGET) and the way WSMIndications are received*/
        if (registerProvider(pid, &entry ) < 0 ) {
               printf("Register Provider failed\n");
                exit(-1);
        } else {
                printf("Provider registered with PSID = %d \n", entry.psid);
        }
        /*Transmit some packets*/
        ret = txWSMPPkts(pid);
        if (ret == 0 )
               printf("All Packets transmitted\n");
       else
                printf("%d Packets dropped\n", ret);
        return 0;Confidential Page 46 8/29/2012
}


int buildWMETARequest()
{
    tareq.action = TA_ADD;
    tareq.repeatrate = 100;
    tareq.channel = taarg.channel;
    tareq.channelinterval = taarg.channelinterval;
    tareq.servicepriority = 1;
}

/*Fill up the data structure  to register a PROVIDER application*/
int buildPSTEntry(){
        entry.psid = 5;
        entry.priority = 1;
       entry.channel = 172;
        /*This is the Port where WSMIndications will be received*/
        entry.serviceport = 8888;
        entry.repeatrate = 10;
      entry.channelaccess = 0;
        return 0;
}

/*Build a request to transmit a WSM packet*/
int buildWSMRequestPacket()
 {
        wsmreq.chaninfo.channel = 172 ;
```

```
        wsmreq.chaninfo.rate = 3 ;
        wsmreq.chaninfo.txpower = 15 ;
        wsmreq.version = 1 ;
       wsmreq.security = 1 ;
       wsmreq.psid = 5 ;
       wsmreq.txpriority = 1;
        memset ( &wsmreq.data, 0, sizeof( WSMData ));
        return 0;
}

/*Build a request to start a WBSS*/
int  buildWMEApplicationRequest() {
        wreq.psid = 5 ;
        wreq.repeats = 1;
        wreq.persistence = 1;
        /*WRSS Request channel should be same as that of PROVIDER*/
        wreq.channel = 172;
        return 1;
}

/*Transmit the packets here*/
int txWSMPPkts(int pid) {
               int pwrvalues;
               int ratecount;
              int txprio;
               int ret = 0, pktcount, count = 0;
               unsigned char mac[6] =  { 0x00, 0x03, 0x7f, 0x07, 0x81, 0x8b};
               printf("Transmiting...\n");
        /* catch control-c and kill signal*/
        signal(SIGINT,(void *)sig_int);
        signal(SIGTERM,(void *)sig_term);
        while (1) {
              wsmreq.chaninfo.txpower = 15;
               wsmreq.chaninfo.rate = 1;
               wsmreq.txpriority = 1;
               usleep(2000);
              ret = txWSMPacket(pid, &wsmreq);
              if( ret < 0) {
                       drops++;
               }
               else {
                       packets++;
                       count++;
               }
               printf("Transmitted #%llu#                                        Dropped #
%llu#\n", packets, drops);
        }
        return drops;
}



void sig_int(void)
{
        int ret;
        ret = stopWBSS(pid, &wreq);
        removeProvider(pid, &entry);
        printf("\n\nPACKTES SENT = %llu\n",packets);
        printf("PACKTES DROPPED = %llu\n",drops);
        printf("remotetx killed by control-C\n");
        signal(SIGINT,SIG_DFL);
        exit(0);
}

void sig_term(void)
```

```
{
        int ret;
        ret = stopWBSS(pid, &wreq);
        removeProvider(pid, &entry);
         printf("\n\nPACKTES SENT = %llu\n",packets);
         printf("PACKTES DROPPED = %llu\n",drops);
         printf("remotetx killed by control-C\n");
         signal(SIGINT,SIG_DFL);
        exit(0);
}
```

2.5 Sample Application to Receive Data on a Remote Host

Here is the sample code to receive data on a remote host.

```
#include "wave.h"
#include <stdio.h>
#include <ctype.h>
#include <time.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <ifaddrs.h>

static WMEApplicationRequest wreq;
static WMEApplicationRequest entry;
static WSMIndication wsmrxind;
static int devicemode = WAVEDEVICE_REMOTE;
static WMEApplicationRequest aregreq;
static int pid;
static uint64_t count = 0;

void    receiveWME_NotifIndication(WMENotificationIndication *wmeindication);
void    receiveWSMIndication(WSMIndication *wsmindication);
void sig_int(void);
void sig_term(void);
int buildUSTEntry();
int rxWSMPPkts(int);
int confirmBeforeJoin(WMEApplicationIndication *);
/*This program demonstrates how to recive WSMP packets on a HOST, from a USER registered
on a TARGET device*/

int main (int argc, char *argv[])
{
        int ret;
        char server[255];
        int blockflag = 0 ;
        //struct ifreq ifr;
        WSMIndication rxpkt;
#ifdef  WIN32
           char szHostName[255];
           char *szLocalIP;
           struct hostent *host_entry;
           WIN_SOCK_DLL_INVOKE
           gethostname(szHostName, 255);
#endif
        pid = getpid();
        if(argc < 5) {
```

```
                printf("usage: remoterx <TARGETIP>[user-req type<1-auto> <2-unconditional>
<3-none> ] [imm access ] [extended access ]\n");
                exit(-1);
        }
        setRemoteDeviceIP(argv[1]);
        /*Set the Remote Device IP */
        buildUSTEntry();
        if( ( atoi( argv[2] ) > USER_REQ_SCH_ACCESS_NONE ) || ( atoi(argv[2]) <
USER_REQ_SCH_ACCESS_AUTO ) ) {
                printf("User request type invalid: setting default to auto\n");
                entry.userreqtype = USER_REQ_SCH_ACCESS_AUTO;
        }
        else
        {
                entry.userreqtype = atoi(argv[2]);
       }
        entry.schaccess = atoi(argv[2]);
        entry.schextaccess = atoi(argv[3]);
        devicemode = WAVEDEVICE_REMOTE;
           ret =  invokeWAVEDevice(devicemode, blockflag ); /*blockflag is ignored in this
case*/
        if (ret < 0 ) {
        } else {
                printf("Driver invoked\n");
        }
#if 0
        /*Get the IP of eth0*/
        sfd = socket(AF_INET6, SOCK_STREAM, 0);
        if(sfd >= 0) {
               memset(&ifaddr, 0, sizeof(ifaddr));
#ifdef WIN32
           if((host_entry=gethostbyname(szHostName))!=NULL){
                     szLocalIP = inet_ntoa (*(struct in_addr *)*host_entry->h_addr_list);
                    sin->sin_addr.s_addr = inet_addr(szLocalIP);
                    aregreq.ipv6addr = sin->sin_addr;
                     entry.ipaddr = sin->sin_addr;
                }
#else
                if( getifaddrs(&ifaddr) == 0 )
                {
                  for( ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next )
                  {
                     if( ifa->ifa_addr == NULL ) continue;
                        if( ( ifa->ifa_flags & IFF_UP ) == NULL ) continue;
                             if(ifa->ifa_addr->sa_family == AF_INET )
                                {
                                    sin4 = (struct sockaddr_in *)(ifa->ifa_addr);
                                    if(inet_ntop(ifa->ifa_addr->sa_family,(void *)&(sin4-
>sin_addr), str, sizeof(str)) == NULL )
                                    {
                                        printf("IPV4 Interface = %s: inet_ntop failed\n",
ifa->ifa_name );
                                      }
                                      else
                                      {
                                         inet_pton( AF_INET, argv[1], &inaddr );
                                         printf("IPV4 Interface =%s      %s\n",ifa-
>ifa_name, str );
                                      }
                                       inet_pton(AF_INET6, str, &sin6->sin6_addr );
                                }
                                else if( ifa->ifa_addr->sa_family = AF_INET6 )
                                {
                                    sin6 = (struct sockaddr_in6 *)(ifa->ifa_addr);
```

```
                                    if(inet_ntop(ifa->ifa_addr->sa_family,(void *)&(sin6-
>sin6_addr), str, sizeof(str)) == NULL )
                                     {
                                         printf("IPV6 Interface = %s: inet_ntop failed\n",
ifa->ifa_name );
                                     }
                                else   {
                                                inet_pton( AF_INET6, argv[1], &inaddr );
                                                printf("IPV6 Interface =%s      %s\n",ifa-
>ifa_name, str );
                                        }
                                        inet_pton(AF_INET6, str, &sin6->sin6_addr );
                                   }
                         }
                        aregreq.ipv6addr = sin6->sin6_addr;
                         entry.ipv6addr = sin6->sin6_addr;
                }
                else {
                         ret = inet_pton( AF_INET, argv[1], &inaddr );
                         if( ret != 1 )
                         {
                                 ret = inet_pton( AF_INET6, argv[1], &inaddr );
                                if( ret != 1 )
                                         perror("inet_pton() failed");
                         }
                         aregreq.ipv6addr = inaddr;
                         entry.ipv6addr = inaddr;
               }
#endif
        }
#endif
getUSTIpv6Addr(&entry.ipv6addr,"eth0");
    aregreq.ipv6addr = entry.ipv6addr;
        /*Register a call back function with LIBWAVE to receive WME Notifications and
WSMIndications from TARGET*/
        registerWSMIndication(receiveWSMIndication);
        /*Set the notification and service PORTS*/
       aregreq.notif_port = 9999;
        entry.serviceport = 8888;
        /*Tell LIBWAVE where to listen for notifications*/
        setWMEApplRegNotifParams(&aregreq);
        /*Start recieving packets*/
       printf("Registering User\n");
        if (registerUser(pid, &entry) < 0)
       {
                printf("Register User Failed \n");
                printf("Removing user if already present  %d\n", !removeUser(pid,
&entry));
                printf("USER Registered %d with PSID =%d \n", registerUser(pid, &entry),
entry.psid);
        }
        signal(SIGINT,(void *)sig_int);
        signal(SIGTERM,(void *)sig_term);
        while(1);
        /*On exit its better to call removeUser(pid, &entry)*/
        return 0;
}

int buildUSTEntry()
{
        entry.psid = 5;
        entry.userreqtype = USER_REQ_SCH_ACCESS_AUTO;
        return 0;
}
```

```
/*LIBWAVE calls this function when it receives a WSMP packet (aka WSMIndication) from
TARGET*/
void receiveWSMIndication(WSMIndication *wsmindication)
{
        printf("WSMP Packet received, packet number=%d\n", ++count);
        /*Process your packet here*/
}

int confirmBeforeJoin(WMEApplicationIndication *appind)
{
        printf("\nJoin\n");
        return 1; /*Return 0 to stop Joining the WBSS*/
}
void sig_int(void)
{
        int ret;
        removeUser(pid, &entry);
        signal(SIGINT,SIG_DFL);
        printf("\n\nPackets received = %llu\n", count);
       printf("remoterx killed by kill signal\n");
        exit(0);
}
void sig_term(void)
{
        int ret;
        removeUser(pid, &entry);
        signal(SIGINT,SIG_DFL);
        printf("\n\nPackets received = %llu\n", count);
        printf("remoterx killed by kill signal\n");
        exit(0);
}
```

2.6 Steps for Compiling

1. The release have 3 .tar files i.e. locomate-mips.tar.bz2, locomate-toolchain.tar.bz2 & locomate-x86.tar.bz2. The Locomate-mips.tar.bz2 is for applications on the LOCOMATE while locomate-x86.tar.bz2 is for remote applications on host system(pc or laptop) & locomate-toolchain.tar.bz2 is for compiling applications for LOCOMATE.

2. To extract the contents of these files you have to give following commands on the command line:

```
tar –Pjxvf  locomate-toolchain.tar.bz2

tar –Pjxvf  locomate-mips.tar.bz2

tar –Pjxvf  locomate-x86.tar.bz2
```

3. After executing the first commands toolchain will be extracted to /opt/build/ (make sure that /opt/ have write permissions). After executing the next two commands the sample applications, binaries, include files & libraries required to complile your application will be extracted to /usr/src/locomate-release/ for two different platforms i.e. mips & x86 in two separate directories.

4. The `src` folder is having all the source files(.c files). You can add your own programs to this folder. The `lib` folder is having all the libraries related to LOCOMATE. The `incs` folder is having all the header(.h) files required to compile the `src` folder. The `bin` folder is having the pre-compiled executables of src folder.

5. After adding your application don’t forget to add your program entry in the Makefile. Refer to the Makefile in src folder.

6. For compiling just give “`make`” on command line.

Note:

For x86 applications & compiling applications minimum requirement of libc version is GlibC 2.5 or higher.

2.7 Instructions For Windows Users

1.Extract locomate-x86.tar.bz2 and Go to --> "/usr/src/locomate-release/x86-win/". x86-win conatins all source, include and library files which are required for your compilation.

2.The src folder is having all .c files. The incs folder contains header files which are required to compile the src folder. The lib folder having libraries related to LOCOMATE. The bin folder having all pre-compiled executables of src folder.

3.By changing .c files of src folder and recompile the src files you will get your own executables.

4.To Recompile or make your own applications minimum requirements

a. Microsoft Visual Studio Tool. b. Ipv6 Enabled on Your machine.

5.In Windows XP to enable IPV6 enter this command "netsh int ipv6 install" in CMD and For higher versions than Windows XP by default enabled.

6.Open the visual studio command prompt and cd to x86-win folder. Run Batchfile.bat then all applications will recompile and we will get our new executables in bin folder.